

APPROVAL SHEET

Title of Dissertation: Measuring and Monitoring Technical Debt

Name of Candidate: Yuepu Guo
Doctor of Philosophy, 2016

Dissertation and Abstract Approved: (_____)
Carolyn Seaman
Associate Professor
Department of Information Systems

Date Approved: _____

ABSTRACT

Title of Document: MEASURING AND MONITORING
TECHNICAL DEBT

Yuepu Guo, Ph.D., 2016

Directed By: Associate Professor Carolyn Seaman,
Department of Information Systems

Technical debt is a metaphor for the effects of delayed software development and maintenance tasks. Due to the delay, software may carry immature artifacts in its lifecycle, e.g., immature design, incomplete documentation, etc., which may affect subsequent development and maintenance activities, and so can be seen as a type of debt that the system developers owe the system. Incurring technical debt is common for software projects and can often increase productivity in the short run. However, like paying interest on the debt, such benefit is achieved at the cost of extra work in future. In this sense the technical debt metaphor characterizes the relationship between the short-term benefits of delaying certain software tasks or doing them quickly and less carefully and the long-term cost of those delays or shortcuts. It should be noted that the extra cost may not be realized if the portion of a system containing technical debt will never be touched in the future. Therefore, the problem boils down to managing risk and making informed decisions on which delayed tasks need to be accomplished, and when. Currently, managers and leaders of software maintenance carry out this risk analysis implicitly, if at all. However, on large systems, it is too easy to lose track of delayed tasks or to misunderstand their impact.

The result is often unexpected delays and compromised quality. Therefore identifying, measuring and monitoring technical debt would help managers make informed decisions, resulting in higher quality of maintained software and greater maintenance productivity. The objective of this research work is to contribute to a comprehensive technical debt theory that characterizes the cost and benefit sides of technical debt management. To achieve this objective, we carried out two types of studies – retrospective study and case study – on real software projects. Through decision simulation, the retrospective studies uncovered the benefits of explicit technical debt management. The case studies revealed the costs of technical debt management. The results from the studies provide insights into the technical debt management problem, thus facilitating decision making in software projects and practical application of technical debt management in the software industry.

MEASURING AND MONITORING TECHNICAL DEBT

By

Yuepu Guo

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2016

© Copyright by
Yuepu Guo
2016

Dedication

This dissertation is dedicated to my family in Wenxi, China – my parents, Dongchi Guo and Xuexiang Xi, my sisters, my aunts, uncles, and cousins, who have supported me and believed in me through all these years.

Acknowledgements

I thank my mentor Dr. Carolyn Seaman for her tremendous help, support, and encouragement throughout my Ph.D. program, especially the hardest dissertation phase. I also thank my colleagues – Liwei Dai, Susan Mitchell, Rodrigo Spinola, Nico Zazworka and Tony Zhang who have supported me and helped me with my dissertation in different ways. Lastly, I thank our industry collaborators, including Xerox Corporation, Samsung Brazil, Kali Software, ABB and Fraunhofer for inspiring my dissertation research and hosting the studies on technical debt management.

Table of Contents

Table of Contents	iv
List of Tables	viii
List of Figures	ix
Chapter 1: Introduction	1
1.1 Technical Debt Concept.....	3
1.2 Technical Debt Characteristics	6
1.3 Current Technical Debt Management Research and Practice.....	10
1.4 Research Questions	13
Chapter 2: Related Work	15
2.1 Technical Debt Literature	15
2.1.1 Definition and Classification	17
2.1.2 Causes and Impact	21
2.1.3 Management Approaches.....	24
2.2 Software Risk Management.....	35
2.2.1 Overview	35
2.2.2 Boehm’s Approach	37
2.2.3 SEI’s Approach.....	38
2.2.4 Risk Management Process	40
2.2.5 Risk Analysis	43
2.2.5.1 Risk Estimation.....	43
2.2.5.2 Risk Assessment	47
2.2.6 Empirical Study and Evaluation	52
2.2.7 Summary of Software Risk Management.....	56
2.3 Software Quality Assessment	58
2.3.1 Overview.....	59

2.3.2 Quality Metrics	62
2.3.2.1 Design and Code Metrics.....	63
2.3.2.2 Metric Evaluation.....	67
2.3.3 Program Analysis and Code Smells.....	69
2.4 Software Effort Estimation	71
2.4.1 Software Size Metrics	72
2.4.2 Effort Estimation Techniques	75
2.4.2.1 Algorithmic Methods	76
2.4.2.2 Analogy-based Estimation	80
2.4.2.3 Expert Estimation.....	81
2.4.3 Summary of Cost Estimation.....	84
2.5 Summary of Literature Review.....	85
Chapter 3: Research Overview	88
3.1 The Proposed Technical Debt Management Approach	89
3.1.1 Identifying Technical Debt	92
3.1.2 Measuring Technical Debt.....	94
3.1.3 Decision Making.....	97
3.1.4 Summary of Proposed Approach.....	100
3.2 Research Design.....	102
3.2.1 The Retrospective Studies.....	103
3.2.1.1 Procedure	104
3.2.1.2 Data Collection	109
3.2.1.3 The SMB Study.....	113
3.2.1.4 The Hadoop Study	114
3.2.2 The Case Studies.....	116
3.2.2.1 Procedure	116
3.2.2.2 Data Collection	117
3.2.2.3 The Tranship Case Study	118

3.2.2.4 The EducationHub Study	119
3.3 Summary	121
Chapter 4: The SMB Study.....	123
4.1 Overall Approach.....	123
4.2 The Subject Project and Technical Debt Item	124
4.3 Measurement.....	127
4.4 Data Collection	129
4.5 Data Analysis	132
4.6 Results.....	133
4.7 Conclusion	135
Chapter 5: The Hadoop Study.....	137
5.1 Process	138
5.2 The Subject Project and Classes	139
5.3 Measurement.....	142
5.4 Data Collection	147
5.5 Data Analysis	147
5.6 Discussion.....	153
Chapter 6: The Tranship Study	159
6.1 Subject Project	159
6.2 Study Process	160
6.3 Data Collection	161
6.4 Data Analysis	168
6.5 Results and Findings	169
6.5.1 Costs of Technical Debt Management.....	171
6.5.2 Planning Process and Decision Factors	174
6.5.3 Benefits and Impact	177
6.6 Discussion.....	179
6.6.1 Costs and Benefits of Explicit Technical Debt Management	180

6.6.2 Limitations and Validity Threats	183
Chapter 7: The EducationHub Study	185
7.1 Subject Project	186
7.2 Study Process	187
7.3 Data Collection	190
7.4 Data Analysis	192
7.5 Results and Findings	195
7.5.1 Costs of Managing Technical Debt.....	195
7.5.2 Process Deviation.....	200
7.5.3 Obstacles	202
7.6 Discussion.....	206
7.6.1 Costs and Obstacles	206
7.6.2 Evaluation of Validity.....	209
7.6.3 Limitations	210
Chapter 8: Discussion	212
8.1 Retrospective Studies.....	214
8.2 Case Studies	217
8.3 Addressing the Research Questions.....	226
8.3.1 RQ1: Costs and Benefits	226
8.3.2 RQ2: Decision-making	230
8.4 Contributions.....	231
8.5 Research Implications and Future Work.....	234
Bibliography	236

List of Tables

Table 1. ISO 9126 Quality Characteristics	59
Table 2. Technical Debt Item.....	91
Table 3. Cost and Benefit of the Simulated Decisions	108
Table 4. Total Cost of the Decision Alternatives.....	108
Table 5. Data Descriptions.....	112
Table 6. Effort to Change to ActiveSync (cost X) at D3	130
Table 7. Effort Estimate to Change to ActiveSync at D1 (P).....	131
Table 8. Interest Amount and Change Probability Estimates.....	132
Table 9. Class dfs.DFSck for Decision Simulation	148
Table 10. Class dfs.DFSCClient for Decision Simulation	150
Table 11. Class mapred.TextInputFormat for Decision Simulation	151
Table 12. Class mapred.PhasedFileSystem for Decision Simulation	152
Table 13. Baseline Size and Productivity of the Project.....	162
Table 14. Main Codes of the Coding Scheme	169
Table 15. Technical Debt Items by Type.....	170
Table 16. Changes in the Technical Debt List over the Sprints.....	171
Table 17. Costs of Technical Debt Management Activities over the Sprints.....	172
Table 18. Coding Scheme	194
Table 19. Costs of Technical Debt Management.....	199

List of Figures

Figure 1. Technical Debt Quadrants	18
Figure 2. Cumulative Functionality of the Two Design Strategies	28
Figure 3. Cumulative Functionality in Two Scenarios	29
Figure 4. Development Investment Decision Approach.....	30
Figure 5. Cost/Benefit matrix for Nine God Classes	32
Figure 6. Boehm’s Risk Management Approach.....	38
Figure 7. SEI Continuous Risk Management (CRM) Paradigm.....	40
Figure 8. Risk Analysis Matrix.....	47
Figure 9. A Decision Tree.....	49
Figure 10. Release Planning Process	98
Figure 11. An Initial Technical Debt Management Framework.....	101
Figure 12. Timeline of System W’s Events on Technical Debt Item A	105
Figure 13. Timeline of the SMB Evolution	125
Figure 14. Decision Simulation Poces	133
Figure 15. Cost/benefit of T1 at Decision Points.....	134
Figure 16. Change Likelihood and No of Potential Technical Debt Instances.....	141
Figure 17. Code Churn History and Refactoring Decision.....	143
Figure 18. Data Collection Spreadsheet.....	166
Figure 19. Important Project and Case Study Dates.....	187
Figure 20. Technical Debt List Extract.....	191
Figure 21. Axial Coding	195
Figure 22. Cost Pattern of technical debt Management.....	207
Figure 23. Mapping of Cost Categories.....	220

Chapter 1: Introduction

The past few decades have seen an advance in computer technologies, increased expectations of software and a rapid change in the environments where software is applied. All these factors constitute pressure on software evolution [1]. In fact, most software systems have to be modified after delivery to fix bugs, to improve performance or to adapt to new environments. Modification after delivery with these purposes is called software maintenance, an essential part of the software lifecycle.

Maintenance of software systems will be carried out so long as the cost of replacing them with new ones outweighs the cost of modification. For example, it is not worthwhile to re-develop the whole system because of a new required formatting change in the system generated reports. Therefore software systems usually have multiple versions or releases in their lifecycle. The maintenance phase of some large legacy systems may be as long as several decades.

Besides the long time span, software maintenance consumes the majority of the overall life cycle costs [2-4]. A number of management and technical problems, such as staff turnover, budgetary pressure and quality of system documentation, are tied to the high cost and low speed of software maintenance [5]. On one hand, software maintainers are usually not the persons who developed the system originally. It's difficult and time-consuming for them to comprehend the system, often with little support from documentation. Therefore, software maintainers are often lacking in thorough knowledge or understanding of the system, which leads to poor design and implementation of modifications. On the other hand, maintenance is often performed

under tight time and resource constraints. Software maintainers have to focus only on current requirements, and often take shortcuts to save time and effort. In such situations, long-term maintainability is given little attention or completely ignored. As a result, more low quality components emerge, which in turn adds more constraints on future maintenance tasks and makes modification more difficult and unpredictable. Hereby the quality of maintained software diminishes over time, with respect to its internal system structure, adherence to standards, documentation, understandability, etc. [1].

This problem described above has been expressed with a metaphor called “technical debt”, analogous to monetary debt in the finance domain. The problem is not new to the software community. Actually it has been studied for decades from different perspectives and by different people [1, 6, 7]. However, framing this phenomenon as “debt” facilitates communications between technical and business stakeholders. Moreover, with the technical debt metaphor, ideas from the financial domain can be used to inspire new approaches to improving decision making in software maintenance management.

This dissertation research focuses on the technical debt management problem with the objective of uncovering the cost and benefit characteristics of managing technical debt in an explicit manner. To achieve this research objective, we designed two types of studies, each of which is targeted to one side, i.e. cost or benefit, of technical debt management. A total of four studies have been carried out, which are presented from Chapter 4 to Chapter 7 respectively. In the following subsections, we first introduce

the technical debt concept, how it evolved and the characteristics of technical debt compared to its financial counterpart. Then we discuss the current technical debt management research and practice, which leads to the motivation of this dissertation research. Based on the discussion, we introduce the research questions in the end of this chapter.

1.1 Technical Debt Concept

Technical debt is a metaphor for the effects of delayed software development and maintenance tasks. This metaphor is closely related to software quality. Since technical debt refers to delayed software maintenance tasks, one of the consequences of incurring technical debt is immature artifacts, such as immature design, incomplete documentation or unfinished testing, in the software lifecycle. Immature artifacts are those that currently work correctly in the system, but fail to meet certain quality criteria of software projects. For example, a software system that, over time, becomes highly coupled and contains many redundant modules may be currently functioning properly, but any addition of new functionalities is difficult and requires extra effort to deal with the coupling or redundancy issues. This design is considered immature if extensibility is one of the requirements for the system. Although they do no harm in the current stage, or may even have benefits such as reduced design time in the above example, these immature artifacts may burden software maintenance in the future. Therefore, the immature artifacts should be fixed, modified or changed sooner or later, when the benefit they can bring does not outweigh the cost they incur. In this sense, fixing the immature artifacts remaining in a system is a task left undone and can thus be seen as a type of debt that the system developers owe the system.

The presence of technical debt can be seen as a type of diminished software quality, although not all software quality issues can be characterized as technical debt.

Software quality can be affected by many factors, but a common reason for the quality decline is that maintenance is often performed under tight time and resource constraints, with the minimal amount of effort and time required. Typically, there is a gap between this minimal required amount of work and the amount required to make the modification while also maintaining the level of software quality. The gap represents shortcuts taken by managers or developers, intentionally or not, in order to increase maintenance speed. In the case of intentionally taking shortcuts, software managers make the decision on which modifications must be implemented without compromise, which modifications can be done quickly with minimal effort and time (and thus quality), and which modifications can be deferred. Hereby technical debt is incurred as a strategy for software managers to trade off software quality with productivity, which may have additional benefits such as that gained from earlier time to market than their competitors. However, sometimes technical debt is incurred unintentionally, e.g. by a novice developer who lacks the knowledge and experience to perform the maintenance tasks without compromising quality. In such cases of unintentional debt, the effect remains same as intentional debt because it's likely to take the developer more time and effort to fulfill the maintenance task while still keeping the level of quality. Therefore, for technical debt, intentional or not, the effect is that the maintenance time or cost is reduced in the current stage, i.e., technical debt speeds up software development in the short run. However, this benefit is only one of the effects of technical debt on software maintenance.

Like paying interest on a debt, such benefit is achieved at the cost of extra work in future. Using the above coupling and redundancy example (a software system with highly coupled and redundant modules functions properly at present, but faces the challenge in adding new modules and requires extra effort to deal with the coupling or redundancy issues.), the project may save time and effort in the design of modifications (resulting in more coupling and redundancy), but the high coupling level of the system will incur additional maintenance cost in future. Assuming that four particular modules in this system are highly coupled with each other, when one of those modules is modified to support new functionality, the other three modules are affected and are likely to need modification as well. Therefore these four modules should be refactored to lower the coupling level. At this point there are two choices: refactoring these modules before any changes are requested to them, or postpone the refactoring to sometime in future. The latter choice would incur technical debt. The cost of the debt (“interest”) can be estimated by considering the probable future cost of a change to one of these modules, under the two different decision options (refactor now and refactor later). If the decision is to refactor now, then the future cost of a change to one of the four modules is likely to be just the cost to change that one module. If the decision is to refactor later, then the likely future cost of a change to one module is the cost to change that module, plus the cost to change the other three modules, because they are still highly coupled to the first module. So the decision about whether to refactor now or later must take into account not only the cost of refactoring (and whether that cost should be incurred now or later), but also

the extra cost of modifying extra modules when a modification to one module is required in the future.

Accordingly, planning must take into account not only the short term advantage of incurring technical debt, but also the longer-term costs. A strategy may become less promising, even void, if the long term cost is too high. Given the long and short term effects of technical debt, software managers have to balance the benefit of incurring technical debt with the associated cost.

In summary, the concept of technical debt can be defined as the short term benefit (usually in the form of reduced time) and the long term cost (usually in the form of higher maintenance cost) of delaying or shortcutting development and maintenance tasks, which prevents software from achieving its ideal quality and thus results in immature software artifacts.

1.2 Technical Debt Characteristics

The word “debt” is defined as “something owed” in Webster’s dictionary. It can refer to assets or moral obligations that a person owes others, but its most common form is financial debt, which has become an indispensable part of modern life. By contrast, “Technical Debt” is only a metaphor, which means that it is borrowed from or inspired by concepts in other domains. Whether it is financial or technical, the use of the word “debt” in the two terms suggests commonalities between them. Therefore a close comparison of the two concepts will shed light on the essence of technical debt.

For financial debt, a debtor is concerned about what the obligation is and how it can be fulfilled. To be specific, the debtor should be clear about how much he borrowed, the cost for the money borrowed and when it needs to be paid back. He could select different payment methods, which bring different degrees of flexibility as well as overall cost. The amount of money that the debtor borrowed is the principal, while the cost of using the borrowed money is the interest on the debt. Likewise, technical debt also has the same two elements. In the software world, the principal is the amount of effort required to complete a task that was previously delayed, while the interest is the potential penalty, in terms of increased effort and decreased productivity, that will have to be paid in the future as a result of not completing the task in the present.

In spite of these similarities, big differences arise between the two in terms of the precision of measurement and uncertainty. For those who borrow money from financial firms or other persons, such as an auto loan or a mortgage, the amount of money they want to borrow is predetermined. In addition, the debtors will be given either an interest rate (fixed) or a rule that prescribes how the interest rate fluctuates or what the fluctuation will be based on (adjustable). Once the debtors make their payment plan, it is easy to precisely calculate the monthly payment as well as the interest, i.e. the total cost, that the debtors will pay. Different payment plans can be simulated easily to help the debtors find the most suitable one that can maximize the debtors' utility. By contrast, it is not easy to calculate the principal and interest on technical debt. Even worse, even identifying technical debt is still a challenge both technically and economically.

Since it is what the developers owe the system, technical debt can be estimated by measuring the effort required to pay off the debt, e.g., the effort to execute the deferred test cases for a module. Since the 1960's, various software effort estimation techniques based on either formal estimation or expert estimation have been proposed, such as COCOMO, function point analysis and Wideband Delphi [8-10]. The goal of the research in this area is to improve the accuracy of software effort estimation. However, the accuracy of the existing approaches heavily depends on the context where the approaches are applied [11]. In addition, some approaches are too complicated to be applied though theoretically promising in their estimation accuracy. Hence the dominant estimation approach is expert estimation [12]. The dominance of expert estimation indicates software effort estimation is still immature and subject to estimation errors. Therefore, estimating technical debt by evaluating the required effort may not produce precise results.

Incurring technical debt causes immature software artifacts, which fail to meet certain software quality criteria. Therefore, technical debt can be identified by comparing the quality of software artifacts with quality standards. However, software quality is a complex concept. On one hand, software quality has many attributes and different groups of people may have different views of software quality [13]. On the other hand, some attributes of quality, e.g. maintainability and usability, are hard to assess directly. Therefore, software quality standards usually take the form of guidelines or heuristics. Thus using such guidelines or heuristics to detect technical debt has to rely on human judgment and inevitably is subjective. Another approach to software quality assessment is to use software quality metrics. A software metric is any type of

measurement of a software attribute. No matter what metrics are used, the relationships between the metrics and software quality attributes must be determined so that it is clear that software quality can be properly measured using these metrics. Unfortunately, it remains unknown what the relationship is between software quality indicators and future maintenance cost and quality (i.e. interest). There is evidence that they are correlated, but it's not enough to derive from the evidence that manipulating the values of these indicators will actually have an effect on reducing interest. Given the difficulties presented above in using software quality indicators as indicators of technical debt, identifying technical debt in this way would be a complicated process.

For financial debt, it is known whether a debtor needs to pay the interest and if so, the total amount of the interest can also be determined before he goes into debt. For technical debt, this is not always the case. For example, if a certain portion of the system in fact has no defects, then no harm is done in saving some time by not testing it. Similarly, if a particular module is never going to be modified in the future, then failing to update its related documentation will save some time during modification of the module, without any adverse consequences. Therefore, technical debt involves uncertainty that it may or may not have associated penalty. The likelihood that the penalty is realized depends on whether changes will be requested on that artifact and the kind of changes that will be made. This is unique to technical debt.

Ideally, software managers would incur technical debt only on those artifacts where there will be no interest, or give low priority to these artifacts while focusing on those

that are subject to penalty. However, the difficulty is that it is rarely known beforehand if a particular portion of the system has defects or not, or if a particular module is ever going to need modification, i.e. if it is going to incur interest. In this sense, technical debt can be considered as a particular type of risk in software maintenance as it has the basic elements of a risk – the potential loss/penalty and the associated uncertainty. Therefore, the problem of measuring and managing technical debt boils down to managing risk and making informed decisions about which delayed tasks need to be accomplished, and when.

1.3 Current Technical Debt Management Research and Practice

The characteristics of technical debt imply a strong relationship with software quality assessment, software development effort estimation and software risk management, none of which is a new research area. In the discipline of software quality, a number of metrics and models have been proposed to assess, monitor and predict software quality. Research results from the area of software development effort estimation facilitate software project planning, investment analyses and product pricing.

Software risk management has even been integrated into software development processes to improve software quality and productivity. Although research achievements are fruitful in each area, there is limited work that has leveraged the power of approaches from all three areas to address the technical debt problem.

Therefore, we discuss key achievements in the three research areas in Chapter 2 to identify approaches that can be applied to technical debt management. Actually this dissertation research has drawn inspiration from the results and findings in these areas, including using risk exposure analysis to prioritize technical debt instances and

cost estimation techniques for technical debt evaluation. We expect that leveraging the research results from these areas could further advance the research on technical debt management.

Attention to the technical debt problem has not been only from the research community. In fact this metaphor originated in the software industry, where technical debt spreads widely, affects most software products and systems, and has reached the level that cannot be ignored anymore. It has been reported that technical debt of an average-sized application of 300,000 lines of code is \$1,083,000, which means each line of code carries \$3.61 of debt [14]. These figures are based on the estimated cost of paying off the debt, but does not take into account its impact if not paid off (i.e. interest). Moreover, technical debt may be incurred under various conditions with different reasons, which raises the difficulty in addressing this problem [15]. In a recent interview study, practitioners reported that technical debt is unavoidable, and thus effort should be made to manage it, not avoid it [16].

Now that technical debt is a common and recognized phenomenon in the software industry and it could have serious impact on software projects if left unattended, whether or not to manage technical debt is no longer a choice. The question that remains is how to manage technical debt. In the context of improving software quality and productivity, technical debt management refers to the practice of identifying the technical debt currently existing in a software system, understanding its present and future impact, and leveraging the technical debt information to make decisions about the software project. In other words, technical debt management includes technical

debt identification, measurement and decision making. It also includes continuous monitoring of technical debt because technical debt changes as software evolves and actions are taken on it. But decision making is the central problem of technical debt management. All of the other activities and the information they gather serves this purpose. In particular, the ultimate goal of this dissertation research is to help software managers make informed decisions in their project management practice, by integrating technical debt information into their software processes.

Currently, managers and leaders of software maintenance efforts carry out technical debt analysis and management implicitly, if at all. Decisions are largely based on a manager's experience, or even gut feeling, rather than hard data gathered through proper measurement. However, on large systems, it is too easy to lose track of delayed tasks or to misunderstand their impact. The result is often unexpected delays in completing required modifications, compromised quality and extreme cost overruns. By framing the problem of delayed maintenance tasks as a type of "debt", the technical debt metaphor reveals the essence of the problem – it brings a short-term benefit, such as higher productivity or shorter release time, but which might have to be paid back with "interest," later. Many practitioners find this metaphor intuitively appealing and helpful in thinking about the issues. Discussions about this topic are also pervasive on the web in such forms as personal blogs and online forums. However, what is inadequate is an underlying theoretical basis upon which management mechanisms can be built to support decision-making.

1.4 Research Questions

The objective of this research is to contribute to the establishment of a theoretical basis for technical debt management, using empirical research methods, that characterizes the cost and benefit sides implicit in software management practice. The emerging theory resulting from this empirical work describes the characteristics of the costs and benefits of explicitly managing technical debt, as compared to the implicit consideration of the issue in decision making, which is current industry practice. The results of this work characterize the costs and benefits through analysis of qualitative and quantitative data collected from practice. This cost-benefit characterization leads to our research questions.

RQ1: What are the characteristics of the costs and benefits of measuring and monitoring technical debt?

RQ2: In what ways does technical debt information contribute to decision making?

Moreover, we also develop and evaluate, based on emerging theory, a technical debt management mechanism for technical debt over the lifetime of a software system. The mechanism (consisting of measures and procedures) will facilitate software managers to keep track of the technical debt in their projects, understand better the impact of the technical debt and make informed decisions.

To address these questions, we carried out two types of studies – retrospective studies and case studies, on real software projects. Through decision simulation, the retrospective studies uncovered the benefits of explicit technical debt management.

The case studies revealed the costs of technical debt management by applying the proposed technical debt management approach in real time.

The results of this research contribute to the advance of software maintenance research and practice by providing insights, backed by empirical evidence, into the phenomenon of technical debt, and mechanisms for operationalizing the concept (i.e. providing decision support), of technical debt.

The remainder of this dissertation is organized as follows. Chapter 2 presents the related work. This begins with a review of the recent studies on technical debt, key findings and their influences on this research. Then it presents important work from three related areas – software risk management, software development effort estimation and software quality metrics, from which a risk management approach and an effort estimation model are picked for this study. In Chapter 3, the research methodology, including the proposed approach to technical debt management, and the research methods, is described and followed by elaboration of the retrospective studies and the case studies. From Chapter 4 to Chapter 7, the particulars of the implementation of each study, along with the findings, are presented. Chapter 8 concludes this dissertation with discussion of the results and findings from the four studies, the limitations and future work.

Chapter 2: Related Work

Besides the scholarly research on technical debt, which has started growing in recent years, the problem we are investigating is also related to existing concepts in other established disciplines. This chapter first reviews discussions in the public domain and foundational research work about the technical debt issue in section 2.1. Then concepts and approaches from three supporting areas - risk management, software quality assessment and software effort estimation - are reviewed in section 2.2, 2.3, and 2.4.

2.1 Technical Debt Literature

The term “technical debt” was introduced by Cunningham in 1992, where he presented the metaphor of “going into debt” every time a new release of a system is shipped. The point was that a little debt can speed up software development in the short run, but every extra minute spent on not-quite-right code counts as interest on that debt [17]. Technical debt originally referred to delayed or “quick and dirty” work in the design and implementation phases, resulting in immature code, but this metaphor has been extended to include any immature artifacts in the software development lifecycle.

Originally the technical debt metaphor has been mostly used as a communication device as shifting the dialog from a technical vocabulary to a financial vocabulary makes discussions clearer and easier to understand for non-technical people [18]. Discussions about this topic are pervasive on the web in such forms as personal blogs and online forums. The foci of the discussions vary from technical debt classification

and identification of technical debt, to solutions for controlling it. These discussions serve as a platform to foster reflections on the essence of technical debt and facilitate exchange of new ideas, thus providing opportunities for researchers to extract important topics and then investigate them in a scientific manner.

In the software research community some studies have been conducted to understand the causes (e.g. different code anomalies) and impacts (e.g. various aspects of maintainability) of technical debt. Meanwhile, approaches have been proposed to facilitate technical debt evaluation (e.g. quantification) and management. There is a growing body of technical debt research focusing on technical debt identification. Actually technical debt identification has become one of the most fruitful areas of achievement in technical debt research [19]. Many approaches have been proposed or adapted for technical debt identification. Specific tools and techniques have also been developed based on these approaches. For example, Izurieta and Bieman's work is targeted to identifying "grime", a metaphor for non-pattern related code that has a negative impact on maintainability, accumulated in design pattern realizations [20]. Bohnet and Doellner proposed to use "Software Maps", which visualize internal software quality and risk, for technical debt identification [21]. Zazworka et al.'s work on technical debt identification is based on code smells, which are patterns indicating poor programming practices and bad design choices and thus can be used as indicators of technical debt [22]. Similar to code smell detection but targeted to the architectural level, extended augmented constraint networks (EACN) have been used to transform software architectural models into dependency relations, which facilitates detection of software architectural decay [23]. Technical debt identification

is a significant area of technical debt research although it is not directly related to this dissertation work.

Technical debt classification and studies of its causes and impacts are foundational for this dissertation research. Investigation of the causes of technical debt further have contributed to the literature on technical debt classification. Investigating different ways that technical debt can be classified then facilitates understanding of the nature of technical debt. Based on classification, strategies and approaches could be developed to manage different types of technical debt, which is the objective of this dissertation work. In addition, foundational work in technical debt classification, causes, and impacts, also helps in understanding how technical debt manifests, and thus facilitates development of technical debt identification techniques. Therefore, in section 2.1.1, we first introduce technical debt definitions and classifications. Then in section 2.1.2, we present the results from previous studies on the causes and impact of technical debt. Finally in section 2.1.3 we discuss some technical debt management approaches proposed by other researchers in this field.

2.1.1 Definition and Classification

Early work in this area focuses on establishing the foundation of technical debt research by conceptualizing the phenomenon and classifying technical debt. There are multiple dimensions by which technical debt can be classified. In McConnell's article, technical debt was categorized as unintentional and intentional debt [18]. Unintentional debt occurs due to a lack of attention, e.g. lack of adherence to development standards or unnoticed low quality code that might be written by a

novice programmer. Intentional debt is incurred proactively for tactical or strategic reasons such as to meet a delivery deadline. The intentional debt was further broken down to short term debt and long term debt. The short term debt refers to small shortcuts, like credit card debt, while the long term debt may result from strategic actions, like a mortgage. Based on this classification, Fowler created the technical debt quadrants, as shown in Figure 1 [24]. The quadrants are formed by two dimensions – deliberate/inadvertent and reckless/prudent. Each quadrant represents a particular type of debt that takes the combination of the values from each dimension. This type of classification is helpful in finding the causes of technical debt, which has been demonstrated by Yli-Huumo et al.’s study [25]. In this study multiple causes of the intentional and unintentional technical debt have been identified through interview of the personnel working on two product lines in a software company. Finding of the causes of technical debt could further lead to different identification approaches. For example, for reckless and inadvertent debt, especially design debt, source code analysis may be required to identify it.

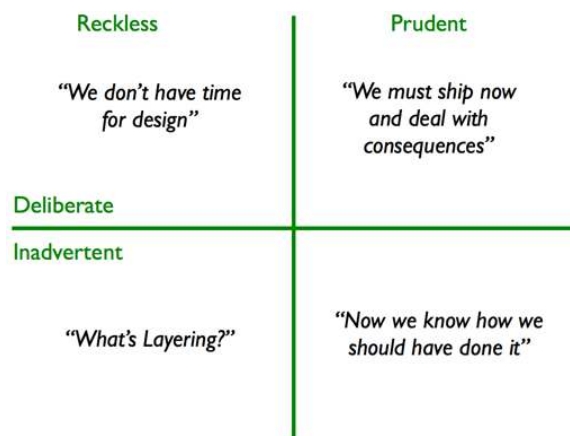


Figure 1. Technical Debt Quadrants [24]

Technical debt can also be classified in terms of the phase in which it occurs in the software lifecycle – design debt, testing debt, defect debt, etc. [26]. Design debt refers to the design that is insufficiently robust in some areas or the pieces of code that need refactoring; testing debt refers to the tests that were planned but not exercised on the source code. This type of classification sheds light on the possible sources and forms of technical debt, each of which may need different measures for identification, and approaches for management. For example, comparison to coding standards may be required to identify and measure design debt, while testing debt measures require information about expected testing adequacy criteria.

Moreover, this type of classifications makes it easy to associate technical debt with a set of phenomena. Some of these phenomena manifest themselves in obvious ways.

For example

- documentation is partially missing,
- the project to-do list keeps growing,
- minor changes require disproportionately large effort,
- code cannot be updated because no one understands it anymore,
- it's too difficult to keep different portions of the system architecturally inconsistent.

These phenomena can be seen as symptoms of technical debt [27].

Revolving around the technical debt metaphor, much debate and discussion has ensued on what technical debt is, the limits of the metaphor and how it should be scoped [28, 29]. People from different perspectives, for example, industry software

practitioners vs. academic researchers, have different points of view of this concept, hence their understandings and emphases vary [30]. The lack of a proper definition and scope has created difficulties in investigating technical debt and in communicating among different research groups. In an effort to alleviate this problem, Tom et al. conducted a comprehensive review of the technical debt literature [31]. In Tom et al.'s work the technical debt concept was defined with a set of dimensions, properties and outcomes by consolidating the work from different research groups. The dimensions of technical debt proposed in their study largely conform to the classification based on software lifecycle, including code debt, design and architectural debt, knowledge distribution and documentation debt, and testing debt. The only difference is the addition of "environmental debt", which refers to the debt incurred due to environmental changes such as technology advance. Critical components for the technical debt concept were defined as "attributes", which include monetary cost, amnesty, bankruptcy, interest and principal, leverage, and repayment and withdraw. According to the findings of their study, technical debt may impact software quality, development productivity, project risk and team morale. To the best of our knowledge, Tom et al.'s work is the first formal research that tries to standardize the vocabulary of technical debt and its classifications. Similar to Tom et al.'s work, Li et al. recently conducted a systematic literature review on technical debt and its management [19]. In their work they compared the technical debt concepts and classifications defined by different research groups, described their similarities and conflicts, but they didn't attempt to reconcile the conflicts to develop their own technical debt classification. Ampatzoglou et al.'s work [32] is another systematic

review on the technical debt literature, but it focuses on the financial aspect of technical debt management. A technical debt financial glossary was developed through synthesizing different ways of using the financial terms on technical debt, merging similar terms and reconciling conflicts between the definitions of the terms from different studies. The glossary contains terms such as Return on Investment, option and hedging, providing a common vocabulary for technical people and project managers and hence facilitating communications between them.

2.1.2 Causes and Impact

As discussed in Section 2.1.1, technical debt may be incurred by different causes. Several studies aiming at this aspect have been performed by researchers. For example, to identify these causes as well as the consequences of technical debt, Lim et al. interviewed 35 software practitioners including managers, developers, and quality assurance engineers [16]. Through the interview study they found that time/budget constraints and bad decisions are the two primary causes although there are others. Consequences of incurring technical debt vary with different types of technical debt. This study helps understand the nature of technical debt by presenting insights from the software practitioner's point of view.

Investigation of the causes of technical debt may focus on a particular phase of the software lifecycle, in which case the causes of technical debt are more specific. For example, Wiklund et al. studies the technical debt regarding test automation in a telecommunication system [33]. Through interview of the project personnel they found out that non-transparency of their testing facility infrastructure and the lower

quality of testing code than production code are the main contributors of technical debt.

Another interview study on technical debt was conducted by Klinger et al. in 2011 [15]. Instead of focusing on the decisions made by individuals on the project level, they approached the technical debt issue from an enterprise perspective. In this study they explored the premise that incurring technical debt on the enterprise level is more like using financial leverage than making a bad decision, which is often the case when technical debt is incurred on the project level. The findings from their study describe the complexity of incurring technical debt in the enterprise context and how the causes of enterprise-level technical debt are different from those on the project level.

Through a multiple case study Izurieta and Bieman investigated the causes and impact of design pattern decay, deemed as a type of technical debt at the design level [34]. The results of the study show that grime, i.e. the non-pattern related code, is the primary cause of the design pattern decay, which eventually increased dependencies between design pattern components, reduced pattern modularity, and decreased testability and adaptability.

There are other studies that either investigated the technical debt issue in a particular type of software organization, such as Giardino et al.'s study [35] on software startup companies, or gathered the perceptions and experiences of software practitioners on the causes and impact of technical debt, such as Holvitie et al.'s work [36].

Reviewing these studies gave us multiple perspectives to look into the technical debt phenomenon. The findings of these studies offer insights with different levels of depth into the causes of technical debt, which together depicts a typical way of technical debt formation: it starts with time pressure, a common problem that most software projects have to face. When the planned work can't be completed by the deadline, cutting corners becomes an easy choice for software managers to deal with the problem. As a result, tasks with low priority are postponed and compromises are made on those "uncritical" parts. While individually they may not be noticeable, these compromises accumulate over time to finally have significant impact on the software project such as paying high maintenance cost sometime in the future, as the technical debt metaphor suggests.

Besides the causes, the impact of technical debt on software projects and businesses is another essential part of the technical debt problem. Yli-Huomo et al. [25] investigated the technical debt problem in their case study of two product lines in a software company. Through interviews, they collected various instances in which technical debt seriously impacted their projects. One of the instances is that they decided to put patches on a calendar synchronization function to allow modification of the functions depending on it, rather than refactor the function. It turned out that this decision cost them hundreds of hours in fixing bugs after the first release of the function.

The impact of technical debt was also reported by Lim et al. [16]. In their study they interviewed software practitioners with various roles such as developer, designer or

project manager about their experience on technical debt in the projects involving them. According to the findings of the study, one impact of technical debt was the unpredictability of the system, which made more difficult the estimation of the time and effort required to make changes and eventually resulted in higher maintenance cost.

To provide a concrete idea of the magnitude of technical debt in the real world, CAST evaluated technical debt in hundreds of applications from their repository and converted the debt to financial cost per lines of code [14]. Based on this report, there are also discussions on what types of technical debt may trigger higher costs in what part of a system [37].

2.1.3 Management Approaches

In order to manage technical debt, a way to quantify the concept is needed. One approach is to monitor the changes in software productivity during the software development process. In many cases, development organizations let their debt get out of control and spend most of their future development effort paying crippling “interest payments” in the form of harder-to-maintain and lower-quality software. Since the interest payments hurt a team's productivity, the decrease in productivity can reflect how much debt an organization has [38].

Since technical debt is closely tied to software quality, approaches to software quality assessment can be used to evaluate technical debt. The SQALE Method is an approach of this type [39]. The method is based on a software quality model called SQALE (Software Quality Assessment based on Lifecycle Expectations), which is in

turn based on the ISO 9126 standard. The model is organized as a three-level hierarchy. The top level is composed of quality characteristics such as reliability and maintainability. The second level contains sub-characteristics for each characteristic. For example, the sub-characteristics for “changeability” are data related, architecture related and logic related changeability. The third level is composed of requirements that relate to the source code’s internal attributes, e.g. “the number of derived classes is less than 40”. The third level requirements actually define the “right code” for the quality target. Then non-compliance is the distance between the current state of the code and the right code, which is measured by the analysis model using remediation indices. A remediation index represents the remediation effort of correcting the non-compliances against the model requirements. The total effort, calculated by aggregating the indices to the top level, represents the amount of technical debt currently in the system. The effectiveness of this approach depends on the accuracy of the remediation indices, which may be derived from historical project information.

Another metric based on the same rationale was proposed by Smit et al. [40]. The metric was named “convention adherence”, which measures the severity of a violation of the predefined coding conventions. Convention adherence was considered as an indicator of the technical debt accumulated in a system with the assumption that less adherence requires more maintenance effort, but the relationship between the two was not validated in their study.

Other metrics for software quality can also be used to measure technical debt. For example, if the software is inflexible or overly complex, then future changes will be

more expensive. From this point of view, coupling, cohesion, complexity, and depth of decomposition are metrics that can be applied to the problem of characterizing technical debt [41]. Ultimately a good design is judged by how well it deals with changes [41], so time and effort required for changes (in particular the trend over time) is also an indicator of technical debt.

Nugroho et al.'s work defines technical debt as the cost of repairing quality issues to achieve an ideal quality level, and the interest on technical debt as the extra maintenance cost spent for not achieving the ideal quality level [42]. The ideal quality level in their approach is represented by a set of software metric values such as the lines of code (LOC), McCabe's cyclomatic complexity and fan-in/fan-out. These metrics are mapped onto ratings—five level (star) ordinal scales—for properties at the level of the entire software product, such as volume, duplication, and unit complexity. Five stars represents the ideal quality. The mapping uses risk profiles to characterize software systems with different quality levels. A risk profile is a partition of the size of a system into four risk categories: low, moderate, high, and very high risk. For example, a software system with at least 75% or more lines of code in low risk category has 5-star quality level. From the historical project information the profile of each quality level can be derived. Then Rework Fraction (RF) can be calculated to create a RF table. A Rework Fraction is defined as the percentage of lines of code that need to be changed to improve the quality of software to a higher level. With the RF table, the change effort of improving the system from its current quality level to the ideal quality level can be estimated, thus the technical debt currently in the system can be easily estimated. Likewise, the interest can also be estimated. The advantages

of this approach lie in the use of the objective quality measures and fewer inputs and assumptions. However, it is not applicable to situations where the project lacks historical information.

Technical debt can also be measured by characterizing the cost of paying the debt. For example, after determining what types of work are owed the system (e.g. architecture redesign, code refactoring, and documentation), the labor, time and opportunity cost can be estimated for fixing the problems identified. This cost reflects the amount of principal of technical debt currently in the system.

Besides the evaluation approaches, strategies for paying off technical debt have also been discussed. One strategy is to determine the time point at which technical debt has to be paid [43]. Figure 2 illustrates how good design could affect the cumulative functionality of a software project. A good design does require upfront time and effort, but it makes software flexible and easy to accommodate changes so that the project can go faster for longer. Neglecting design could save short-term time, but the consequence is the accumulated technical debt, which will slow down the productivity later. Therefore, the two strategies, i.e., good design and no design, have different trends in terms of time and cumulative functionality of the project. Since the “no design” strategy can no longer prevail beyond the point at which the two lines join, the point can be considered as the breakeven between principal and future interest. In other words, this approach can be used to determine whether it is worthwhile to incur debt, when it should be paid in order to minimize the total time or maximize the overall productivity.

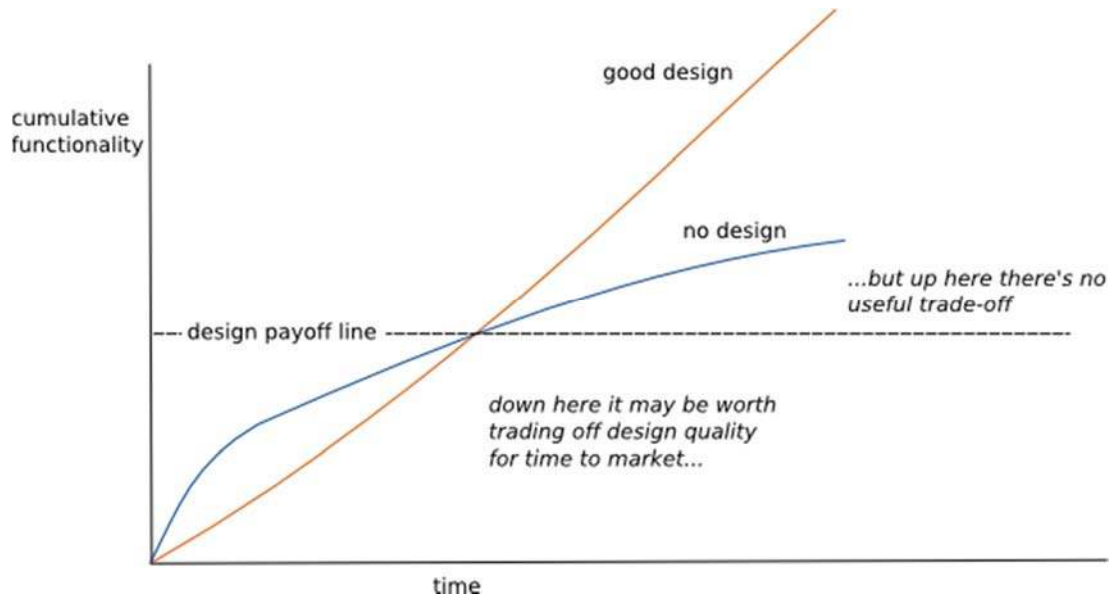


Figure 2. Cumulative Functionality of the Two Design Strategies [43]

The above discussion only illustrates the rationale behind different design strategies. By contrast, Ramasubbu and Kemerer performed a rigorous analysis on these strategies [44]. In their study 69 customers of a commercial enterprise software package were tracked regarding functionality growth of the software package over its ten year lifecycle. The customers were categorized into three groups – early, base, and late adopter of the software package, according to the pace at which they added customized functionalities through source code modification. Early adopter refers to the customers who tend to add business functionality at a faster pace than the vendor firm could fulfill using its standard releases; base adopters just use what the vendor has released without any modification; while late adopters tend to delay upgrading to the latest release of the software or only uses a subset of the available functionalities. In the language of technical debt, early adopter and late adopter correspond to higher-debt scenario and lower-debt scenario respectively, shown as dash line in Figure 3. The solid line in the figure represents the functionality accumulation of base adopters

over time. The study hypothesized that the functionality accumulation of the software adopted by the three types of adopters follow different pathways with different costs and benefits over time due to the effect of technical debt. For example, early adopters can add features to the software at higher speed and hence achieve higher customer satisfaction than the base and late adopters, in the early stage of software lifecycle, by sacrificing software quality, especially maintainability, but the speed is decreasing over time as more and more effort has to be devoted to maintenance rather than adding new features.

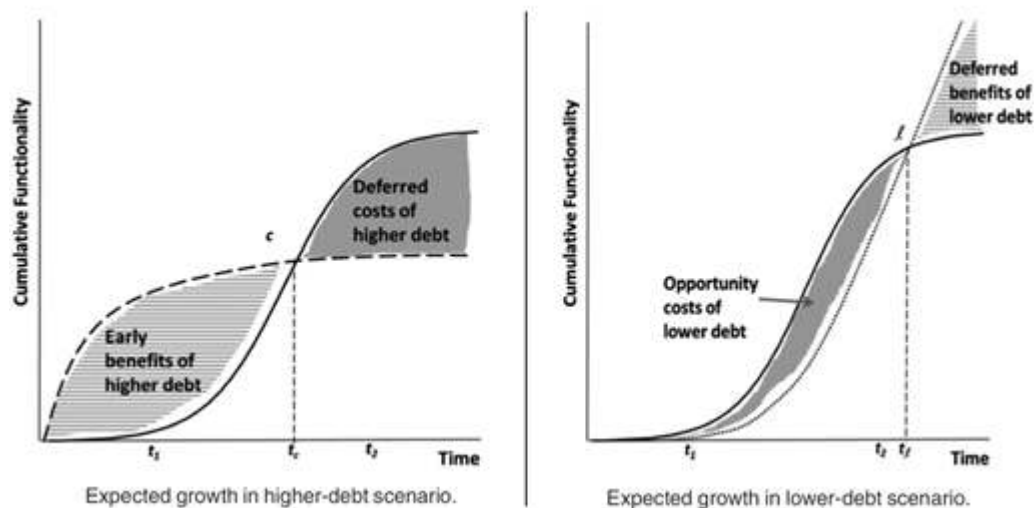


Figure 3. Cumulative Functionality in Two Scenarios [44]

Through statistical analysis, these hypotheses were tested and the model, i.e. the patterns of software evolution in different debt scenarios, were validated. Based on the results of the study, they proposed an approach to control software evolution and maximize the benefit for software managers. The approach set 3 decision points for each of the debt-carrying strategies, as show in Figure 4. At these decision points software managers may make a decision on whether they keep the current trajectory or change to another one. While the approach is theoretically promising, the problem

it seeks to address is still on a high level and hence the approach mainly serves as a guideline for software development management. Moreover, if the debt-carrying strategy is changed in the process of the evolution, the trajectory would be much more complicated than an “S” shape. In that case only timing the check points and watching the changing signs may not lead to the optimal decision that maximizes the overall benefit of the strategy.

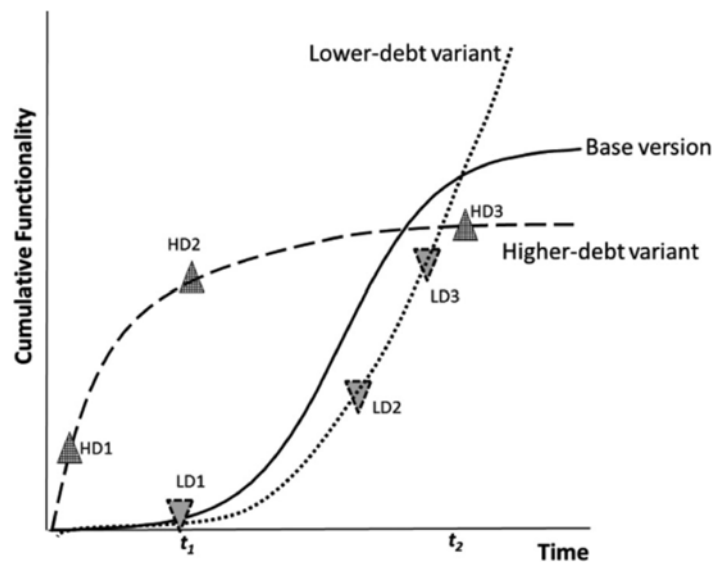


Figure 4. Development Investment Decision Approach [44]

On a conceptual level, technical debt is studied as a whole, but in practice, technical debt in a system is manifested as individual pieces, which we call technical debt items. Considering technical debt in this way, managing technical debt comes down to determining when and which debt item should be paid off. One strategy is to pay the highest interest debt items first [27], however, the principal of the debt item should also be considered as it is the current cost for paying it off.

When numeric estimation for principal and interest is impossible or hard to achieve, an alternative approach is to use ranking to prioritize the technical debt items. In Zazworka et al's work technical debt items (classes) can be identified using a set of criteria composed of software metrics and thresholds [22]. The thresholds and metrics used to evaluate these classes came from the definition of the code smell "god class", which refers to a class that implements too much responsibility on its own. For the cost of paying off the debt, i.e. the principal, the classes are ranked in terms of their distance to the thresholds with the assumption that a class that is close to the thresholds will be easier to refactor than one that is multiple magnitudes outside. For the interest on the debt, the classes are ranked according to defect and change proneness. The priority of these classes for refactoring is finally determined by two dimensions: cost and interest, as shown in Figure 5. One strategy for paying off the debt associated with god classes is to choose classes starting with the upper left corner of the chart, and then moving down and to the right. The classes in the upper left corner (e.g. GodClass8 and GodClass7) are most likely to have a higher effect on future maintenance (high interest), but are relatively cheap to pay off (low principal).

Fontana et al. conducted a similar study using code smell as a technical debt indicator. The objective of this study is to advise on which type of code smell indicates more serious problems than others and thus should be addressed with priority [45]. In the study they first identified code smells from the subject software applications. Then a set of complexity metrics are applied before and after the code refactoring to evaluate the impact of code smells. The proposition was that if removal of a type of code smells makes more improvement on the metrics than other types of

code smells, refactoring the code infected by this type of code smells has more value and should be given higher priority.

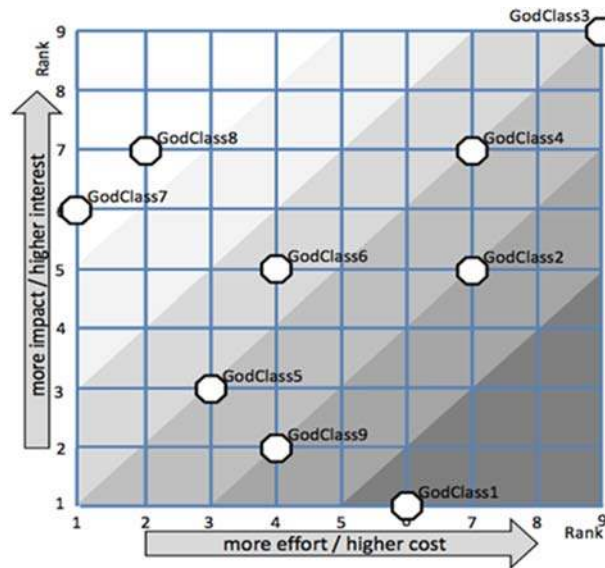


Figure 5. Cost/Benefit matrix for Nine God Classes [22]

The strategies and approaches mentioned above for technical debt decision making are less formal, based on personal experiences from tackling similar problems in related research areas, and they reflect early thoughts on the technical debt management problem. By contrast, some other approaches for technical debt decision making take a more formal approach. For example, Schmid proposed an approach to formalize the technical debt concept and decision making [46]. In this approach, technical debt is modeled using implementation cost, rework cost and future evolution path. Thus technical debt decision making turns into an optimization problem. Although this is theoretically promising, the approach is not practical for real world application due to the hard requirements for finding an optimal solution, even with relaxed assumptions. Therefore, Schmid proposed a simplified approach that only considers evolution cost, refactoring cost and the probability that the

predicted evolution path will be realized. In the sense of technical debt decision making, this simplified approach is very similar to the principal-interest-probability approach we use and hence supports its application in this study. Another example is Nord et al.'s work in which technical debt for two distinct delivery strategies, i.e. quick delivery vs. low rework cost, were evaluated using an architectural metric [47]. With a concrete software project, they demonstrated the importance of architectural debt information in decision making related to product delivery.

Agile development appears to be more prone to technical debt accumulation compared to traditional software development approaches, due to its delivery-oriented focus. To take advantage of agility while still maintaining a stable infrastructure for long term health, Bachmann et al. proposed a set of architectural tactics for system decomposition, architecture development and staffing, and proposed a way to make these tactics work together harmoniously [48]. Although the tactics do not explicitly address the technical debt management issue, they offer another angle to view and investigate the technical debt problem.

In this section we described some technical debt management approaches proposed by other researchers in the field. Most of the approaches focused on a particular aspect of the technical debt management problem such as the quality metrics for evaluating technical debt. Some of them were actually basic ideas for reflection on the problem, e.g. finding the breakeven point at which technical debt should be paid off. But managing technical debt requires a holistic approach that can address the main aspects of the problem – whether and how technical debt should be paid off.

Even in the aspect they are targeted to, the approaches described above have limited power to be applied in practice due to the stringent assumptions or application conditions. For example, the SQALE method and Nugroho et al.'s approach heavily rely on historical project information, which are often lacking or unavailable. Therefore, the technical debt management approach we propose in Chapter 3 embraces all main components for technical debt management, including both principal and interest. Moreover, the framework was designed to allow human judgment in all stages to overcome the limitation of the approaches that rely on historical project information. Moreover, none of these approaches in the literature have been tried in practice, which leads to the objective of this research – developing and empirically exploring a technical debt management approach in terms of its costs and benefits.

In a recent case study, Martini et al. [49] explored technical debt management in a real industrial environment by identifying causes of architecture debt, modeling debt accumulation, and evaluating different refactoring strategies in terms of the frequency of “crises” as defined by the authors. Thus, Martini et al.'s study comes closest to our study. However, the study didn't focus on the economic aspect of technical debt management. Therefore, the management strategies they discussed in the study do not address such questions as when and what technical debt should be paid off, which our studies were targeted to answer.

2.2 Software Risk Management

As described in Chapter 1, managing technical debt is more complicated than managing economic debt. One of the reasons is that technical debt involves uncertainty. The uncertainty of technical debt warrants its proximity to the concept of software risk. Therefore, managing technical debt could leverage the approaches to software risk management. In this section we present what has been accomplished in the research area of software risk management. In particular, we review the risk management processes, approaches and techniques from which we draw inspirations for this research.

2.2.1 Overview

Risk can be defined as the potential that undesirable events with negative consequences will occur [50]. This definition indicates that risk has two basic elements - the uncertainty with respect to the occurrence of the events and the loss if the events occur. Risk management, as a multidisciplinary problem, has been investigated in different areas. The goal of project risk management, in general, is to reduce the threats to project success no matter what areas it is applied to. Given their dynamic, abstract and complex nature, software projects face many challenges and risks. Risk management for software projects involves identifying and addressing the incidents that endanger a successful software operation or lead to implementation difficulty, delay or rework [51].

Since the 1970's, various techniques and approaches have been proposed to deal with software risks [51-56]. Among them the most famous is Boehm's approach [51],

which established risk management as an important research field in software project management and laid the foundation for most of the work in this field [57]. Another line of research on software risk management was conducted by the Software Engineering Institute (SEI) at Carnegie Mellon University. The outcome of this research was a risk management framework called Continuous Risk Management (CRM) Paradigm. SEI's CRM was considered one of the most comprehensive collections of practical techniques that can be used in various steps during risk analysis [57]. Given the important role of the above two approaches in software risk management, we present their details in Sections 2.2.2 and 2.2.3 respectively.

In addition to these risk management approaches specifically designed for software risk, some software researchers have leveraged approaches from other fields. Costa et al. proposed an approach to calculate the probability distribution of losses and earnings that can be attained from a software project portfolio [58]. By mapping risk concepts in software engineering to the financial domain, they applied credit risk theory for loan operations to assess software risks. The theory was extensively used by financial institutions but was never applied in the software engineering domain. Similarly, Kumar proposed the use of concepts from the finance domain, such as hedging, to study software risks and project decisions [59]. Their work provides a new perspective to understand risks in the software engineering domain.

Managing risks in software projects is a process of identifying risk, assessing risk, and taking steps to reduce risk to an acceptable level [60]. These basic steps for managing software risks have been defined or described by different researchers in

their work, especially in the comprehensive risk management approaches [51, 53-55]. Since technical debt is closely related to software risks, a process for managing risk is key to developing a technical debt management process. A summary of the risk management processes defined in different approaches will be presented in section 2.2.4, followed by a discussion of the applicability of the risk management process to technical debt management.

Within the risk management paradigm, risk analysis is still the central problem and major challenge due to the difficulty in quantifying and predicting software risks. Therefore developing and improving risk analysis methods remains a hot topic in this field. There is a large body of research on risk analysis methods and techniques, including risk assessment frameworks, models, and techniques for estimating risk impact and probability of occurrence, which we will elaborate in section 2.2.5.

Besides the risk management approaches and risk analysis techniques, there are empirical studies in this field. Some studies attempted to evaluate risk management approaches, while other studies focused on risk patterns in software projects and relationships between risk control strategies and project success. Details of these will be given in section 2.2.6.

2.2.2 Boehm's Approach

Boehm's risk management approach introduced measures for software risks and consolidated risk techniques into a single framework. His approach consists of two major steps – risk assessment and risk control. Risk assessment is divided into three sub-steps: risk identification, risk analysis and risk prioritization. Risk control

includes risk management planning, risk resolution and risk monitoring. Each of these steps is supported by a set of techniques. For example, software risks can be identified using a risk checklist. Figure 6 shows the structure of the approach.

Figure 2.3

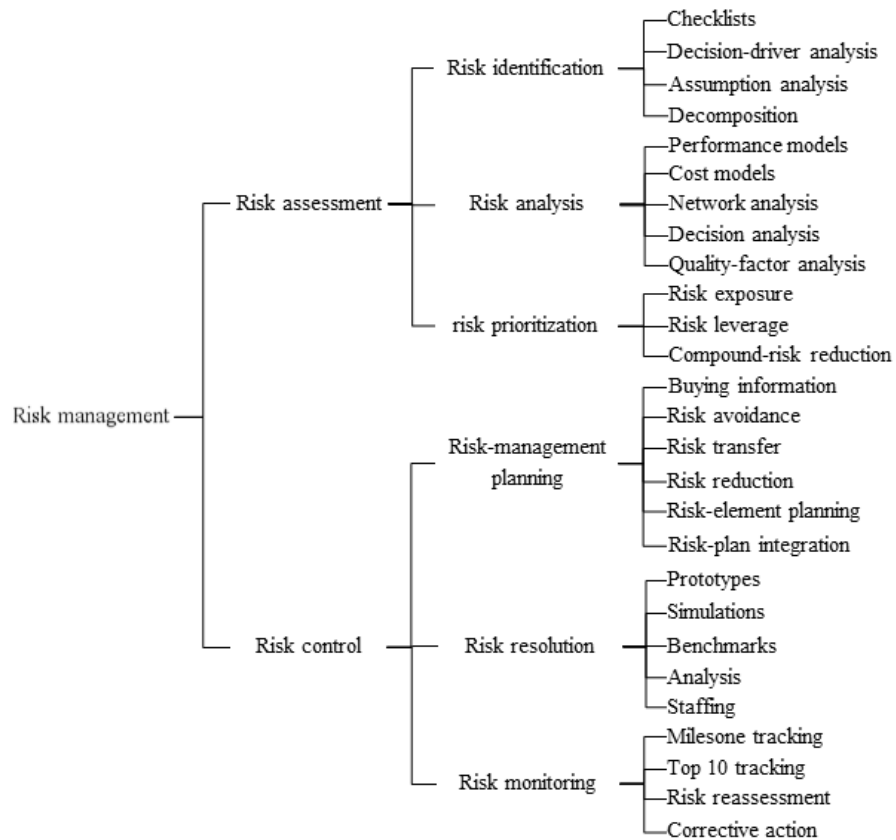


Figure 6. Boehm's Risk Management Approach [51]

2.2.3 SEI's Approach

This line of research started with the development of a risk taxonomy [61]. In the taxonomy there are three top categories - product engineering risk, development environment and program constraints. The product engineering category covers risks regarding software lifecycle phases such as requirements, design and testing. The development environment category includes risks of processes such as the development process and the management process. The program constraints category

consists of risks on resources required for projects and interfaces between the system and external entities such as customers. Along with the taxonomy is a questionnaire containing 194 questions, which cover all the entries in the taxonomy, to facilitate risk identification. SEI's risk taxonomy is a landmark effort in classifying risks into known groups or classes. This taxonomy has inspired many risk management practices.

Based on the taxonomy, a risk evaluation method was developed and enhanced to form a paradigm - Continuous Risk Management (CRM) [56, 62, 63]. CRM consists of five continuous tasks to guide the risk management process:

- identify software risks using SEI's risk taxonomy,
- analyze the risks to determine their probability of occurrence, impact on the organization and the relationships between individual risk statements,
- plan risk mitigation for the identified risk areas,
- track individual risks, mitigation plans, and the risk process to determine the effectiveness of mitigation actions and the risk process,
- control deviations from planned risk mitigation actions and make decisions, e.g., close, re-plan, continue tracking.

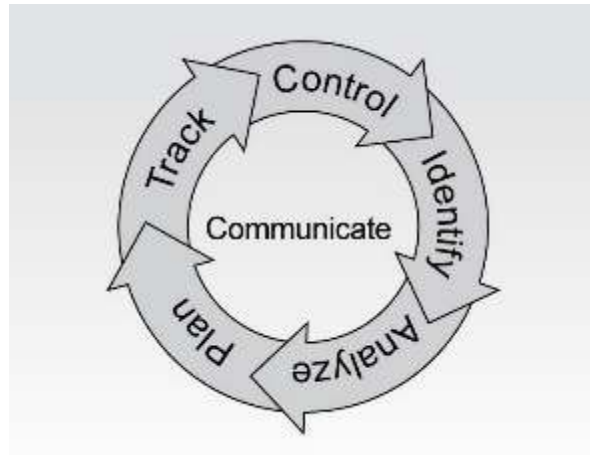


Figure 7. SEI Continuous Risk Management (CRM) Paradigm [63]

As shown in Figure 7, CRM emphasizes that effective risk management should be a continuous process. Communication is represented as an encompassing activity to stress that the flow of information throughout the project or organization is essential to successful risk management [63].

2.2.4 Risk Management Process

Since risk is a general phenomenon in all segments of our society, there are risk management processes in other domains. For example, in the ISO standard, the process of risk management consists of establishing the context, identifying risk, assessing risk, planning risk treatments, implementing and evaluating the plan [64]. The risk management process is also defined in the standards issued from institutes such as U.S. Department of Defense (DoD) and NASA [65, 66]. Similar to these risk managing processes outside the software domain, Boehm considers software risk management as a process that involves two primary steps, each with three subsidiary steps, as shown in Figure 6 [51]. SEI's approach defines the risk management process as a set of continuous tasks including identifying, analyzing, planning, tracking and

controlling the risk, as shown in Figure 7 [63]. Although these risk management processes pertain to different domains and their definitions are presented in different ways, they are consistent with one another in terms of the basic components that the process should possess. Here we synthesize the risk management approaches into the following basic steps:

- (1) Identify the risks regarding project, product and business,
- (2) Analyze the risks identified in the first step to determine their priority by evaluating their likelihood and consequences,
- (3) Propose, plan and apply strategies to control the risks,
- (4) Evaluate the effectiveness of the planned risk control strategies.

Managing risks follows an iterative process that continues throughout the project. On the one hand, the identified risks need to be updated if they are already handled. On the other hand, the risks have to be re-evaluated as more information regarding the risks becomes available or the environment changes. In general, a risk management plan is required to continuously monitor and track risks of the project.

Since technical debt can be considered as a particular type of risk in software projects, we mean to follow the same process (identify, analyze, control and evaluate) to manage technical debt. Furthermore, some approaches used for risk analysis, especially those used for probability estimation and risk prioritization, are also applicable to technical debt management.

However, technical debt management is not exactly the same as risk management. For example, approaches for technical debt identification are essentially different

from the approaches for software risk identification. Risk identification approaches aim to identify potential problems related to the software project, product or business. These approaches include intuitive methods such as brainstorming, and history-based methods such as leveraging a risk taxonomy or list for risk identification [25, 26, 35, 40, 41]. By contrast, technical debt identification approaches deal with the problem of what the technical debt items are and where they reside. Identifying technical debt relies on assessment of software quality, which will be elaborated in section 2.3. The risk analysis step is discussed in more detail in section 2.2.3.

After the risks are identified and analyzed, actions should be taken to put the risks under control. Several risk control strategies, such as risk avoidance, risk transference, risk acceptance and risk mitigation have been proposed [63, 65, 66]. While controlling risk is concerned with choosing and then applying the best strategies for each of the identified risks, controlling technical debt generally involves one particular risk control strategy – accepting the risk because the potential benefit may outweigh the associated loss. More than a selected risk control strategy, controlling technical debt is a matter of tracking it to determine when and if the "acceptance" strategy is no longer cost effective, i.e. when the potential benefit no longer outweighs the cost.

Since the main parallels and similarities between software risk and technical debt are relevant in the second, or analysis, phase of the risk management process, we discuss in more detail approaches and findings related to risk analysis in the following

section. As argued above, existing approaches for identifying and controlling risks are of limited applicability to technical debt.

2.2.5 Risk Analysis

Among the risks that software projects may encounter, some are severe but have little chance to occur, while others may occur frequently with little impact. This raises the question of which risk should be handled first. In addition, constraints of software projects such as budget and schedule may not allow all risks to be handled and controlled. Moreover, the cost of eliminating some risks may be higher than the loss that the risks can incur. Therefore, the goal of risk analysis is to prioritize the risks so that software managers know where their effort should focus.

Risk analysis starts with estimation of the potential loss of the risk and the likelihood that the loss will occur, which are the primary elements of risk. Then, based on the results of risk estimation, the risks are assessed through different approaches.

2.2.5.1 Risk Estimation

The simplest metric for evaluating the potential loss of risk, i.e., risk impact, is to use an ordinal rating scale, which defines the impact as low, medium and high [62]. Then the severity of risk impact can be assigned a numerical value in terms of the context in which the risk is evaluated. For example, a risk with low impact in a project might mean \$1000 loss. However, evaluating the severity of the consequences is often quite difficult for immaterial assets such as loss of image of a company.

Another approach to severity estimation is to evaluate the recovery costs, e.g. time and efforts, of the impact in case of occurrence. In the area of software engineering, there are various models for software cost estimation, such as COCOMO [8] and Function Point Analysis [9]. These models leverage historical data and current project characteristics to achieve better estimation of effort for a project than pure human judgments. Using risk recovery cost as an estimate of risk impact is applicable for estimating technical debt in that the negative impact of technical debt is the extra effort required in future maintenance if the debt is not paid at present.

Similar to estimation of the potential loss of risk, probability of occurrence can also be roughly evaluated as high, medium and low [62]. The more accurate estimation takes a numerical value from 0 to 1 or a range such as 10% - 15%. If the evaluation is based on one's experience, then it is considered a subjective measurement.

Historical data is the major source that can be used for more accurate and objective estimation of the probability. Empirical analysis of historical data could discover the probability distribution of risk factors, which contribute to the occurrence of risk. Then risk models can be constructed. Inputs to the risk model usually contain uncertain variables or random variables. For any given set of input values, the model calculates outputs that are the impact of the risks - loss or benefit. Risk models can be used in several ways, but one effective way is to explore the possible outcomes using simulation. During simulation, inputs of the model are randomly generated from probability distributions to simulate the process of sampling from an actual population. The distribution chosen for the inputs should most closely match

historical data or best represent current state of knowledge. Finally the outputs are used to calculate the probability distribution of the risk impact.

Although historical data is helpful, it should be noted that determining the rate of occurrence may be difficult because statistical information is not always available on all kinds of past events. Moreover, statistical methods have a hard requirement for the volume of the data. If an event has rarely happened in the past, statistical estimation is not applicable. Therefore subjective estimation is still used in some cases.

There are a variety of techniques for subjective probability estimation. These techniques fall into three categories in terms of how they address the bias that the subjective measurement has. The first type of techniques estimates probability directly by assigning a label (e.g. High, medium or low) or a value (e.g. 30%) to it based on one's experience. This type of technique allows more bias in the estimation because the label can be interpreted subjectively and the value is artificial and does not usually reflect the real range of probability for a given risk [67].

The second type of technique assesses risk probability by providing values against which the probability of the risk occurrence can be compared [68]. In other words, it asks whether the probability of the risk is more, less, or the same as a given value. The aim of all these techniques is to adjust the comparator until the assessor cannot distinguish between the risk probability and the given value. This value is then used as the best estimate of the risk probability. Although they appear to be simple, the comparative approaches face difficulties such as the problem with understanding the

comparators. In addition, assessments using comparative techniques are subject to perceptual bias and heuristics as with the first type of techniques [69].

The third type of technique is an indirect approach. It involves coming up with the situations or scenarios that might occur for a given risk on a project. Each scenario is characterized by the state of a set of attributes and has an associated probability that the related risk will occur. Then the probability of the risk occurring can be inferred by finding which scenario represents the current scenario in which the risk is being analyzed [70]. For example, the technical risk of a software project may come from the use of a new development tool. That is, the risk is that the new tool will have problems and cause delays and higher costs on the project. The scenarios in which the risk occurs are characterized by the availability of alternatives for the tool and the quality of technical support provided by the vendor of the tool. Then one scenario is that alternative tools are available and technical support of the tool is excellent. The associated probability in this scenario can be estimated as “low” in terms of the state of these attributes. While another scenario is that no alternative tool is available and technical support is poor. The associated probability in this scenario is “high”, comparing with the previous scenario. We can follow the same way to develop all the possible scenarios and estimate the associated probability. Then estimating the probability that the risk will occur equates to finding the matching scenario. Assume that a new development tool is currently used in the project without alternatives, and the vendor of this tool provided poor technical support for their customers in the past year, then the technical risk of this project is considered as high. This approach has

the benefit of being less subjective than other assessments because it is based on known facts about the project rather than relying on subjective opinions.

2.2.5.2 Risk Assessment

The most straightforward risk assessment method is a risk analysis matrix, where both the impact of risk and the probability of occurrence are measured using an ordinal scale. The horizontal dimension of the matrix represents the probability of occurrence and the vertical dimension represents the severity of the impact. Then risks with catastrophic impact and high probability of occurrence are ranked high, followed by risks with catastrophic impact and moderate probability or critical impact with high probability, and so on, as shown in Figure 8 [62]. Once the risks are ranked, the high risks will be considered first. The limited resources will be used to deal with these high risks first, then medium risks and low risks. The advantage of this method is that it does not require a high level of measurement scale on risk impact and probability of occurrence, which is often the case during the initial stage of risk analysis or when more accurate estimation is not available due to lack of historical data.

Probability \ Severity	Very Likely	Probable	Improbable
Catastrophic	High		
Critical		Medium	
Marginal			Low

Figure 8. Risk Analysis Matrix [62]

Risk can also be prioritized in terms of risk exposure, a term used in Boehm's risk management approach [51]. Risk exposure is defined by the formula below:

$$\text{Risk Exposure} = \text{Prob}(\text{UO}) * \text{Loss}(\text{UO})$$

Where Prob (UO) is the probability that an Unsatisfactory Outcome (UO) occurs and Loss (UO) is the loss associated with the UO. The risk exposure, in terms of probability theory and statistics, is the expected value of the risk. Expected value is a well-established way of calculating uncertain events. The use of expected value in risk prioritization takes into consideration both the loss and probability of occurrence. Moreover, it can be used with different measurement units and scales, and the results can be easily aggregated and disaggregated [71]. However, expected value does not convey all the information in the primary two elements of risk, i.e., risk impact and probability of occurrence. For example, risks with very high potential loss and low probability of occurrence may be handled in a different way from those risks that occur frequently with little loss though the risk exposures of the two kinds of risks are equal. Therefore, this risk prioritization method should be combined with other techniques to reduce the bias of expected value.

One application of the theory of expected value in decision science is the decision tree. A decision tree uses a graph to model decision alternatives, states of nature, probabilities attached to the states of nature, and conditional benefits or losses. Then the expected loss/benefits for each decision alternative can be calculated. The goal of decision tree analysis is to select the best course of action in situations when facing uncertainty. Therefore it is often used for risk analysis.

Figure 9 illustrates a decision tree for a product development decision. The leftmost node (the root node) is a small square called a decision node. The branches emanating to the right from a decision node represent the set of decision alternatives that are available. One, and only one, of these alternatives can be selected. The small circles in the tree are called chance nodes. The number shown in parentheses on each branch of a chance node is the probability that the outcome shown on that branch will occur at the chance node. The right end of each path through the tree is called an endpoint, and each endpoint represents the final outcome of following a path from the root node of the decision tree to that endpoint. The expected values of decision branches are calculated from right to left. Decisions are made by selecting the branch that yields maximum benefit or minimum loss.

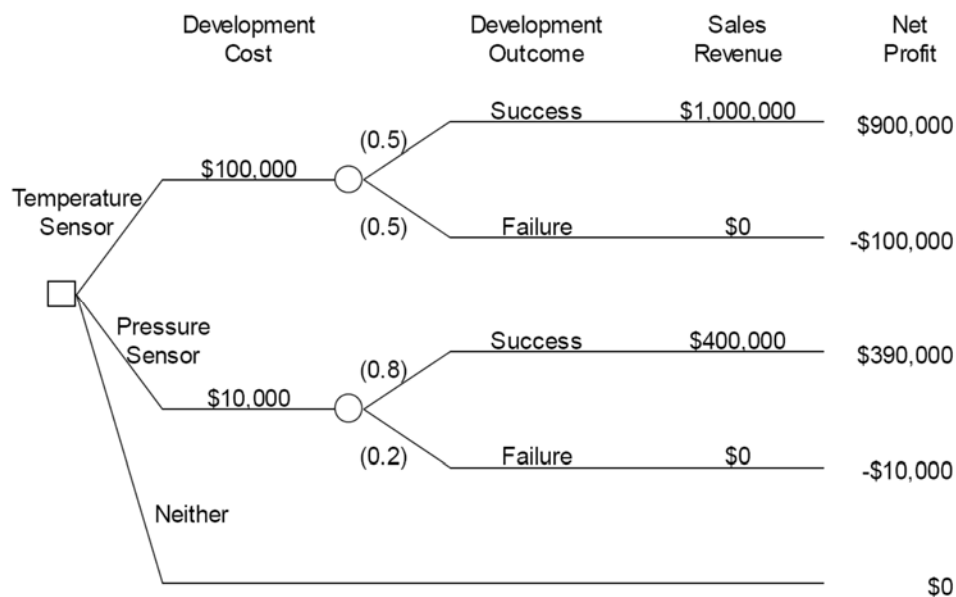


Figure 9. A Decision Tree [72]

Decision trees can be used to represent problems involving sequences of decisions, where decisions have to be made at different stages in the problem. However,

decision tree analysis assumes that probabilities of conditions are independent. This hinders application of decision tree analysis for complicated problems.

Another approach to risk prioritization is network analysis, which, in particular, refers to Critical Path Method (CPM) [73]. CPM is a project modeling technique, which uses a project network logic diagram to present all the activities required by the project, the time or duration for completing the activities and the dependencies among the activities. CPM analysis, as its name suggests, is to identify the critical path, a sequence of project network activities that add up to the longest duration, thus determining the overall project duration. Then the critical activities, which are those along the critical path, can be identified. If CPM is applied to risk management, the risks associated with the critical activities are considered having more impact on the project and should be given higher priority and more attention.

Risk exposure analysis, decision tree, CPM and other quantitative approaches provide more accurate and objective assessment of risk. However, the degree of accuracy and objectivity that these risk analysis techniques can achieve depends on how the primary elements of risk, i.e., potential loss and probability of occurrence, are estimated. In addition, risks often involve some combination of political, social, economic, environmental and technical factors, and therefore it is difficult to place a number on the risks when the factors are coupled with the obscure nature of personal perceptions [74]. The subjective risk assessment approaches have been proposed to overcome the shortcomings of the quantitative risk analysis methods. This type of approach uses human opinions and judgment to prioritize risks. For example, Delphi

Method [75] is an iterative process for consensus-building among a panel of experts who are anonymous one to another. Through several rounds of questionnaire and revision, consensus or stable results regarding the judgment can be achieved. When applying this method to risk analysis, the judgment they will make is the priority of risks, that is, which risks are more critical than others. The rationale behind this method is that group judgments can reduce the subjectivity introduced by human judgment and thus are more valid than individual judgments.

Analytic Hierarchy Process (AHP) is another method that can be used for risk prioritization. AHP, developed by Saaty in the 1970s, is a multiple criteria decision-making methodology [76]. The process involves structuring the decision problem with multiple choice criteria into a hierarchy, assessing the relative importance of these criteria through pair-wise comparison, comparing alternatives for each criterion, and determining an overall priority of the alternatives [76]. Based on mathematics and psychology, AHP helps capture both subjective and objective evaluation measures, providing a useful mechanism for checking the consistency of the evaluation measures and alternatives suggested by the team and thus reducing bias in decision making. Since software projects usually involve multiple stakeholders who may assess the same risk from different perspectives, software risk analysis may have to use multiple criteria. In such situations, AHP is a proper choice for risk prioritization in that it converts individual preferences into ratio scale weights that can be combined into a linear additive weight for each alternative. Thus the overall risk priority can be easily synthesized from the alternatives. AHP is also suitable for the situation that risk elements cannot be objectively measured, e.g., historical data are

not available or the statistics are not applicable, and hence significant personal judgment and subjective evaluation have to be used for risk prioritization.

2.2.6 Empirical Study and Evaluation

As the risk management approaches have been proposed and applied, researchers in this area paid attention to the performance of the approaches and started evaluating their effectiveness. Lyytinen et al. proposed a socio-technical model to synthesize a set of risk factors and resolution [77]. Then they used the model to analyze four classical risk management approaches. They concluded, through the study, that the four approaches differ significantly in terms of managers' role and possible actions. The implication of their study is that different risk management approaches may vary in emphases and applicability for addressing different risk management concerns. Therefore, software managers should be aware of these differences so that the appropriate approaches can be selected.

Addison and Vallabh conducted a study to examine whether software risks in software practice can be effectively addressed by the risk control strategies proposed in literature [78]. In the study they surveyed software project managers on the effectiveness of risk control strategies based on their experience. Meanwhile they compared the risk control strategies employed by experienced project managers with those used by inexperienced project managers. The results of the study confirmed the effectiveness of most risk control strategies proposed in literature. The study revealed differences of software risks in terms of impact and frequencies of occurrence, thus providing a guideline for software project managers.

Based on Boehm's approach [51], Heemstra and Kusters proposed a risk management method [79]. The method uses the risk checklist and involves group decision making and a risk advisor. The method was evaluated using five software projects. They conducted cost-benefit analysis on the proposed method and the results demonstrated the effectiveness of the method. The findings from this study also indicated the usefulness of a short risk checklist, group decision making involving all stakeholders of the project and a risk advisor, which are all emphases of this method.

Williams et al. evaluated SEI's risk management approach based on their experience with software-intensive United States Department of Defense (DoD) projects, which followed the SEI's continuous and time risk management [80]. In their work both the effective and ineffective aspects of the approach were discussed with some concrete examples from the projects. Although it was not a formal evaluation, the discussion offered insight into the applicability and effectiveness of the SEI's approach.

Software projects usually face a variety of risks. These risks influence software projects in different ways such as project schedule, software quality or the project cost. They are also different in terms of magnitude of the negative impact and probability of occurrence. One of the research topics in risk management revolves around the following questions:

- What risks are more influential to project success relative to others?
- Are there patterns to the risks in software projects?

Motivation of this research topic came from the belief that software managers should focus their effort on those risks that are critical to the projects. Some empirical studies

have been done to address such questions. For example, Wallace et al. studied risk factors in software projects with different levels of risk [81]. This study was based on their previous work on risk classification [82]. The study focused on risk patterns in software projects and the influence on project performance. In the study 507 software managers were asked to rate each risk statement from six risk dimensions that characterized their most recently completed project using a seven-point Likert scale. The ratings of the statements were aggregated to form the measures of these risk dimensions. Based on the aggregate measures, *k*-means cluster analysis was performed to group the projects as high, medium and low risk projects. The study revealed how different categories of risk contributed to the level of project risk, i.e., risk patterns. For example, one of the findings is that the requirements, planning and control risk categories are closely associated with high risk projects. Han and Huang conducted similar research to investigate the relationship between different categories of risk [83]. The study adopted the DoD's risk approach [40] to assess the risk factors. Data were collected through a web-based survey of software project managers, who were asked to rate the 27 software risks used in the DoD risk assessment method [40] in their most recently completed software project. The results showed that the requirement risk dimension is the primary area among the six risk dimensions, which is consistent with Wallace et al.'s findings.

Since software projects usually have multiple stakeholders, there are multiple dimensions and perspectives from which to view and define project success, such as low cost or a high quality product. According to the definition, software risks are the events that prevent the projects from achieving success. Therefore, the effectiveness

of risk control strategies depends on how they can affect project success. To study the relationship between risk control strategies and project success, Jiang and Klein surveyed 196 software project managers to elicit the types of risks encountered, the impact they have on different categories of project success, and the types of strategies that are deployed to mitigate known risks [84]. In the study they identified the patterns among the different categories of risks. Hypotheses on relationships between risks, control strategies and project performance were framed and tested based on the collected data. Results from the study indicate that the strategies involving behavioral aspects tend to be more influential in risk reduction than are those aimed at technical risks. Based on the findings they also gave suggestions for both practitioners and researchers.

Another topic in the area of software risk management revolves around risk issues and control strategies in a particular application domain. The goals of this research center on characterizing risks in the domain. For example, Conrow and Shishido's work provides concrete examples of risk management approaches applied to DoD projects and the effectiveness of the approaches [85]. Aubert et al. studied the undesirable outcomes of IT outsourcing [86] by using transaction cost and agency theory to analyze and assess the associated risk factors. By studying relationships between the undesirable outcomes and the risk factors, they identified the most influential risks in IT outsourcing. Similar studies were conducted to investigate risk factors in ERP projects [87, 88]. Through survey or case study, these studies revealed risk factors that are unique or critical to ERP projects, which deserve more attention and effort from software managers in the application domain.

2.2.7 Summary of Software Risk Management

Risk refers to the events that result in potential loss and have some probability of occurring in the future. The potential loss and the probability of occurrence are the primary elements of risk. A software project usually has many kinds of risks that jeopardize the success of the project. To control these risks and help the project achieve its goals, various risk management approaches have been proposed. These approaches either provide a general risk management framework, such as Boehm's risk approach [51] and SEI's risk management paradigm [63], or address a particular risk management problem such as risk probability estimation. Managing software risks follows the process of risk identification, risk analysis, risk control and risk evaluation. The goal of risk analysis is to estimate the risk impact and probability of occurrence and prioritize these risks base on the estimation. Risk impact can be estimated based on human experience and opinion. It can also be measured by the recovery cost. Therefore, approaches to software cost estimation can be used to estimate risk impact. Similarly, the probability of occurrence can be subjectively estimated. There are approaches to reduce the bias of subjective estimation. The use of historical data could improve estimation accuracy. After the impact and probability of occurrence are estimated, the risks can be assessed using quantitative risk analysis approaches such as risk exposure analysis and network analysis, or judgment-based approaches such as risk analysis matrix, DELPHI and AHP. These approaches vary in terms of applicable situation and performance. There are empirical studies regarding the effectiveness of risk management approaches, risk patterns in software projects and relationships between software risk and project performance.

This review is not comprehensive in that it does not cover all aspects of software risk management. Instead, it focuses on the aspects that are relevant to technical debt management. Since it possesses the primary attributes of a risk, i.e., potential loss and probability of occurrence, technical debt is considered as a particular type of software risk. Thus approaches to risk management can be adopted for technical debt management.

Actually the technical debt management approach we propose in Section 3 was inspired by software risk management. Managing technical debt using the proposed approach follows the same process as software risk management – identifying, measuring, controlling and evaluating technical debt. Considering the exploratory nature of the studies we've performed, we adopted simple risk analysis approaches to measure technical debt rather than apply sophisticated models during the initial stages of this line of study. In addition, the technical debt management framework presented in Section 3 is designed to be flexible and easy to incorporate results from ongoing study. Therefore, these simple approaches can be tailored, enhanced or even replaced with sophisticated approaches in the future when more details of technical debt management are uncovered and understood.

The principal, interest probability and interest amount of technical debt in our proposed approach are initially estimated as high, medium or low, as is often done in risk estimation. The estimation is refined with the aid of historical data whenever possible. We also use the risk analysis matrix for risk prioritization and risk exposure analysis in controlling technical debt. As implied by risk management, technical debt

should be continuously monitored to decide whether it is becoming more or less probable and whether the potential effects of technical debt have changed.

2.3 Software Quality Assessment

The extensive use, and increasing scale and complexity of software have raised concerns about software quality. It was reported that software organizations invest around 80% of their development resources for issues related to their products' quality [89]. Given the importance of software quality, quality assessment remains a hot research topic in the field of software engineering.

The concept of software quality has been defined in different ways in literature, e.g., extending the quality concept of hardware to software, or considering the characteristics unique to software [90-93]. Hereby software quality can be defined as “user satisfaction”, “measurable properties of software” or “fitness for use” [93]. In spite of differences in the definitions, software quality essentially means the degree to which software conforms to specified requirements and the degree to which software meets customer or user needs or expectations [94]. From the perspective of software quality, incurring technical debt may lead to software artifacts that fail to meet certain criteria. In this sense, quality assessment is the foundation for detecting technical debt. Studying approaches to software quality assessment could contribute to technical debt identification. Moreover, comparing the quality of immature software artifacts with the quality standard is an important approach to evaluating technical debt [42, 95].

2.3.1 Overview

Software quality is a multi-faceted concept in that the requirements that software should meet come from different groups of people, who have different views on software quality. For example, software users may focus on reliability and usability when they consider software quality, while developers may be more concerned about maintainability. A variety of models and approaches have been proposed to assess software quality. According to the goals they mean to achieve, these models are categorized as quality control models and prediction models [92]. Quality control models are used to check whether a software system meets predefined software quality criteria, to track software quality changes or to evaluate effectiveness of quality control and assurance activities. This type of model usually presents software quality as a hierarchical structure of characteristics, each of which contributes to total quality. The highest level of the characteristics is a set of attributes, such as reliability, usability, maintainability and testability. These attributes can be further decomposed into features or characteristics and sub-characteristics. Table 1 shows the quality model defined by ISO [96]. Other similar models that have been widely used are McCall's quality model, Boehm's quality model, and Dromey's quality model [97-99].

Software Quality					
Functionality	Reliability	Usability	Efficiency	Maintainability	Portability
Suitability	Maturity	Understandability	Time	Stability	Adaptability
Accuracy	Fault-tolerance	Learnability	Behavior	Analyzability	Installability
Interoperability	Recoverability	Operability	Resouce	Changeability	Replaceability
Security		Attractiveness	Utilization	Testability	Co-existence

Table 1. ISO 9126 Quality Characteristics

A typical software quality assessment process starts with identifying a small set of high level quality attributes, as defined in the ISO and IEEE quality standards [96, 100]. Then these attributes are further decomposed into more specific attributes at lower levels. The next step is to assign weights to each attribute. Since these attributes may conflict with each other, the weights assigned to them depend on the goals that the quality assessment means to achieve [90]. Then the value of each attribute is given a rating score. Finally the overall quality score is calculated by synthesizing the rating score of each attribute and the weight of the attribute. The key step of the process is to give each attribute a rating score, which is often performed by humans and thus subjectivity is inevitable. In some cases a positive attribute to one evaluator is negative to another. Therefore, software metrics have been proposed to address the subjectivity introduced by human judgments. The software metrics assess software quality attributes indirectly using software attributes that have objective measurements.

A quality model needs to be evaluated before being applied to quality assessment. Software quality models can be empirically evaluated at two levels. The first level of evaluation deals with validity of individual software attributes. The second level of evaluation is to validate the overall performance of the software quality model [101]. This second level of evaluation often starts with gathering metric data by applying the model to subject systems. Meanwhile the quality of the subject systems is evaluated by experts. Comparing the model results with the experts' opinion leads to the conclusion on the effectiveness of the model.

Another type of quality model focuses on software quality prediction, which uses product or process-related metrics to predict the fault proneness of a software system. Since identifying and fixing defects in later stages of software development or even after delivery may lead to high cost [102], it is beneficial to predict which modules are fault-prone in early stages of development so that limited resources can be devoted to the most problematic areas. However, some quality attributes such as reliability and maintainability are difficult to measure directly. Therefore one research direction in this area is to study the relationship between software quality and other measurable attributes such as software size and complexity. The goal of the research is to find effective software quality predictors – the measurable software attributes that are related to software quality. Various techniques have been used in software quality models. For example, Koru and Liu constructed their defect-prediction model using classification trees [103]. Ten- fold cross validation was employed to evaluate the performance of the model. Nagappan et al. used multiple linear regression analysis to model the relationship between software quality and selected software metrics [104]. Historical data from the prior projects were used to build the regression model for estimating the post-release field quality of the current project under development. They concluded, through empirical evaluation of the model, that the multiple regression approach is practical for measuring software post-release field quality and the model is effective in detecting low quality programs. Since the relationships between the measure of quality and measures of software attributes (software metrics) are often complex and nonlinear, the accuracy of conventional approaches is restricted. For this reason, neural network has also been adopted for

modeling non-linear functional relationships, which are difficult to model with other techniques. Khoshgoftaar introduced neural networks to model non-linear relationships in this area [105]. Fourteen structured metrics were selected as the input variables to the neural network and the total number of faults found in the programs was the output variable. In the model these authors used principal components analysis to eliminate the correlations among the product metrics and generate new metrics that have better performance on quality prediction.

Although quality assessment models may differ in goals and involved techniques, they are all based on objective measures of software artifacts. The transition from qualitative quality assessment to quantitative assessment using metrics reflects the expectation on measuring software quality more precisely. Therefore, software metrics are central to quality assessment.

2.3.2 Quality Metrics

A software quality metric is a quantitative measurement of the degree to which software possesses a given attribute that affects its quality [100]. Software quality metrics were developed for either predicting software attributes, e.g., the number of defects, or identifying the anomalous components whose characteristics fall out of normal range [92]. Since the concept of software quality involves software product and software process, quality metrics can be categorized as product metrics and process metrics. Product metrics measure the attributes of software products, e.g., complexity of a software module, while the process metrics measure software process performance or conformance. Within the product category, software quality metrics

can be classified in terms of the software lifecycle stage in which the to-be-measured software artifacts are created, e.g., metrics for requirement specification, design model, source code, testing, etc. [91]. As described early in this chapter, technical debt can also be classified in this way. Among the technical debt categories, defect debt and testing debt are obvious and easy to detect. Documentation debt may not be so obvious, but detection of documentation debt largely relies on manual inspection. Design and code debt may also be identified by manual review and inspection, but it is time-consuming and labor-intensive considering the large volume of involved artifacts such as source code. Therefore, using software quality metrics provides a cost-effective approach to technical debt identification in that it can be implemented automatically. In this sense, quality metrics for design models and source code are more important for technical debt management than the metrics in other categories. Design model metrics and source code metrics differ in the phase of software lifecycle when they are constructed. They are also applied to different software artifacts. Design model metrics are used for design documentation such as a class diagram, while source code metrics are applied to source code. There is overlap between the two types of metrics in that some aspects of design quality can also be derived and measured at the source code level. Moreover, quality assessment models based on source code metrics have been shown to outperform design metric based models [106].

2.3.2.1 Design and Code Metrics

The major problem of design debt is that it makes software less understandable, more complex, and harder to modify. In other words, design debt compromises software

maintainability. Therefore, metrics for evaluating software maintainability can be useful for this kind of technical debt identification. According to Boehm's quality model, software maintainability consists of three characteristics – testability, understandability and modifiability, which are further decomposed to sub-characteristics such as structuredness, conciseness and augmentability. These characteristics are closely tied to software complexity as it is intuitive that the more complex the software, the more difficult it is to be comprehended, tested and modified. A lot of complexity metrics have been proposed to measure software maintainability.

Lines of code (LOC) is a metric used to measure software size. Because software usually becomes more complex as its size increases, LOC is also an indicator of software complexity. However, LOC has been criticized for, among other things, being language-dependent and not considering program structure. McCabe proposed a complexity metric based on graph theory to measure the number of linearly independent paths in source code [107]. It was defined as:

$$M = E - N + 2P$$

where M is cyclomatic complexity (the cyclomatic number); E is the number of edges of the graph; N is the number of nodes of the graph and P is the number of connected components. M is computed using a control flow graph, which can be derived from the software program. Cyclomatic complexity is believed to be an indicator of understandability and testability of the software program.

Besides the internal complexity of software modules, which can be measured by size or cyclomatic complexity, the complexity of software is also affected by interactions among different modules of the software program. This type of complexity is measured by structure metrics [90]. Structure metrics are designed to measure the complexity resulting from interactions among modules, the degree of modularization, cohesion and coupling. Henry and Kafura proposed a complexity metric based on the principle that good design should use limited fan-in and fan-out [108]. The metric is defined as:

$$C_p = (fan-in * fan-out)^2$$

where C_p is the structural complexity of a module; fan-in is the number of parent modules of a module; fan-out is the number of child modules of a module. By combining the metrics for module complexity and structural complexity, software complexity can be measured on the system level. System complexity is one such metric. It is defined as the sum of structural complexity and data complexity, which measures how much information is being used in a method. [109].

The emergence of object oriented technology in the software industry has created new challenges in measuring software quality in that traditional software metrics, which focus on either data or function of software programs, are not sufficient for characterizing, assessing, and predicting the quality of object oriented software systems [110]. Therefore, metrics that reflect the specificities of the object oriented paradigm have been defined and validated. Since object oriented technologies use a different approach from the traditional functional decomposition and data flow

development methods, the object oriented metrics must focus on combining function and data as an integrated object rather than focusing on each one separately [111].

Object-oriented metrics are primarily applied to the concept of classes, cohesion and inheritance. These metrics can also be used as predictors of fault-prone classes and thus to determine whether they can be used as early quality indicators. Since object-oriented techniques gained popularity in software development, many quality metrics specific to object-oriented design and programming have been proposed. Here are some of the widely used metrics defined in literature [112].

- **Weighted Method Per Class (WMC):** the count of the methods implemented within a class or a sum of complexities of the methods (measured by cyclomatic complexity). As WMC increases, the class will have more potential impact on the child classes. Moreover, classes will become more specific, thus limiting the degree of reuse. Therefore WMC is an indicator of software maintainability and reusability.
- **Lack of Cohesion of Methods (LCOM):** the degree of similarity between methods. The similarity of methods is defined as the number of methods that access the same attribute(s). High cohesion indicates high class subdivision. Lack of cohesion increases complexity, thereby increasing the likelihood of errors in the development process. This metric evaluates efficiency and reusability.
- **Number of children (NOC):** an indicator of the influence of a class on a system. As the number of children increases, the likelihood of improper abstraction of the parent becomes greater, but it also leads to greater

reusability. Although inheritance can reduce the complexity by reducing the number of operations and operators, it makes maintenance and design difficult [111].

Since a single metric can only focus on one particular aspect of software quality, these metrics each have their own advantages as well as shortcomings regarding measurement. Thus, a combination of different metrics could give a more balanced view of the overall quality of software. Of course, the conflicts among the metrics must be addressed before combining them. Actually more sophisticated metrics have been developed and used for software quality assessment or quality-related purposes, e.g., code smell detection[113]. Code smells refer to the patterns in source code that indicate poor programming practices or code decay. Therefore code smells are promising as symptoms of technical debt, in particular design debt. Approaches to code smell detection will be presented in section 2.3.3.

2.3.2.2 Metric Evaluation

To gain the insight required for understanding and evaluating software quality, a large number of metrics have been developed. Many software quality attributes can be measured by multiple different metrics. For example, the size of a software system can be measured by Lines of Code or by Function Points – units of measure that represent the functional size of a software application. This has created the need to determine which set of metrics is most effective for measuring a particular attribute in a particular context. Several studies have been conducted to correlate software metrics with quality or validate the significance of the quality metrics proposed in the literature.

Weyuker evaluated software complexity metrics in her study [114]. Through this study she proposed a set of abstract properties that a sound metric should satisfy. For example, one property is that “a measure that rates all programs as equally complex is not really a measure” [114]. These properties were presented as formulae and formed an axiomatic model. Based on the analysis of four well-known complexity metrics, she claimed that this model provides the foundation for comparing and evaluating software complexity metrics in a formal way. Inspired by Weyuker’s approach, Tian and Zelkowitz came up with another axiomatic model and proposed a complexity metric selection technique [115]. The proposed model formulates the selection problem as a constrained optimization problem. Thus the metrics can be evaluated by determining the feasible region and then finding the optimal solution.

The metric evaluation approaches using axiomatic models fall into theoretical validation category. By contrast, there is another category of evaluation approaches that empirically validate if an internal metric is associated with an external attribute. Only if the association with the external attribute is demonstrated should the metric be used in models for measurement and prediction. Therefore some researchers are concentrating on revealing or validating the relationship between internal metrics and the corresponding external attributes that they intend to measure. Emam et al. investigated the effect of class size on the validity of object-oriented metrics in their study [116]. To demonstrate the confounding effect of class size, they examined the association between object-oriented metrics and fault-proneness of the classes using data with and without size control respectively. A regression model was built for hypothesis testing. The analysis results supported their argument that class size is a

strong confounder, hence it is necessary to re-examine the conclusions of previous work made by other researchers. Koru and Tian's study investigated the relationship between structure metrics and the change-proneness of software modules [117]. They used ranking and a tree-based clustering technique to identify the two types of modules, high-change and low-change. Through statistical analyses they concluded that high-change modules are different from those with highest measure values. Their findings provide guidance for software practitioners to identify change-prone modules, though some of them are contrary to common intuition. Basili et al. conducted an empirical study to evaluate the effectiveness of a set of object-oriented design metrics in prediction of fault-prone classes [110]. Their study demonstrated that most of the metrics are effective predictors of fault-prone classes in early phases of software development. The study also revealed that the object-oriented metrics outperformed the traditional code metrics.

2.3.3 Program Analysis and Code Smells

Program analysis refers, in general, to any examination of source code that attempts to find patterns or anomalies thought to reveal specific behaviors of the software. Some types of program analysis focus on patterns that indicate poor programming practices and bad design choices. Such patterns, termed "bad code smells" [118], are believed to cause maintainability problems over time because they make the software less understandable, more complex, and harder to modify. Code smells can be categorized in terms of difficulty of detection.

- **Primitive smells** are violations of predefined rules and development practices and can be directly detected based on the source code. An example of

primitive smells are violations of coding rules that define how source code should be structured on different levels or which control structures should be used. A rule such as “one Java source file should not contain more than one Java class” can be translated into a code smell by constructing a rule statement out of its negative. Program analysis is able to detect these violations effectively.

- **Derived smells** are higher level design violations such as high coupling degree among modules. One approach to detecting this type of code smells is a metric-based approach, which uses complex computations and extractions of facts from source code. They can be computed based on Boolean expressions that include code metrics and thresholds [113]. An example of a derived smell is a class that implements too much responsibility, named a “God class”. The following expression was proposed to evaluate if the “God class” smell is present:

$$WMC > 47 \ \& \ ATFD > 5 \ \& \ TCC < 0.33$$

where WMC is the weighted methods per class, ATFD is the number of accesses to foreign class data, and TCC is tight class cohesion. Detecting this type of code smell requires a rigorous definition of the used metrics (e.g. how are methods weighted when computing WMC?) and a set of baselines to define the included thresholds.

- **Manually detected smells** can only be discovered by a human inspector. A typical example is the quality (not the amount) of documentation present; a

computer can neither judge if the documentation is easy to understand nor if it fits the actual implemented code statements.

Many bad smells, and the rules for automatically detecting them, are defined in the literature [118]. Examples of well-known bad smells include duplicated code, long methods, and inappropriate intimacy between classes. For many of these rules, thresholds are required to distinguish a bad smell from normal source code (e.g. how long does a method have to be before it smells bad?). With the rules and corresponding thresholds, many code smells can be detected by automatic means, and several smell detection tools exist [119].

2.4 Software Effort Estimation

Software cost is a critical factor in software project management in that many activities, such as project planning, contract negotiation and software pricing, are based on cost estimation. In technical debt management the decision of whether to keep or pay off debt also depends on the present and future cost of the debt, which can be measured by the effort required to pay off the debt. Therefore, cost estimation is essential for managing technical debt. Reviewing the cost estimation approaches could facilitate choosing methods for technical debt estimation that are appropriate for a domain, and that can be applied usefully in a given context.

Since software development costs are primarily the cost of involved effort, approaches to cost evaluation focus on effort estimation. Software development effort is usually measured using person·day, person·month or person·year. Then the development cost can be computed as the effort multiplied by labor cost per time unit.

According to the availability of the required resources, the project duration and schedule can also be estimated.

Research on software effort estimation started in the 1960's. Since then various models and techniques have been proposed. Effort estimation approaches can be categorized as algorithmic methods, which use mathematical formulae to model software cost, or non-algorithmic methods [120]. The category of non-algorithmic methods includes analogy-based estimation methods and expert estimation methods.

2.4.1 Software Size Metrics

Most algorithmic methods use size as the primary input variable. Although there are lots of factors affecting effort estimation, software size is considered the most influential one. Here are some widely used metrics for sizing software.

Lines of Code (LOC) is not consistently defined. It may refer to the number of lines in the text of the program's executable source code (physical LOC), or the number of executable statements (logical LOC)[121]. Different LOC measuring tools may use different definitions of LOC. Although LOC is easy to obtain, it is only available after the software construction phase in the software lifecycle. Therefore, LOC needs to be estimated if it is required in an early stage of the software lifecycle. In addition, LOC is language-dependent. That is, the same functionality implemented in different programming languages has different LOCs.

Code churn is a LOC-based metric. It refers to the number of lines added, modified or deleted to software source code from one version to another [122]. It has been used

in many studies [123-125] as a common surrogate measure of software maintenance effort. There are also studies about the validity of this measure for software change effort. To leverage the change information for software effort estimation, Graves and Mockus developed a model with a set of factors including the size of a change [126]. Then they conducted regression analysis to validate the model using the change history of a large software system in a period of 45 months from the configuration management system. Based on the results of the study, they concluded that the size of a change, which can be measured by code churn, is a critical contributor to change effort. In Mockus and Votta's study 170 maintenance tasks in a large software project were investigated to identify reasons for software changes [127]. Through this study they also found strong relationships between the size of a change and the time required to carry it out. A recent study by Sjøberg et al. claimed an even stronger correlation (Spearman's $\rho = 0.59$) between code churn and maintenance effort [128]. However, this strong correlation has not been supported by all studies. For example, after analyzing 98 corrective maintenance tasks in a 40K LOC software application, Emam found code churn is only moderately correlated with change effort [129]. He further explained that the result is consistent with common sense as some difficult changes, e.g., corrective change, could consume considerable amount of effort, but result in very few changes in terms of LOC. Sjøberg et al. discussed the possible reasons for these differences in their study [128]. One of the reasons they gave was that different types of maintenance tasks, e.g. corrective vs. adaptive task, were used in these studies, which may require different number of lines of change per unit of effort. Based on the results of their study, they further argued that code churn is a

reasonable surrogate for non-corrective maintenance effort when real effort data is not available.

Function points is a measurement of the number of functions that a software system provides [130]. This metric is based on the functionality of the program and was proposed by Albrecht [131]. The total number of function points in a program is computed by measuring the following program features: external inputs and outputs, external interfaces, and internal and external files used by the system. Each of the features is given a weighting value that varies from 3 (for simple inputs) to 15 (for complex internal files) in consideration of different complexity of the features. Then the unadjusted function-point count (UFC) is computed as

$$UFC = \sum (\text{number of elements of given feature}) \times \text{weight}. \quad (1)$$

This unadjusted function-point count is either directly used for cost estimation or is further modified by additional factors affecting the overall complexity of the system. The adjusted function-point count takes into account the degree of distributed processing, the amount of reuse, the performance requirements, etc. The final adjusted function-point count for the system is the product of the UFC and these project complexity factors. Since function-point measurement is based on requirement and design specifications, it is independent of implementation language and can be obtained in early stages of the software lifecycle. However, the complexity weighting is subjective because it depends on the estimator. In addition, the measurement may be biased when the system is dominated by some features, e.g., a database system where input and output operations are dominant [132].

Function points can also be used to estimate LOC. If historical data are available, the average number of LOC per function point can be estimated. Then LOC is computed as the average number of LOC per function point multiplied by the number of function points.

Object points is a measurement based on the number and complexity of screens, reports and components written in 3G languages [133]. Each of these objects is counted and given a weight ranging from 1 (simple screen) to 10 (3GL component) and the object point is the weighted sum of all these objects. Similar to function point measurement, object points can be used in early phases of software lifecycle. In addition, this measurement is easy to use as it is only concerned with the three types of objects.

2.4.2 Effort Estimation Techniques

Algorithmic methods, as mentioned previously, use mathematical algorithms to produce effort estimates. Putnam's model [134] and COCOMO [8] are examples of algorithmic methods. Non-algorithmic methods include analogy-based estimation methods and expert estimation methods. As its name suggests, expert estimation methods involve consulting one or more experts and the estimates are based on experts' opinion and experience. Delphi [75], Top-down, Bottom-up and Parkinson's law [135] belong to expert estimation. The algorithmic estimation and the two categories of non-algorithmic estimation methods are presented and reviewed in the following sub sections, along with introductions of some representative methods from each category.

2.4.2.1 Algorithmic Methods

Algorithmic methods use mathematical formulae to model software cost. In the models the cost estimate is a function of a number of cost factors/drivers. A typical algorithmic model can be expressed as:

$$Effort = a \times Size^b \quad (2)$$

where a is a constant representing differences in local environments under which the software is developed, $Size$ refer to the code size of the software, b is the exponent of software size, indicating the non-linear relationship between software size and the effort. a and b are estimated based on analysis of historical data extracted from past projects. Since algorithmic models provide a way to generate a scientifically based estimate, they have attracted many researchers working in the area of software cost estimation. A number of algorithmic methods have been proposed, e.g., Putnam's model [134], COCOMO [8] and Function Point Analysis [136].

Among the algorithmic methods, the most famous one is COCOMO, which was developed by Barry Boehm and launched in 1981 [8]. It is often mentioned as COCOMO 81 to differentiate from its follow-up, COCOMO II [137]. In the model the code size is measured in thousand LOC (KLOC) and effort is measured in person·months. COCOMO applies to three classes of software projects: simple and well-understood projects, more complex projects and embedded projects. The model consists of a hierarchy of three increasingly detailed and accurate forms.

Basic COCOMO models software development effort and cost as a function of program size. a and b are assigned different values for different types of projects.

Basic COCOMO is good for quick estimates of software costs, but its accuracy is limited because it does not consider factors such as hardware constraints, personnel quality and experience. Intermediate COCOMO computes software development effort as a function of program size and a set of cost drivers that include subjective assessments of product, hardware, personnel and project attributes. This extension considers a set of four “cost drivers”, each with a number of subsidiary attributes. Each attribute receives a rating on a six-point scale that ranges from “very low” to “extra high”. An effort multiplier is then applied to the rating. The product of all effort multipliers results in an effort adjustment factor (EAF). Thus the Intermediate COCOMO formula is defined as

$$E = a (KLoC)^b \cdot EAF \quad (3)$$

The Detailed model incorporates all characteristics of the intermediate version, but adds an assessment of each cost driver's impact in each phase of the software process. Thus an estimate based on the detailed model is able to account for the influence of individual project phases.

Compared with COCOMO 81, COCOMO II provides more support for modern software development processes. In the new model the simple, complex and embedded project classification has been replaced with Precedentedness, Development Flexibility, Architecture or Risk Resolution, Team Cohesion and Process Maturity, reflecting the consideration of more modern factors and allowing a fine tuning to the exponent. COCOMO II employs the Application Composition, Early Design, and Post-Architecture models in place of Basic, Intermediate and

Detailed models, thus differentiating various stages in a project when each of the models would be more appropriate.

Algorithmic methods use cost-estimating relationships to associate the system characteristics with the estimates of effort and duration. The relationships are represented as a and b , the coefficients in formula (3). The coefficients are computed based on historical data using statistical analysis techniques. Regression analysis is one such technique that has been widely used for constructing cost estimation models. In spite of variations in regression analysis techniques, the basic idea behind them is the same – finding the trend (relationship between independent variables and the dependent variable) that best fits the historical data, which is measured by overall deviation of estimated values of the dependent variables from the actual values. The ordinary least squares regression analysis has assumptions such as uncorrelated independent variables and a linear relationship, which restricts the use of this technique. To address the limitations, advanced regression analysis methods have been proposed. For example, the robust regression analysis uses only the data points lying within two (or three) standard deviations of the mean response variable to eliminate the impact of data outliers on the model [138]. Other techniques are also applied in conjunction with regression analysis to better model software cost. For example, a Bayesian approach was used to calibrate COMOMO II and an improvement in performance has been demonstrated [138].

A lot of studies have been conducted to evaluate algorithmic cost estimation models [130, 139-144]. Although findings from these studies are not completely

consistent or comparable due to differences in development environments, sources of historical data and subject systems, they all confirmed that algorithmic models require calibration for a particular situation to improve the accuracy of estimation.

Calibration is the process of determining the deviation from a standard in order to compute the correction factors. For cost estimation models, the standard is considered historical actual costs. The result of calibration is the adjusted cost factors whose values reflect local circumstances under which the cost is estimated.

One of the advantages of algorithmic models is that they are able to be calibrated. Through calibration the model can reflect the specific situations where the model is used. Thus it can yield more accurate and objective results. In addition, the use of mathematical formulae in the models produces repeatable results, makes it easy to modify input data, refine and customize formulae and hence facilitates sensitivity analysis. However, the calibration is based on past project data. If the situations, e.g., the development process, change and differ from the situation under which it is calibrated, the model will be subject to significant estimation error without recalibration [8]. Another problem of algorithmic models is that software size, the major input of the model, can only be roughly estimated with significant subjectivity in the early stages of the software lifecycle. As a result, this leads to inaccurate cost estimation. Also, the requirements for a minimum number of historical data points needed for many statistical analyses increases the difficulty of applying algorithmic models. This type of model cannot achieve its potential usefulness when the organizations do not have enough data from past projects, or not all the attributes are

properly measured, which is often the case. Furthermore, effort involved in model calibration may hinder the application of algorithmic methods in practice.

2.4.2.2 Analogy-based Estimation

Analogy is a cognitive process and has been used as a problem-solving method for a long time. The basic idea of this method is to use solutions to similar past problems for solving new problems. When applied to software cost estimation, it refers to the process of comparing the target project to previously completed similar projects with known effort to derive the cost estimates. This approach was formalized as a cost estimation methodology by Shepperd and Schofield [145]. They addressed the major problems of the analogy-based method – how to characterize software projects and how to measure similarity. In the proposed method, the selection of project characteristic is based on expert opinion and data availability. The similarity between projects is measured by Euclidean distance in n-dimensional space where each dimension corresponds to a characteristic of the projects. Estimation by analogy starts with characterizing the proposed project. Then the most similar completed projects in terms of the selected characteristics are identified. Finally the cost of the proposed project is derived from these projects whose cost information is stored in the historical database.

Compared with algorithmic estimation methods, analogy-based estimation is easier to understand and implement. Another advantage is that analogy-based estimation is based on data from actual projects, thus avoiding reliance on expert recall [146].

Some evaluation studies have demonstrated that the performance of analogy-based estimation is comparable or better than algorithmic methods [145, 147].

The major problem of this approach lies in measurement of similarity between the source projects and the target project. It is not clear what attributes should be used to characterize projects and how these attributes contribute to cost estimation. Therefore, this method should be avoided if there are significant differences in important characteristics between the source projects and the target project. Data availability may also be a problem because few organizations have enough data on past projects and it is rare that two or more software projects are very similar given that software technologies and development environment are changing rapidly.

2.4.2.3 Expert Estimation

Expert estimation is another category of cost estimation approaches. This type of approach uses experts' experience to provide cost estimates. The experience includes the knowledge of development efforts gained from the projects they have participated in the past. Based on their experience experts can make an estimation of the cost of the target project. Expert estimation approaches can be considered an educated guess about the effort required to develop an entire project or a portion of it [146]. Since expert estimation mainly relies on experts' experience, the estimation results may be subjective and have bias. To overcome this downside, experts usually give three estimates – the lowest possible value, the mostly like value and the highest value of the cost. The final cost estimate is the expected value computed based on these three values [91]. This is computed as:

$$S = (S_{opt} + 4S_m + S_{pess}) / 6 \quad (4)$$

where S is the estimated software size, S_{opt} is the optimistic estimate of size (the lowest possible value), S_m is the most likely size and S_{pess} is the pessimistic estimate of size (the highest possible value). Beside the expected value approach, other methods have been applied in expert estimation to reduce the subjectivity introduced by human judgment and to increase the accuracy of estimation.

Wideband Delphi is a group consensus technique proposed by Boehm [8] based on the Delphi method, which has been described in section 2.2.5.2 in the context of risk analysis. The major steps of Wideband Delphi are the same as those of the standard Delphi method. The difference between the two methods is that Wideband Delphi introduces group discussion meetings in each round of the iteration. The experts can exchange ideas directly with one another in the meeting. Moreover, the experts do not need to give the rationale of their estimates. Therefore, Wideband Delphi facilitates communications among the experts and reduces the time for convergence of different estimates. It leverages the free discussion advantage of group meeting techniques, while keeps the advantage of anonymous initial estimation of the standard Delphi method [8].

Top-Down method starts estimation at the system level. A cost estimate of the overall project is made firstly by considering the project characteristics such as development process and functionality. Then the estimated cost is assigned to components and sub-components according to the project and system structure. Since the cost estimation starts from high level, this method takes into account some costs associated with the

overall system rather than any individual element, e.g., system integration cost. This is one advantage of the method. The main disadvantage of this method is that it may overlook some components that significantly affect project effort and cost. As a result, the overall cost may be underestimated. In addition, the estimate of overall project cannot be justified because there are no details on how the estimation is made [92]. The top-down method is suitable for cost estimation in the early stages of the software lifecycle because it does not rely on the system hierarchy, which is not available until the detailed design phase.

Bottom-up method takes the opposite direction in the course of cost estimation.

Therefore its advantages and disadvantages are complementary to the advantages and disadvantages of the top-down method. The bottom-up method starts with estimating cost for individual elements of the project. The estimates of the individual elements are summed up to form the estimate of the overall project. To work at the component level, the project needs to be decomposed hierarchically. Therefore the significant cost drivers associated with particular elements could be identified and taken into consideration when estimating the cost, but some costs at the system level may be missed. Since system decomposition is a pre-condition, this method is difficult to apply in the early stages of the software lifecycle.

Besides the subjectivity and bias involved, expert estimation also has other disadvantages. Compared with algorithmic methods, expert estimation cannot produce repeatable results. Therefore, sensitivity analysis (where the reliance of the estimate on small changes in the parameters) cannot be performed because

differences in the outcome due to changes in the inputs cannot be separated from differences due to variation among the experts. Such sensitivity analysis is often an important step in effort estimation. In addition, the quality of estimates derived from expert judgment is limited by the estimators' experience and objectivity [8].

However, expert estimation has some advantages, which are exactly the weaknesses of algorithmic methods. Expert estimation does not require historical data. It can integrate exceptional cost factors into expert experience when making estimation. Therefore, expert estimation is suitable for the situation where the proposed project has unique characteristics, such as using new technology or a new development process, and no cost data from past projects can be used for model calibration. It has been demonstrated that formal models not calibrated to the local circumstance do not perform well, while, on average, expert judgment is able to compete with algorithmic models in terms of accuracy of cost estimation [11].

2.4.3 Summary of Cost Estimation

Cost estimation is an important part of software project management. A huge body of research work has been done in development and validation of cost estimation approaches with the goal to improve the accuracy of estimation. However, software cost estimation is still an immature area in that some basic issues, which are prerequisites for developing cost estimation methods, have not been addressed [148]. There is no single approach to software cost estimation that can fit all circumstances. Each approach has advantages as well as shortcomings. The performance of a cost estimation model depends on the context in which it is applied [149]. Therefore,

selecting an appropriate cost estimation approach relies on the knowledge about the “home ground” where the approaches perform best. If no single approach is satisfactory, different approaches may be used together as a combination of approaches could improve the accuracy of estimation [138]. When making the decision, one should take into consideration other factors such as the usability of the approach and the implementation costs.

In a general scenario of technical debt management, paying off debt involves implementing a maintenance task. In this sense selecting an approach for technical debt estimation is nothing more than choosing a cost estimation approach for software maintenance projects. However, in our research, the best choice is the estimation method currently used in the project even if it’s not the most accurate estimation approach. The reason for the choice is that our study designs should avoid introducing new factors that may have effects on the evaluation object, and changing the cost estimation method during the study could have such effects.

2.5 Summary of Literature Review

Although the software maintenance problem described by the technical debt metaphor has been present for a long time, formal research on technical debt started just a few years ago. The raised attention of the research community to this topic came from pervasive discussions on technical debt in the software industry. Early work in this area focuses on establishing the foundation of technical debt research, including formalization of the definition, classification, analysis of the causes and effects. The discussions provide multiple perspectives from which researchers can understand the

essence of the technical debt problem and are helpful in framing our research questions. The foundational research work on technical debt inspired us in study design and preparation. Therefore, in this chapter we first presented the discussions on technical debt from the software community, which was followed by a review of the foundational research on technical debt. Then we reviewed the literature from three related research areas – software risk management, software quality assessment and software effort estimation. With the goal of finding usable research results for technical debt management, the review of literature in each area starts with analysis of the relationship between technical debt and the area. Technical debt is treated as a particular software risk due to its negative impact on software projects and the uncertainty involved. Technical debt can be identified through software quality assessment because technical debt refers to the software artifacts with immature quality. Quantification of technical debt relies on software cost estimation approaches because risk can be measured by its recovery cost. Based on analysis of the relationships, the relevant concepts, processes and approaches can be identified from these research areas. Review of these processes and approaches focuses on their applicability to technical debt management. In addition, findings from empirical evaluation are also presented to highlight the effectiveness of the approaches and the conditions they should satisfy. The outcome of the literature review includes processes and approaches that can be adopted for technical debt management. These include the risk management process and risk exposure analysis, cost estimation methods that can be used for technical debt estimation under different circumstances, and the concepts that inspire new ideas on technical debt management such as code

smells. In general, research in the three areas provides the foundation and a comprehensive toolkit for investigation of the technical debt problem.

Chapter 3: Research Overview

The use of the technical debt metaphor provides an intuitive way to describe the tradeoff associated with delaying software development and maintenance tasks. It has helped software practitioners discuss and understand the problem. Models and approaches have also been developed in recent years to facilitate its management. However, most of the approaches focus on identification of technical debt. Few models or approaches directly address the core problem of technical debt management, that is, how technical debt information can be used to facilitate software decision making. Moreover, fewer of them were empirically evaluated. Therefore, the effectiveness of these approaches, or even the effectiveness of the idea about managing technical debt in an explicit manner, haven't been validated. The goal of this research is to reveal, based on a series of studies on technical debt in real software maintenance projects, the cost and benefit characteristics of explicit technical debt management, refine the management approach we proposed and empirically validate the effectiveness of the approach. The studies were carried out to answer the following research questions.

RQ1: What factors contributed to the costs and benefits of measuring and monitoring technical debt?

RQ2: In what ways does technical debt information contribute to decision making?

In this chapter we first introduce the proposed approach, a technical debt management framework, in section 3.1. An overview of this framework, including the main

components, the implementation process and the characteristics, is described at the beginning of this section. Following the overview are the approaches to identifying, measuring and monitoring technical debt and how they can be used in the proposed framework, which are presented in three subsections of section 3.1. Then two scenarios are given to illustrate the use of this framework in decision making, followed by a summary of the framework at the end of section 3.1. The proposed approach presented in section 3.1 serves as an initial framework with which to start investigating the technical debt management problem in the context of the studies. The research design for uncovering the cost and benefit characteristics of explicit technical debt management is presented in the next section, i.e. section 3.2. This section begins with an explanation of our research design choices and the subject projects. Then two study designs, the retrospective study and the case study, are described in section 3.2.1 and section 3.2.2 respectively. Detailed procedures and the data required by the studies are elaborated in the sub-sections of section 3.2.1 and section 3.2.2. Then this chapter ends with a brief revisit of the proposed management approach and the research design.

3.1 The Proposed Technical Debt Management Approach

The purpose of the proposed approach is to provide an initial technical debt management framework that can be used as a measurement instrument for our studies, as well as be expanded and tailored according to the ongoing results of the studies described in the next section of this chapter. This framework is designed to be flexible and to incorporate human judgment at all stages. In this way, it can easily be modified to incorporate results, understanding, new models, and emerging theories.

The proposed approach to technical debt management centers on a “technical debt list.” The list contains technical debt “items”, each of which represents a task that was left undone, but that runs a risk of causing future problems if not completed. Since incurring technical debt is often used as a strategy by project managers [16], a technical debt item can also refer to technical debt candidates, i.e. tasks that the project manager thinks can be deferred. Examples of technical debt items include modules that need refactoring, testing that needs to be done, inspections/reviews that need to be done, documentation (including comments) that needs to be written, architectural compliance issues and known defects that need to be fixed, etc. Each item includes a description of what part of the system the debt item is related to and why that task needs to be done, and estimates of the principal and the interest. As with financial debt, the principal refers to the cost to eliminate the debt (the effort required to complete the task for a technical debt candidate). The interest is composed of two parts. The “interest probability” is the probability that the debt, if incurred and not repaid, will make other work more expensive over a given period of time or a release. The “interest amount” is an estimate of the amount of extra work that will be needed at some point in the future if this debt item is incurred and not repaid. For example, the interest probability for a testing debt item is the probability that latent defects exist in the system that would have been detected if the testing activity had been completed. The interest amount would be the extra work required to deal with those defects later in the system’s lifetime, as compared to what it would have cost if they had been found during testing.

ID	37
Date	3/31/2008 (Release 3.2)
Responsible	Joe Developer
Type	Design
Location	Method m in Module X
Description	In the last release, method m was added quickly and is thread-unsafe.
Estimated principal	Medium (medium level of effort to modify m)
Estimated interest amount	High (if we wait to modify m , there might be more dependent modules that need to be modified and thus require more extra effort)
Estimated interest probability	Low (not likely to be adding simultaneous calls to m)

Table 2. Technical Debt Item

Table 2 shows an example of a technical debt item. This debt item has a unique identifier 37, was incurred in Release 3.2 and is owned by developer Joe. Due to a rushed modification in the last release, Method **m** in Module **X** is currently thread-unsafe, which means that simultaneous calls to this method have unpredictable consequences and may result in system failure. Therefore this situation is identified as design debt and calls for refactoring of Method **m**. It is possible that new modules depending on **m** will be added to the system at some point in the future. If refactoring **m** is postponed, more dependent modules may need to be modified, thus requiring more work than would be required to refactor method **m** now. The extra effort required to modify those modules is the interest on this debt item. The principal, interest amount and interest probability were estimated as medium, high and low respectively. This can be translated as meaning that refactoring **m** requires a “medium” amount of effort, the extra effort required to adapt future **m**-dependent modules if refactoring **m** happens later is estimated to be “high”, but the probability that we will be forced to refactor **m** (e.g. when simultaneous calls to **m** become possible in the design of the system) is “low”. Although this is a very coarse-grained

estimation, it is sufficient in the initial stages for tracking technical debt items and making preliminary decisions.

The process of managing technical debt using this approach starts with detecting the technical debt items, including the tasks that are left undone as well as the tasks that may be deferred, i.e. technical debt candidates, to construct the technical debt list.

The next step is to measure the debt items on the list by estimating the principal, interest amount and interest probability as low, medium or high. Then the debt items are monitored and decisions can be made on when and what debt items should be paid or deferred. The technical debt list must be reviewed and updated after each release, when items should be added, removed, or their values should be re-estimated as the circumstances change. That is, technical debt should be continuously monitored. For example, for a technical debt item carried over from last release cycle, its principal should be re-estimated if part of the work to pay off this debt item has already been planned in the current release. The following subsections will describe how to identify, measure and monitor (including in decision making and release planning) technical debt.

3.1.1 Identifying Technical Debt

The first step of technical debt management is identifying technical debt. There are different types of technical debt [26]. They can be identified by different methods. Program analysis, which has been discussed in Section 2.3.3 is one approach to detect design debt. With program analysis, the source code of the subject software is statically analyzed to find code smells – patterns that indicate poor programming

practices or violations of the system architecture. Various techniques have been developed for different code smells, such as the primitive smells and derived smells, but some code smells have to be manually detected, as described in Section 2.3.3. Besides static analysis, dependency analysis techniques, such as the design structure matrix method, have been used to detect architecture violations [150]. Design debt can also be identified by inspecting code compliance to standards. Testing debt can be identified by comparing the test plan to test results. The tests planned but not run are testing debt items. Testing debt can also be identified by finding deficiencies in the test plan or test suite, e.g. low code coverage. Defect debt can be identified by comparing test results to change reports, or examining the bug repository for open bug reports. The defects that are found but not fixed are defect debt items. By comparing change reports to documentation version histories, documentation debt can be identified. If code changes are made without accompanying changes to documentation, the corresponding documentation that is not updated is documentation debt. These types of debt can be identified from different sources, such as the code repository and defect database, where software artifacts or project information are stored. Developers themselves can also add technical debt items based on their own understanding of the current state of the system. Besides the existing debt, the technical debt candidates should also be put on the list. Technical debt candidates are those tasks that are being considered for deferral. After identifying a technical debt item, the fields on the technical debt list such as date, responsible person, type, location and description can be filled out. The rest of the

information, i.e. principal and interest estimates, will be obtained through measuring technical debt, as described below.

3.1.2 Measuring Technical Debt

Different types of technical debt may require different forms of measures. For example, the number of known but not fixed defects is a measure of defect debt. The difference between expected code coverage of the test suites and their actual coverage can be used to measure testing debt. Since the ultimate goal of managing technical debt is to facilitate decision making, it is necessary that all types of technical debt have comparable measurement so that they can be easily monetized and are comparable to one another. Inspired by the technical debt metaphor, the proposed approach uses principal, interest amount and interest probability to measure all types of technical debt. To measure technical debt, each of these elements must be measured respectively. Hereafter these terms are used to refer to the metrics used to evaluate the technical debt. Initially, when a technical debt item is created, all three metrics (principal, interest probability, and interest amount) are assigned values of high, medium, or low. These coarse-grained estimates are sufficient for tracking the technical debt items and making preliminary decisions. More detailed estimates are not made until later, when finer-grained planning is needed and when more information is available upon which to base the estimates.

When more precise estimates are needed, estimation procedures are followed for each metric, based on the type of technical debt item. The details of these estimation procedures depend in part on the form and granularity of the historical data available,

but general estimation procedures are similar and have been defined for each type of technical debt item and each technical debt metric.

To estimate principal, i.e. the amount of effort required to complete a technical debt item task, historical effort data is used to achieve a more accurate estimation beyond the initial high/medium/low assessment. For example, if a debt item is a module that needs to be refactored, the average historical cost of modification of the module can be used as an estimate of the principal of the debt item, i.e., its future modification cost. If an organization has very rich, detailed data on many projects, the estimation would be more accurate and reliable, but even if the historical data is limited, a rough estimate can still be derived that is helpful in the technical debt decision-making process.

Interest probability addresses questions such as how likely it is that a defect will occur in the untested part, or that code containing a known error will be exercised, or that poorly documented code will have to be modified, etc. Interest probability is also estimated using historical usage, change, and defect data. For example, the probability that a particular module, which has not been tested sufficiently, contains latent defects can be estimated based on the past defect profile of that module. Since the probability varies with different time frames, a time element must be attached to the probability. For example, a module may have a 60% probability of being changed over the next year, but a much lower probability of being changed in the next month.

Thus, estimation of principal and interest probability is fairly straightforward.

However, estimation of interest amount is more complicated. Interest amount refers

to the amount of extra work that will be incurred as a result of not completing a technical debt item task, assuming that the item has an effect on future work. This quantity may be very hard to estimate with any certainty but, again, historical data can give sufficient insight for planning purposes. An example will help to illustrate the approach to estimating interest amount.

Suppose there is a technical debt item on the list that refers to a module, X, that needs refactoring. The interest amount in this case would quantify how much extra the next modification to X will cost if is not refactored first. Suppose also that analysis of historical data shows that the average cost of the last N modifications to X is the quantity C. We can assume that the extra effort to modify X (i.e. the interest amount) is proportional to C. The coefficient of C will be a weighting factor, W, based on the initial rough estimate (in terms of high, medium, or low) of the interest amount. For example, suppose a value of 0.1 is assigned to W if the initial interest amount estimate is “low”. This would imply that the “penalty” for not refactoring X before modifying it is about 10%. If the initial estimate is “medium,” 0.5 could be the weighting factor, which would imply a 50% penalty. The more refined estimate for interest amount, then, would be

$$\text{Interest Amount} = W * C \quad (1)$$

This formula implies that, if the unrefactored module X does in fact have an effect on future work, the magnitude of that effect will be a percentage of the average cost of modifying X, where the percentage depends on how severe the penalty is thought to

be. The severity of the penalty is captured by W , based on the initial coarse-grained estimate.

3.1.3 Decision Making

The goal of identifying and measuring technical debt is to facilitate decision making. There are two scenarios in which a technical debt list can be used to help management decide on various courses of action. The first is part of release planning, where a decision must be made as to whether, how many, and which technical debt items should be paid or incurred during the upcoming release. The second is ongoing monitoring of technical debt over time, independent of the release cycle. These two scenarios are described below.

Scenario 1 process:

Suppose significant work is planned for component X in the next release. Should some debt be paid down on component X at the same time? If so, how much and which items should be paid?

Assumptions: There is an up-to-date technical debt list that is sortable by component and has high, medium, and low values for principal and interest estimates for each item.

The release planning process is illustrated in Figure 10.

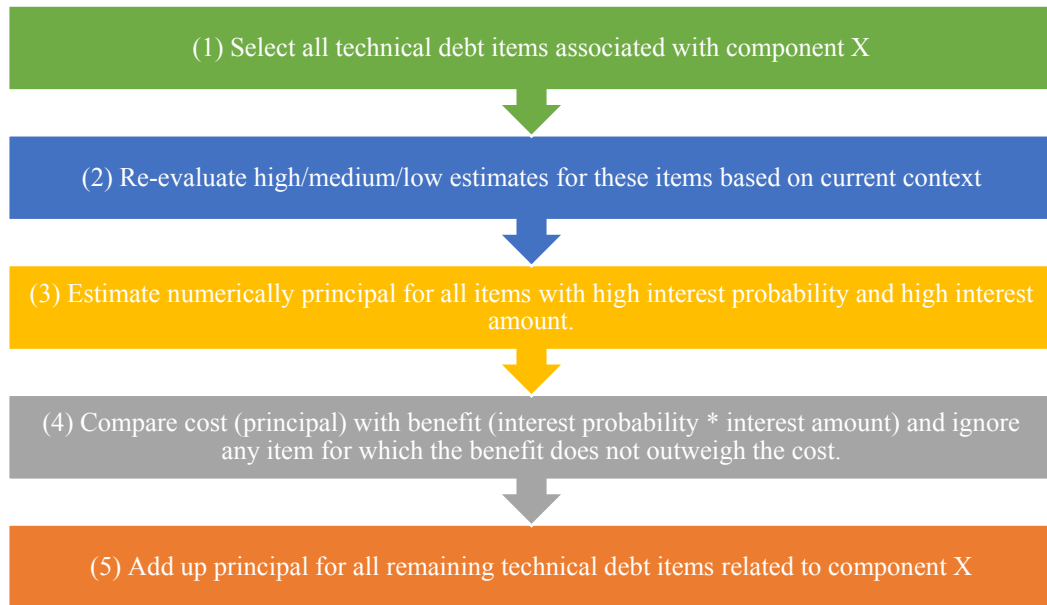


Figure 10. Release Planning Process

In step (1), a subset of the technical debt list is extracted, focusing on items associated with component X, as they are the most likely to have immediate impact and may be easier to pay off while doing other work. Because the cost and impact of these items might be different than originally thought (due to the fact that component X is planned to be modified anyway in this release), in step (2) we re-evaluate the original high/medium/low estimates for these items. For instance, some items may have a greatly reduced principal if they overlap with the planned work on component X in the release.

In step (3), we further restrict the subset of technical debt items we're considering by choosing high-impact items and doing numeric estimates only for those items. In this way, we reserve the time-consuming and difficult task of effort estimation for only those items most likely to be chosen for payoff in this release.

Step (4) operates on this reduced subset, and compares the cost and benefit for each item in that subset. The cost of choosing this item to be paid off is equal to the principal, i.e. the amount of effort required to eliminate this debt item. The benefit of paying off this item is the future extra work that would be avoided by eliminating the debt now. This is represented by the expected interest on this item, calculated as the product of interest amount and interest probability. The result of step (4) is an even more reduced subset of items that not only are related to work currently planned for the upcoming release, but that also are likely to have high impact if not repaid, and for which the benefits of debt payment outweigh the costs.

Finally, in step (5) we add up the estimated principal for the subset of items left after Step 4. This gives us a reasonable estimate of the cost of paying off the technical debt items that should be highest priority for paying off in this release. At this point, as part of the release planning process, we need to decide if this cost can be reasonably absorbed into the next release. If not, then we can use information about the interest (i.e. impact) related with these items to determine if the technical debt items should be prioritized over other tasks in the release (e.g. fixing bugs or new enhancements), and to justify the cost to management. In some cases, we might find that the cost of paying off the technical debt items can reasonably fit into the release schedule, and in fact there is room for further debt repayment. In this case, we repeat steps 3-5 with items with high interest probability and medium interest amount, and vice versa, then with medium for probability and interest, etc., until no more debt repayment can be absorbed by the release.

Scenario 2 process:

Is technical debt increasing or decreasing for a system or for a component? Is there enough debt to justify devoting resources (perhaps an entire release cycle) to paying it down?

Approach: Plot various aggregated measures over time and look at the shape of the curve to observe the trends. The aggregated measures include:

- (1) total number of technical debt items,
- (2) total number of high-principal items,
- (3) total number of high-interest (probability and amount) items,
- (4) weighted total principal, which is calculated by summing up over the entire list (set 3 points for high, 2 for medium, 1 for low) and
- (5) weighted total interest (add points for probability and amount).

3.1.4 Summary of Proposed Approach

The approach we propose for managing technical debt in a software project is based on the technical debt list, which contains the technical debt items that currently exist or may be incurred in the target software system. Figure 11 illustrates the major elements and steps of this approach. Using this approach, technical debt items are firstly identified from a variety of sources using various techniques. Some of these sources and techniques have been mentioned in section 3.1.2, while others will be outlined in more detail in section 3.2. Then the technical debt list can be constructed. The next step is to estimate the principal and interest of these technical debt items. Initial estimation uses an ordinal scale, i.e. high, medium and low. These initial

estimates are subjective and imprecise, but are sufficient to start with. More precise estimation can be achieved using historical effort and change information. Once information required by the technical debt list is complete, the technical debt items on the list can be monitored to facilitate decision making, that is, prioritizing the technical debt items according to their principal and interest and determining when and what technical debt items should be paid off.

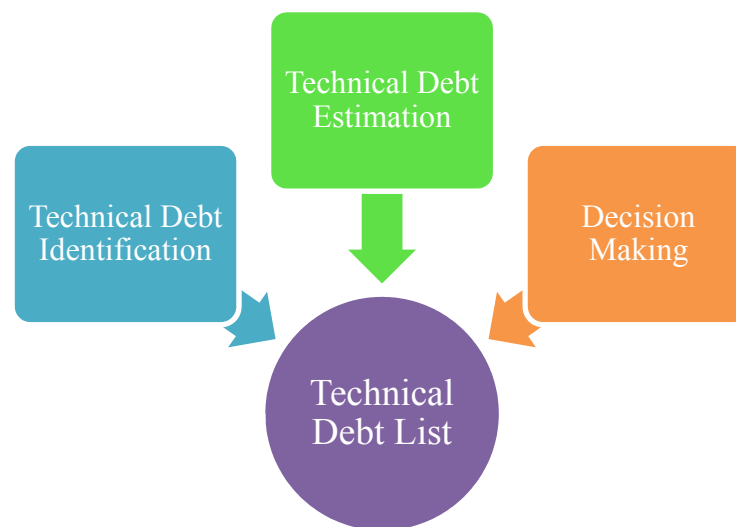


Figure 11. An Initial Technical Debt Management Framework

This approach to technical debt management relies heavily on the analysis of existing data from prior software development and maintenance efforts. The reliability and usefulness of the decision-making support that this approach provides increase with the quantity and quality of the data available. However, any amount of data will provide more support than is currently available. Thus, the approach is useful even with limited amounts of data. Expert opinion, or a combination of opinion and limited data, can be substituted for any of the inputs to the approach that call for historical data.

As mentioned at the beginning of section 3.1, this approach provides a general technical debt management framework. We started with this framework in the research described in the next section, with the expectation that it would be enhanced and refined by the research results from the ongoing studies. The results of the studies and the refinements to the proposed approach are described in chapters 4-7.

3.2 Research Design

The proposed technical debt management approach prescribes a general framework for technical debt management. Based on this framework, we conducted empirical studies of ongoing software maintenance projects to uncover the cost and benefit characteristics of explicit technical debt management. In particular, the studies were designed to address the following research questions:

RQ1: what factors contribute to the costs and benefits of measuring and monitoring technical debt?

RQ2: In what ways does technical debt information contribute to decision making?

The normal way of evaluating a new technique (i.e. RQ1) is to apply the technique, measure its cost, then observe and measure the benefit of the outcome. Comparing the cost and the benefit leads to a conclusion on the effectiveness of the technique.

However, in this case, the horizon for seeing the tangible benefits from technical debt management is so far in the future that this traditional approach is not feasible within the constraints of a dissertation. Therefore our approach separates the investigations of costs and benefits of the approach into different sets of studies. We explore the

benefits retrospectively by applying the technical debt management approach (through simulation) to past releases of several ongoing software projects. One advantage of this retrospective approach is that it does not rely on the actual project release cycle. Separately, we also explore the initial costs of this approach in case studies that follow software maintenance projects through several release cycles in real time.

Although our original research design involved a single comprehensive retrospective study, followed by a single case study with the same software project, this became infeasible due to the realities of empirical research with industry, including lack of data availability, cancelation of projects, and the narrow focus of some development projects. Therefore, rather than carrying out a comprehensive study in a single software project, we implemented studies in several projects and organizations, each of which focuses on part of the technical debt problem. The research design ensures that the retrospective part and the case study part can be run independently. The research goals are achieved by combining the results from all of these studies.

3.2.1 The Retrospective Studies

In this section, we first describe in the abstract the overall approach that we followed for the retrospective studies we conducted. Then, in subsections 3.2.1.3 and 3.2.1.4, we describe the subject studies used in the two retrospective studies.

As reiterated at the beginning of section 3.2, this research investigates the cost and benefit characteristics of implicit technical debt management. The objective of the retrospective study is to address the benefit side of the technical debt management

problem. The retrospective studies are based on in-depth analyses of historical effort, defect, testing, and change data over multiple recent releases. The data are used to retrospectively construct a technical debt list describing some of the debt existing in the system over a given period of time, and to simulate decisions about release planning as they would have been made if the technical debt list had been used. Meanwhile, data regarding how decisions on incurring technical debt were actually made and the factors considered when the decisions were made were also collected in the course of the studies. This type of data are used to address the second research question (RQ2). It also helps explain the reasons behind the costs and benefits revealed.

3.2.1.1 Procedure

The retrospective studies, in general, were implemented with the following steps.

- (1) Select a starting time point for the study, e.g., the beginning of the second release.
- (2) Review the defect, test, change and project planning data, run source code analysis tools, review the source code, and/or query developers to identify technical debt items that were present at the starting time, and choose a set of items to focus the study on.
- (3) Use the identified technical debt items to construct a technical debt list.
- (4) Simulate the decisions on release planning using the proposed technical debt management approach.

(5) Compare the simulated decisions with the actual decisions to determine the principal that would have been paid back and the interest that would have been avoided, if any, if the decisions had been made based on a technical debt list.

To make the study design more concrete and clearer, an example of design debt is used here to illustrate the simulation of decision making. Suppose System W has had three releases R1, R2, and R3. Figure 12 is a timeline showing the time points at which the events (including releases) happened. For example, the figure shows that the first release was completed at R1.

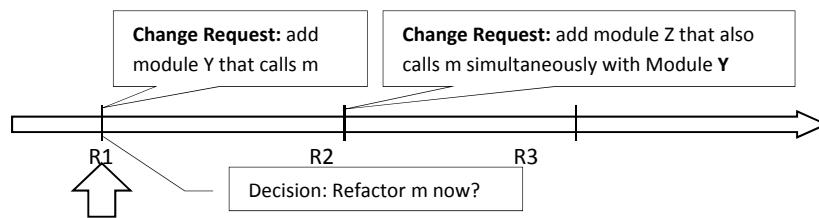


Figure 12. Timeline of System W's Events on Technical Debt Item A

Assume we have data about all the change requests in R2 and R3. First, we create a technical debt list corresponding to the time period after R1 is completed and before R2 is planned, beginning at the time where the vertical arrow in Figure 12 points to. Suppose two technical debt items associated with module X were identified in W. In the last release, Method m was added quickly and is thread-unsafe, which means it will cause a problem if m gets called simultaneously by two or more other modules. We identify this as a design debt item and name it item A. Meanwhile, we found that documentation of X has not been updated. This item is documentation debt and was named item B.

Assume there is a cost model for estimating modification effort and there are data about the usage and change history of System W for estimating the probability that a certain type of changes would happen in the future. The four decision alternatives for planning R2 are paying off neither A nor B, paying off A only, paying off B only, or paying off both A and B.

At the time R2 was being planned, there was a change request to be included in R2 that adds a Module Y to System W that calls and depends on Method m. The decision we are simulating is whether or not to refactor Method m in R2 before adding Module Y. Since it is thread-unsafe, we know that Method m will have to be refactored if and when another module is developed that needs to call m simultaneously with Module Y. If Method m is not refactored before adding Module Y, Module Y will need to be modified when Method m is refactored. Therefore the effort needed to modify Module Y would, in that case, be the extra cost of deferring the refactoring of Method m. In other words, the cost of modifying Module Y is the interest amount of technical debt item A. Whether the interest is realized depends on the probability that any new module calling Method m is added to System W in the future. Therefore, the interest probability of A can be estimated by evaluating how likely a new module calling Method m is added to the system. Assume we have estimated, using the cost model, that the cost for modifying Module Y is 20 person-days¹ and the cost for refactoring Method m is 30. Also assume we've estimated, using the historical data, that it is 50%

¹ There are other units such as person-months, or US dollars, that can be used. The unit selected for software development effort should be consistent with that used in historical data to ease comparison. Person-day will be used as the unit throughout the example. To be concise, it is omitted hereafter.

certain that another module calling Method m will be requested in the next release.

Then technical debt item A is measured as follows:

$$\text{Principal} = 30, \text{Interest Amount} = 20, \text{Interest Probability} = 50\%.$$

Using the proposed approach, we can also estimate the cost and benefit of paying off A in the current release (R2) as follows:

$$\text{Cost} = \text{Principal of A} = 30,$$

$$\text{Benefit} = \text{Interest Amount} * \text{Interest Probability} = 20 * 50\% = 10.$$

Since the benefit of paying off A in the current release does not outweigh its cost, we decide to defer refactoring Method m. This is the simulated decision.

Then we move to the planning of the next release (R3). At that time a new change request, to be implemented in R3, added another new module (Z) that calls Method m simultaneously with module Y, as shown in Figure 12. Therefore the total cost associated with the simulated decision includes the principal of technical debt item A (the cost of refactoring, which was finally completed in R3), plus the full interest amount of technical debt item A (the cost of changing module Y in R3 so that it is compatible with the changes to method m), i.e. $20 + 30 = 50$. This figure for cost will be used in comparing the simulated decision with the actual decision.

Following the same way, we can estimate technical debt item B, simulate the decision on whether to pay it off in the current release and calculate the associated total cost.

Table 3 shows the simulated decision on A and B and the corresponding total cost.

The last row, for technical debt item B, shows that the cost of paying off B in R2 is 45, while the benefit of doing so is 50, so the simulated decision (i.e. the decision that

would have been made if the technical debt list had been used) is to pay off item B. Also, the cost of paying it off (45) and deferring it (100) are also shown.

Technical Debt Item	Simulated Cost/Benefit and the Decision	Total Cost Incurred in R2 & R3	
		A	30/10 => defer
		Defer	50
B	45/50 => pay off	Pay Off	45
		Defer	100

Table 3. Cost and Benefit of the Simulated Decisions

Based on the total cost of decisions on A and B, the cost of each decision alternative is calculated, as shown in Table 4. The difference in total cost between the simulated decision (95) and the decision actually made (130) shows that a benefit of 35 person-hours would have been gained from using a technical debt list and our proposed approach to manage technical debt. Further, comparing the optimal decision (the decision that would have been made with 100% foresight) with that picked by the proposed technical debt management approach (i.e. the simulated decision) shows that another 20 hours could have been saved by choosing the optimal scenario. This sheds light on the effectiveness of the proposed technical debt management approach, thus contributing to refining the approach. In our retrospective studies, this analysis was repeated with data from multiple releases and for multiple technical debt items.

Decision Alternative	Total Cost Incurred in R2 & R3
Pay off none	150
Pay off A (Actual Decision)	130
Pay off B (Simulated Decision)	95
Pay off A & B (Optimal Decision)	75

Table 4. Total Cost of the Decision Alternatives

During the decision simulation, i.e. the retrospective analysis, as the quantity of technical debt items increases, the number of possible combinations of decision

alternatives becomes huge and easy to reach an unmanageable level. In that case, the approach presented in Section 3.1 could be used to limit the set of technical debt items to be considered. Other optimization approaches, such as linear programming (assuming independence between technical debt items), could also be used to choose the technical debt items.

3.2.1.2 Data Collection

Given the fact that technical debt has different forms, such as testing debt and defect debt, several different types of data were required for these studies. For example, to estimate the interest probability of testing debt (i.e., the probability that defects remain in the product that would have been removed if the testing activities had been carried out fully), historical data regarding fault detection ability of the test cases/test suites and defect density of modules was needed. Estimation of the interest probability associated with defect debt requires historical data about the number of known defects that are reported by customers after release. The cost of fixing a defect pre-release versus post-release is required to estimate the interest amount of defect debt. Documentation debt, which indicates that the documentation is incomplete or not updated when the corresponding modules are modified, may have a negative effect on the subsequent maintenance, such as increased time for fixing the problem and increased likelihood of making mistakes. In addition, updating documentation itself will take longer if the modification was made long before. Therefore, estimating documentation debt requires historical data regarding the time and effort for updating documentation.

These data are largely from six categories – change data, effort data, defect data, test data, size data and decision data. Ideally a comprehensive retrospective study should be carried out with all these types of data being available because it warrants a thorough investigation on the nature of the technical debt problem from multiple perspectives. However, it is often the case in reality that some categories of the data are missing or unavailable. Therefore, not all these types of data presented here are required to carry out a technical debt study, but data availability limits the scope of the study. This occurred in our retrospective studies as well. Not all data was available in each of the studies, so the scope was limited, but complementary between studies.

- **Change data** refer to information about changes to requirements, design, source code, test cases/reports/plans and documentation during the software lifecycle, including the proposed changes that were not implemented.
- **Effort data** refer to information about the effort invested in development and maintenance activities, preferably broken down by time period, system module, and change request.
- **Quality data** refers to information about the quality of the system such as defects.
- **Test data** refers to plans and reports describing testing that was applied to the system at different points in time.
- **Size data** refers to the size of the system for each release, in terms of LOC, classes, files, methods, etc., as well as other basic static metric data.

- **Decision data** refer to the information about how the decisions were made on whether to incur or pay off technical debt (even if it wasn't referred to as "technical debt" at the time).

Another type of information that is helpful has to do with business strategies. This information describes the possible benefit of delivering various software features to market at a particular time point. Ideally, the information would be in the form of a detailed business case (i.e. cost/benefit analysis) for each feature implemented or considered. Table 5 describes data required by the retrospective studies by categories.

Category	Description
Change Data	<p>Change description: what were the changes?</p> <p>Type of change: why the changes were made, e.g., bug fix, refactoring, or new requirement from customers?</p> <p>Involved artifacts: the artifacts (requirement specification, design documents, etc.) that are involved in this change</p> <p>Time</p> <p>Originating Release: the release in which the change was proposed.</p> <p>Planned Release: the release for which the change was planned.</p> <p>Actual Release: the release(s) in which the change was actually implemented.</p> <p>Owner: who was in charge of making the changes?</p> <p>Defect ID: Associated defect ID (if the type of change is bug fix)</p>
Effort Data	<p>Budget: estimated cost to implement new functionality or change requests</p> <p>Budget by software development phase</p> <p>Budget by release</p> <p>Budget by system module</p> <p>Budget by change request</p> <p>Budget by test suite/ case: estimated cost of creating and running a test suite/case</p> <p>Actual Cost</p> <p>Actual cost by software development phase</p> <p>Actual cost by release</p> <p>Actual cost by system module</p> <p>Actual cost by change request</p> <p>Actual cost by test suite/ case: actual cost of creating and running a test suite/case</p> <p>Release Dates</p> <p>Planned start date of each release</p> <p>Planned end date of each release</p>

	<p>Actual start date of each release</p> <p>Actual end date of each release</p>
Quality Data	<p>Defect ID: Used to associate the defect with the corresponding change request for fixing it</p> <p>Type of Defect: the category that a defect belongs to in terms of the causes of the defect, e.g., Logic defect, computation defect, data handling defect.</p> <p>Defect detected phase: when was a defect detected, e.g., in inspection, unit test, or system test?</p> <p>Defect detected/fix release: Related release in which the defects were detected and fixed.</p> <p>Severity of defect</p>
Test Data	<p>Test suite/case name</p> <p>Applied modules: the modules that the test suite/case was exercised on</p> <p>Planned code coverage (or other criteria indicating the fault detection ability of the test suite/case)</p> <p>Actual code coverage</p> <p>Test phase: the phase in which this test suite/case belongs to, e.g., unit test, integration test and system test.</p> <p>Time</p> <p>Planned Release: the release for which the test suite/case would be executed</p> <p>Actual Release: the release in which the test suite/case was actually executed</p> <p>Test result: Pass/fail</p>
Size Data	<p>System size by module, function point, LOC or other size metrics</p> <p>The size of each module in terms of LOC or other size metrics</p> <p>Other metric data used for detecting code smells (E.g., WMC refers to weighted method count, which measures the size of a class).</p>
Decision Data	<p>Contributing Factors: what information was taken into consideration when making the decision?</p> <p>Lessons learned: reflection on the problems/loss caused by the wrong decisions.</p>

Table 5. Data Descriptions

Except the last category, the data described above was used for the simulation of decision making. Most of the data were derived from software artifacts, project documentation or the output of source code analysis tools. Decision data were collected through interviews with the project personnel or group discussions if they were not directly available from the project planning documentation. We collected data about unexpected debt repayment, crises caused by unpaid debt and decisions

that had been made based upon an implicit understanding of technical debt. Project personnel were also involved to help interpret the data and add historical context. They tried using the technical debt list and gave feedback for tailoring the proposed approach.

Simulation of the decision making in each study yielded a set of technical debt items, their characteristics and the associated costs and benefits. From the results we have started to build a technical debt theory using these quantitative data and the qualitative decision data gathered from project personnel. The qualitative data help construct the context where the decisions were made so that the factors contributing to costs and benefits of technical debt can be identified, while the quantitative data are used to determine the degrees to which different factors affect the costs and benefits of technical debt. The qualitative data can also explain why the decisions are affected by the factors and how the factors are related with one another. The resulting theory constitutes a detailed descriptive account of the costs and benefits related to technical debt management.

3.2.1.3 The SMB Study

The project we selected for the first retrospective study was a software application from a multi-national company that provides mobile communication products and solutions. The application consists of the development and evolution of a software application for mobile platforms, used as a client solution for Microsoft Exchange Server. The system had over 5 years of evolution and involved 20 software engineers. In this study only one technical debt item was identified and tracked, but it was a big

technical debt item and had serious impact on the project. The debt item was to delay upgrading the communication protocol used to connect the application to MS Exchange Server. The object of this study consists of a sequence of decisions. In this study we started the decision simulation at the point of time when the project needed to decide the first time whether or not to delay the upgrading. The decision simulation ended with the debt being paid off using our proposed approach. The data we collected in this study include the changes in the project, the size of the application, the decisions made during the selected period of time and the effort of changing the communication protocol. In addition, effort required for paying off the debt at different decision points were also estimated by the project personnel. Since only one technical debt item was tracked in this study, the prioritization mechanism in our proposed technical approach was not used. This study focused on the impact of technical debt on the project and the benefit that could be gained through explicit technical debt management.

3.2.1.4 The Hadoop Study

Compared to the SMB project used in the first retrospective study, the subject project of this study has a completely different profile. The project, named Hadoop, was from the open-source software community. This choice was made partially due to the lack of available projects from our industrial partners, but we also expect that more diversified subjects of study may provide us with new perspectives on the technical debt problem and improve the generalizability of the conclusions drawn from the studies for this dissertation research. Hadoop is a software library for the distributed processing of large data sets across clusters of computers using single programming

models. It's based on Hadoop Distributed File system (HDFS) to provide high throughput data access service. It uses a framework, Hadoop YARN, for job scheduling and cluster resource management, and MapReduce, a YARN-based system, for parallel processing of large data sets. Hadoop also includes "COMMON", a utility set to support other Hadoop modules for file system access. As mentioned at the beginning of this chapter, we selected this project because our study used the analysis of Hadoop performed in the previous study as a starting point to select the technical debt instances for our decision simulation. Four classes were finally selected for the decision simulation. The previous case study analyzed the Java core packages of the system (java/org.apache.hadoop) from release 0.2.0 to release 0.14.0 (13 releases in total). There were 126 Java classes with 10.5K NCSS (non-commented source statements) in the release 0.2.0 of the system. It had 373 Java classes with 37K NCSS by release 0.14.0. The decision simulation began from the first release used for this study, or the release in which the classes, i.e., the containers of the selected technical debt instances, were created, and ended in the last release used for this study unless the class has a shorter life. Due to the limited data availability, code churn was used as the proxy of change effort. Finally the simulated decisions were compared with their actual counterparts to derive the possible benefits that might have been gained from explicit technical debt management. The source code of the selected classes were also analyzed to explain the reason behind the changes and the decisions on technical debt repayment. This study focused on characterizing the benefits of explicit technical debt management in the context of a software project with a lot of

minor technical debt issues rather than a big technical debt monolith, such as the one used in the SMB study.

3.2.2 The Case Studies

While the retrospective studies have provided evidence of a benefit from applying the technical debt management approach, they did not shed any light on the costs of the approach. This insight comes from the case studies, which focus on the future releases of the selected projects. In this section, we first describe in the abstract the overall approach that we followed for the case studies we conducted. Then, in subsections 3.2.2.3 and 3.2.2.4, we describe the particulars of each case study as it was implemented

3.2.2.1 Procedure

Before implementation of the technical debt management approach in the project, we collected baseline data, in terms of effort, cost, and productivity of the project, to be used for comparison. We also implemented a simple tool to aid in managing the list and to collect data on its use. The list was kept under version control, along with other project documents. The process of implementation was closely monitored, including all issues raised and their resolutions.

The case studies were conducted with the following steps.

- (1) Train the project personnel on how to manage technical debt using the proposed approach and how to document and report information required by this study.

- (2) In the first release cycle, during which the projects were tracking technical debt for the first time, detailed qualitative and quantitative data was collected concerning the amount of time project personnel spent using the technical debt list, the kinds of problems and questions that came up, and the usefulness of the approach from the point of view of the project personnel.
- (3) After the first release, interviews were conducted with the project, aimed at characterizing in more detail the amount of work required to carry out the approach, the confidence that personnel have in the eventual benefits, and any benefits already observed.
- (4) Data were then collected over at least two successive releases. The data include documentation of decisions made concerning technical debt, the costs of using the technical debt list, its contribution to decision-making, and the rate at which technical debt items are added and removed from the list.

3.2.2.2 Data Collection

Data required for the case study include those types of data described for the retrospective study, as shown in Table 5, but they were collected during the implementation of the new releases. The baseline data about the cost, effort and productivity of the project were collected in the first place, before the proposed approach was applied. Baseline data were collected through project documentation review and interview with project personnel. Beside those data, the case study requires the data about the cost of using the proposed technical debt management approach, which is the core category of the data. This includes costs of meetings, effort spent managing the technical debt list, and time spent gathering data to feed

into technical debt management. The results from these case studies were compared with the benefit data collected in the retrospective studies to characterize the net benefit of the technical debt management approach.

3.2.2.3 The Tranship Case Study

The Tranship project was a software application from a Brazilian software company that provides enterprise-level software development, consulting and training services. The project consists of the development and maintenance of a water vessel management application. It included 25K non-commented lines of code in 245 classes (of which 80 were domain classes). The application was written in Java and based on the MVC framework. The application included administration, resource allocation, ship management, and billing and operation modules, which were mapped to 95 use cases. The average development effort was 37 person-hours per use case. The project was developed with the following infrastructure: Eclipse IDE, Subversion (for code version control), and Trac (a bug and workflow tracking system). The project began in late 2011. There were 1 project manager, 1 technical leader and 7 developers working on the project. The project followed a Scrum-like development process. Each sprint lasted about one week. The case study began with baseline data collection in March 2012. By the middle of November 2012 the project team finished tracking technical debt items for this study. The case study ended in the middle of December, 2012 with the follow-up interview being completed. After the first release during which the technical debt was being tracked for the first time, the research team communicated with the project team to address questions and get feedback in both directions. This step continued until all the questions were addressed and the project

team was clear and comfortable with the proposed approach. In this study we collected baseline data of the project, costs of using the technical debt management approach, changes in the technical debt list, decisions on when and what technical debt items should be paid off, benefits of using the technical debt management approach, and problems and suggestions for the proposed technical debt management approach. These data were collected from either the project documentation or the project personnel.

3.2.2.4 The EducationHub Study

Compared to the project used in the first case study, the subject project of this study has larger size and longer development cycle. The purpose of this software application was to provide an education platform for teachers and students in a school to share information. The platform integrates with other applications such as Moodle, EverNote and Dropbox. It provides call list, note list, messaging service, file sharing and access to libraries. Originally the project planned to carry out 17 sprints. When we started collecting data, the project consisted of 45 user stories with 18 features, implemented in 178,195 LOC (Lines of Code), but it grew over time. The entire application was written in Java.

The subject project was developed by a company (Developer) that developed software systems for a multinational organization of the mobile communication market (Client). Developer and Client were located in different places in the same country and most interactions between the business team on the Client side and the development team occurred over distance, using conferencing tools. At the time of

our study, the Developer had approximately 70 employees, with 60 working on software development. The organization of the software teams was flexible to fulfill the demands of the Client and several projects were conducted simultaneously. Two senior (over 15 years of experience) project managers supervised each project. The team developing the subject project was managed by the most senior project manager, who had over 20 years of experience in managing software projects in industry. Interaction with the Client team was conducted by the project manager and project leader, who addressed issues related to feature and release planning, together with the client. The project followed a Scrum-like development process with each sprint lasting for one month. The technical debt items tracked in this study were identified by different methods, e.g. source code analysis, survey of the developers, etc. Due to a variety of factors, the technical debt management approach we proposed was loosely followed, which created threats to the validity of the study, but also provided us with the opportunity to investigate other aspects of the technical debt management problem, in particular, the obstacles of applying technical debt management in practice and the reasons behind it. Therefore, the focus of this study has slightly changed to the actual technical debt management process followed and the costs incurred by all activities related to technical debt management. We believe revealing the causes of the process deviation would contribute to the improvement of technical debt management approaches and thus eventually benefits software projects for better decision making.

3.3 Summary

This research aims at revealing the cost and benefit characteristics of explicitly measuring and monitoring technical debt, and how technical debt information contributes to decision making. Based on the findings of this research, the effectiveness of explicit technical debt management could be evaluated and the management approach we proposed could be refined and improved to facilitate software maintenance decision making.

Chapter 3 is composed of two main parts – the proposed technical debt management approach and the research design for uncovering the cost and benefit characteristics of explicit technical debt management using the approach. The proposed approach serves as a starting point and framework within which to conduct the research and refine the approach. The approach includes methods for identifying, measuring and monitoring technical debt, a process for managing technical debt and two concrete scenarios in which technical debt information can be applied to decision making in software maintenance.

Using the proposed technical debt management approach, we investigated software maintenance projects to gain insights into the cost and benefit characteristics of explicit technical debt management. Meanwhile the proposed approach was also evaluated in the process. First, two retrospective studies were conducted to determine the benefits of explicitly managing technical debt using the proposed approach. The studies were based on analysis of effort, change and decision data over several recent releases of a project. Second, understanding of the costs of tracking technical debt

came from two case studies in which the proposed technical debt management approach was applied on ongoing releases of the project.

The cases used in these studies vary widely in terms of project size and age, type of technical debt and development cycle. The strategy for selecting these cases meets the “maximum variation” criterion defined for case study research [151]. It increases the generalization power of the findings from these studies. In addition, investigating diversified cases, which may be different in project cost pattern, management style and development environment, yields more insights into the nature of technical debt and thus deepens our understanding on the essence of the technical debt management problem.

The findings from the case studies and the retrospective studies, together, are used to evaluate the cost effectiveness of explicit technical debt management, as well as refinements to the proposed approach for managing technical debt.

Chapter 4: The SMB Study

As mentioned in the first chapter, the objective of this research is to characterize the costs and benefits of explicit technical debt management and how the technical debt information contributes to decision making. In this study we explored the effect of technical debt by using historical data to track a single delayed maintenance task in a real software project throughout its lifecycle and simulating how explicit technical debt management might have benefited the project. This study was a retrospective study, aimed to characterize the benefits of explicit technical debt management. Results and findings of this study have been published at a conference on software engineering [152].

4.1 Overall Approach

The research questions of this study are: (1) what is the impact of technical debt on a software project? (2) whether and how much benefit can be gained through explicit technical debt management? As a retrospective type of study, this study requires a software project that has contained technical debt and has had release cycles in the past. The technical debt in the project should be identifiable as individual instances and they are measureable, that is, the principal and interest of the technical debt instances in the subject project can be estimated. The development cost or effort information of the project, especially of the modules containing technical debt were documented or can be assessed with reasonable accuracy. Moreover, it should be available that the decision information regarding the evolution of the technical debt instance, such as when and why a technical debt instance was incurred, decided to

carry over or paid off. All of the information together constructs an environment required to run the decision simulation – the core approach of the study design. The subject project was provided by our industry collaborator based on the case selection requirements (e.g. a software project with past releases and containing technical debt instances whose value are measurable, availability of the decision information regarding the evolution of the technical debt, etc.) we made for them. This study began with identifying a particular maintenance task that was delayed (i.e. a technical debt item), and then an investigation on the reasons that the task was delayed and the context surrounding the decisions made, both to delay it and later to complete it. We also estimate the effort that would have been needed to fulfill the task when it first arose, the effort required to complete the task when it was actually completed, and the impact of the delay on other tasks that were made more difficult, or that had to be re-done, because of the delay. Then we simulated the decision making using the proposed technical debt management approach described in Chapter 3. In other words, we re-created the decision to delay (or not) the maintenance task using the proposed technical debt management approach, to see if a different decision would have been made. Finally we compare the impact of the actual decision with the hypothesized impact of the simulated decision to determine the benefits that could have been gained from explicit technical debt measurement and management.

4.2 The Subject Project and Technical Debt Item

The selected project was the Samsung Mobile Business (SMB) application, which consists of the development and evolution of a software application for mobile

platforms, used as a client solution for Microsoft Exchange Server, as described in Section 3.2.1.3.

The technical debt item studied is related to a sequence of decisions about the communication protocol used to connect the application to MS Exchange Server. Figure 13 shows this decision evolution. The application implementation began in August 2006 and the communication protocol used was WebDAV with support for MS Exchange 2003. Two months later, a member of the development team, an expert in MS Exchange, warned about the upcoming launch of MS Exchange 2007, and that WebDAV would not support it. Shortly after, in November 2006, a decision had to be made between keeping WebDAV and re-implementing the communication layer using ActiveSync, a technology that would support MS Exchange 2007. Given the time-to-market constraints, the decision (D1) was to keep WebDAV for the following reasons: the application was already developed for WebDAV, the 2007 version had not yet been released, and MS Exchange 2003 was active on the market, creating a real potential for the immediate use of SMB supporting that version.

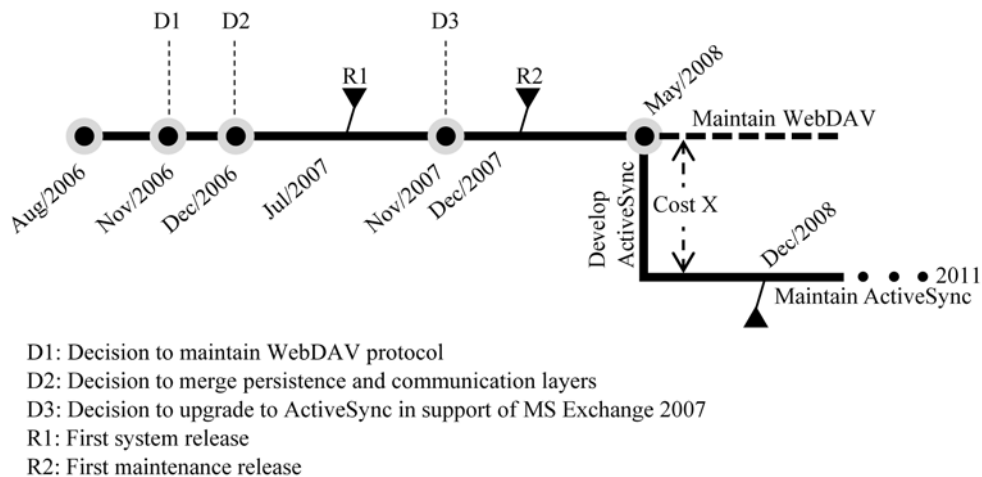


Figure 13. Timeline of the SMB Evolution

A month after this decision and after application performance analysis, it was decided to couple the communication and persistence layers of the system's architecture (D2) to improve system performance. At the time of this decision it was known that: (1) the probability of the release of MS Exchange 2007 was high, (2) the protocol used (WebDAV) did not support this new version, (3) coupling the architecture layers would increase performance, and (4) coupling these two architectural layers would imply greater rework to change the communication protocol in the future. It was decided that the performance considerations were most pressing at the moment, and hence the decision was made to couple the communication and persistence layers of the architecture without upgrading to the ActiveSync communication protocol.

Seven months later, in July 2007, the product was deployed to the first customer (R1). During the second half of 2007, the SMB customers started to migrate their email servers to MS Exchange 2007, breaking compatibility with the implemented version of the SMB communication protocol. In November 2007, it was decided to change the communication protocol to replace WebDAV (D3). At that time, upgrading the communication protocol to ActiveSync required changes not only in the communication layer, but also in the persistence layer due to the coupling of the two layers previously discussed. The changes in the persistence layers could have been avoided if the upgrading of the communication protocol had taken place before merging of the two layers. We use Cost **X** to denote the total effort of upgrading to ActiveSync, as shown in Figure 13. The upgrading work was completed by May, 2008. Seven months later (Dec of 2008) a new maintenance release with full support of MS Exchange 2007 was delivered to the customers.

It is our contention that if decision D1 had been to change the communication protocol, the cost of supporting MS Exchange would have been smaller even in the context of the decision D2 to couple the two layers of the architecture to achieve better performance. The results we present below support this supposition. A core part of our technical debt definition (section 1.1) is that technical debt has a short-term benefit and a long-term cost. In this case, the short-term benefit is that maintaining the old communication protocol can get the product to market quickly, while the long-term cost is that it will take more effort to upgrade the communication protocol in the future. Therefore, we consider that delaying upgrading the WebDAV protocol is technical debt. According to the definition presented at the beginning of Section 3.1, delaying upgrading the communication protocol constitutes a technical debt item, which we denote as T1. Our analysis below simulates what might have happened in this project if the technical debt had been managed explicitly.

4.3 Measurement

We use principal, interest amount and interest probability to measure the technical debt item. The impact is measured by the effort saved (at the points where the debt is not paid off) or required to complete the task (at the point where it is paid off) against the loss caused by technical debt. In this study the principal of T1 (denoted as P thereafter) is the effort to switch to ActiveSync at the time of D1. Since the actual decision was NOT to switch to ActiveSync, the effort to switch at the time of D1 has to be estimated using the project's typical effort estimation approach – expert estimation – and the information that was available at the time of D1.

The interest amount of T1 (denoted as IA hereafter) is the impact that T1 has on the future maintenance of SMB, that is, the extra effort required to switch to ActiveSync because that task was delayed (to after D2). The actual interest amount (IA_{D3}) is estimated using the following equation:

$$IA_{D3} = X - P \quad (1),$$

where X is the total actual effort of upgrading to ActiveSync at the time of D3 (Cost X in Figure 13) and P is the effort that would have been required to change to ActiveSync at the time of D1, which is the principal of T1, as defined above.

In addition, to simulate the decision making, the interest amount also needs to be estimated at D1 and D2. The interest amount at D1 (IA_{D1}) represents the extra work required in the future due to the decision of keeping the WebDAV communication protocol rather than upgrading to ActiveSync. For example, the source code implementing the communication functionality in conformance to the WebDAV protocol would have to be rewritten if the communication protocol is changed to ActiveSync. Therefore, the effort to rewrite the source code in the example is part of the extra work, which will be required when the communication protocol is upgraded, because of the decision at D1 to carry T1. Interest estimation should be based only on the information available and the future situation of the project that the decision maker can foresee, at the point of time where the decision is made (simulated). For example, estimating IA_{D1} should take into consideration the rework cost of changing the communication layer for ActiveSync, the new communication protocol, but not the effort in modifying the persistence layer to conform to ActiveSync because at the time of D1, it remained unlikely for the project team that the two layers would be

merged. By contrast, by the time of D2, it had been determined to couple the two layers to address the performance issue. Therefore, estimating interest amount at D2 (IA_{D2}) should consider the rework effort on the modules affected by the decision to couple the communication and persistence layers, which then later had to be changed again to implement the new communication protocol.

Similar to interest amount estimation, the interest probability of T1 (IP) needs to be estimated at D1 (IP_{D1}) and re-estimated at D2 (IP_{D2}) because the probability may vary in different time frames and the simulation depends on using the information available at each simulated point of time. In this case the interest probability is affected by the probability that coupling the communication layers and persistence layers is required (PC) and the probability that SMB needs to switch to ActiveSync (PM). Since PM and PC are independent of each other,

$$IP = PC \times PM \quad (2).$$

4.4 Data Collection

In this study we collected data regarding the changes in SMB and the decisions made during the selected period of time. These change data and decision data were mainly collected with the help from one of our company contacts. The collection process of these data began with sending our data requirements to the company contact. Then the contact communicated with us if she had any questions about our requirements. Once the contact was clear about the data requirements, she communicated with the project personnel to gather the data, in the form of formal project documentation, or transcripts of informal interviews with the project personnel. Because the contact was

very familiar with the project, she could easily find the right people who could supply the needed data. Thus it saved us a great amount of time and effort in collecting these data. In this process she also served as a translator between us and the project team as our communication with the contact was all in English while the communication between the contact and project team was in Portuguese. The data collected by this means were also translated into English by the contact. Then the contact sent the data to us for review. When we had questions about the data, we communicated with the company contact and she followed up with the project team with our questions. This process ended when all the data requirements were satisfied. These change and decision data explain what happened to the project, why and how it happened, as described in Section 4.2.

We also collected effort data on the release in which the communication protocol was changed to ActiveSync (cost X). The effort data were collected from the project chronograms of the related release cycle. Table 6 shows the break-down of cost X, from which IA_{D3} is derived using formula (1). In this study the effort is measured in staff-hours.

Task	Effort (staff-hour)
Persistence	514
Analysis	292
Rewriting	172
Improvements	238
Stabilization	50
Communication	922
Rewriting	922
TOTAL (Cost X)	1436

Table 6. Effort to Change to ActiveSync (cost X) at D3

In addition, values for P , IA_{D1} , IA_{D2} , IP_{D1} and IP_{D2} were estimated by the project personnel based on impact analysis of the potential changes that could have been foreseen. Table 7 shows the break-down of the estimation for the principal (P). Table 8 shows the estimates of the interest amount at $D1$ and $D2$, and the estimates of PC and PM , which are used to calculate IP_{D1} and IP_{D2} according to formula (2). For example, at the time of $D1$, the project personnel would have expected 465 more staff-hours of effort to be expended to upgrade the communication protocol to ActiveSync in the future, assuming the decision at $D1$ was to maintain WebDAV, in comparison with upgrading to ActiveSync at present. In other words, IA_{D1} is equal to 465 staff-hours. Meanwhile, they also would have expected, based on the information available at $D1$, that the probability of coupling the persistence and communication layers, i.e. PC , was very low (estimated as 20%) and the probability of the need to upgrade to ActiveSync, i.e. PM , was medium-high (estimated as 70%). According to formula (2), IA_{D1} , the probability of spending 465 more staff-hours to upgrade to ActiveSync, is equal to 14%, the product of PC and PM at $D1$.

Task	Effort (staff-hour)
Folder Synchronization	80
Message Synchronization	149
Attachment Download	159
Download Body	119
TOTAL	507

Table 7. Effort Estimate to Change to ActiveSync at $D1$ (P)

Estimates	Point of Time	
	<i>D1</i>	<i>D2</i>
IA (staff-hour)	465	836
PC	20%	100%
PM	70%	70%

Table 8. Interest Amount and Change Probability Estimates

4.5 Data Analysis

With the data collected from the project documentation and personnel, we constructed a technical debt list (defined in our proposed technical debt management approach, in Section 3.1), which contains the item T1. From the time of D1 we started to track T1 retrospectively by estimating P, IA_{D1} and IP_{D1} . Then we simulated the decision about incurring T1 through cost-benefit analysis. That is, we simulated how the decision would have been made, at times D1, D2, and D3, if the project personnel had been using the proposed technical debt management approach described in Chapter 3. In Figure 14 there are three columns, each of which contains the actual decision, the estimated principal, interest amount and interest probability, the cost and benefit associated with the decision, and the simulated decision, at each decision point. The three columns are shown along a time line to illustrate the process of decision simulation. The benefit of incurring T1 at D1 is the principal of T1, i.e. P (507), while the cost is the interest of T1, which is the product of the interest amount and interest probability, IA_{D1} (465) and IP_{D1} (14%). Since the benefit of incurring T1 outweighed the cost, the simulated decision would have been to keep WebDAV at D1, same as the actual decision.

At D2, a decision was made by the project team to couple the communication layer and persistence layer, while still keeping T1 (i.e. not switching to the ActiveSync protocol). Here we re-estimated the interest amount and interest probability as the situation, including the information availability, had changed since D1. Following the same procedure, we simulated the decision at D2. Since the cost of keeping T1 outweighed the benefit, the simulated decision would have been to pay off T1 at D2. However, the actual decision was to not pay off T1 at time D2. Thus, we went to time D3 and used the actual total cost X to determine the impact that the actual decision not to pay off T1 had on the project, which is IA_{D3} . The difference between the net cost of the simulated decision ($P - 0$) and the actual decision (X) is the benefit that could have been gained through the explicit technical debt management, which is equal to IA_{D3} (=929) according to equation (1).

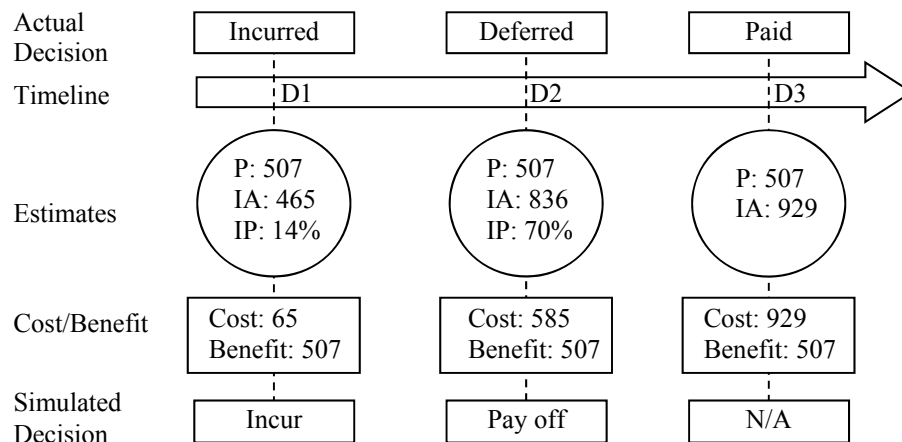


Figure 14. Decision Simulation Process

4.6 Results

The above analysis makes it clear that delaying the upgrade of the communication protocol had significant impact on the cost of the project. By tracking this technical debt item, we can see that it was incurred for the sake of time constraints in the

situation that future demand of the support for MS Exchange 2007 and changes imposed on the system were not clear. This is a typical scenario in which technical debt is incurred. Figure 15 shows the cost and benefit associated with this technical debt item at different decision points.

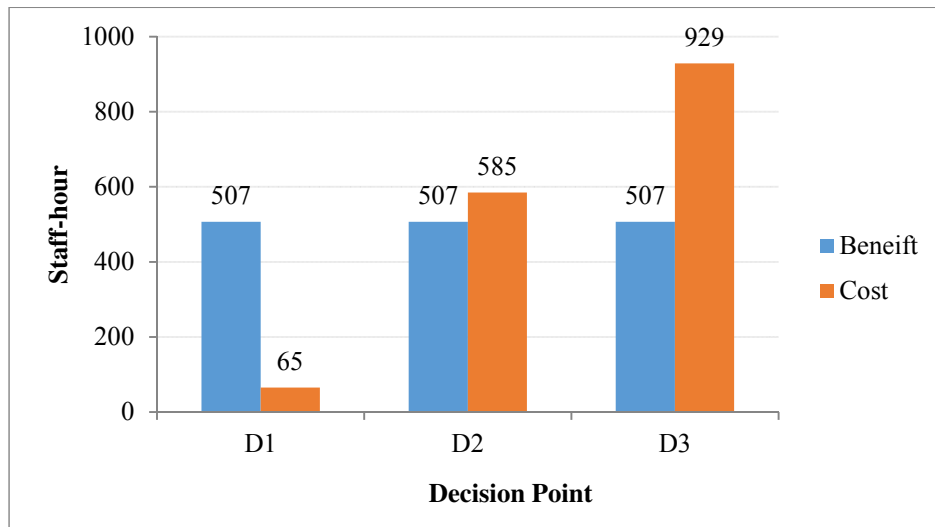


Figure 15. Cost/benefit of T1 at Decision Points

At D1 the interest amount and interest probability were estimated low based on the information available at that time, which translates to low cost of incurring T1 (65) in the language of cost/benefit analysis. Thus it was not a bad decision to keep WebDAV, the old communication protocol, to save time for the project. However, the situation was changing quickly and the project faced a change soon due to the application's unsatisfactory performance. Meanwhile the release of MS Exchange 2007 indicated that changing to a compatible communication protocol would be demanded soon. At that time (D2) both the potential penalty of keeping WebDAV and the probability that the penalty would be incurred were high, which were reflected in the high value of the interest amount and interest probability. The foreseen cost to keep T1 (585) already exceeded the benefit (507), that is, the

situation didn't allow the old communication protocol to be used any longer.

Unfortunately the decision was to couple the communication layer and the persistence layer without changing the communication protocol. When it finally became required that Exchange 2007 be supported, high rework cost was inevitable. The net cost of carrying the technical debt (929) was almost two times the net cost of switching to the new communication protocol before merging the two layers (507). By contrast, if the decision had been made in the same way as the simulated decision, the extra cost would have been avoided.

4.7 Conclusion

This is a case of a type of technical debt that often occurs in practice turning into a big problem for the project. It almost tripled the cost of upgrading to ActiveSync, which was ironically the cost that the technical debt was incurred to save (or at least delay). By tracking the technical debt and measuring the impact, this study provides evidence of the effect of technical debt on software projects and concrete numbers about how serious the problem is, at least in this typical case. Furthermore, the study demonstrates how a decision made without careful analysis could aggravate the negative effect of technical debt, while attention and explicit management of technical debt could make a difference. Therefore we can argue that the project could have gained significant benefit (in the form of cost saving) if a detailed cost-benefit analysis, as we did in this study, had been presented to the software manager before they made their decisions.

In the studied case, delaying the communication protocol change allowed the application to be released on time. Thus the real benefit of incurring the technical debt lies in the value of time to market, which was not considered in our approach. Instead, the principal of the technical debt was used as a proxy of the benefit, which may affect the decisions made in the course. However, this doesn't negate the value of explicit technical debt management. The real benefit of incurring technical debt, i.e. the value of earlier time to market, can be derived through a business case analysis. It's in such occasions that people from the business sector and the technical sector should work together to determine a reasonable schedule for the project, where technical debt is used as an effective device to facilitate communication between the two groups of people and analysis of the cost and benefit of different marketing strategies. Regardless of their decisions, managing technical debt in an explicit manner such as the cost-benefit analysis approach we proposed, can help software managers make informed decisions, thus avoiding surprise in the later stages of the software project and regrets that can only be seen in hindsight.

Chapter 5: The Hadoop Study

As mentioned in Chapter 3, the goals of this dissertation research are to identify the characteristics of the costs and benefits of incurring technical debt, the costs and benefits of explicit technical debt management and how technical debt information contributes to decision making. To achieve these goals, we have designed two types of studies, each of which focuses on one side, i.e. benefit or cost, of technical debt management. In the retrospective studies, the potential benefits of explicit technical debt management were uncovered and measured by comparing the actual decisions with the simulated ones in the course of release planning. The Hadoop study, described in this chapter, is one of the retrospective studies we carried out. Rather than using a software project provided by our industrial collaborators, we chose a project from the open-source software community. Our decision was partially due to the lack of available projects from our industrial partners, but we also expect that more diversified subjects of study can give us new perspectives on the technical debt problem and improve the generalizability of the overall work. In this study we examined whether the presence of technical debt is associated with future maintenance cost and whether the potential saving of those costs by paying down technical debt at various points of time as suggested by our approach can justify the costs of the explicit technical debt management strategy.

This study used the analysis results from Zazworka et al.'s work regarding the commonalities and differences among four technical debt identification approaches and their relationship to indicators of technical debt interest [153]. In their study, four

technical debt identification approaches were applied to multiple versions of Hadoop, an open-source software project, to detect different types of technical debt indicators such as modularity violations, grime and code smells. Meanwhile the interest indicators, including change proneness and defect proneness, were calculated based on the information extracted from the source code and bug repository. Then the relationship between the technical debt indicators and the interest indicators were quantified to reveal whether different technical debt indicators point to different maintenance problems (represented by the interest indicators) and thus lead to the answer of whether these approaches have overlap in terms of technical debt identification. According to the results of their study, modularity violation and dispersed coupling are the top two technical debt indicators that show a significant positive relation with change proneness. For example, a class with modularity violation has higher change probability than a class without the violation. Since change proneness may indicate maintenance problems such as high change effort, classes with technical debt indicators may be more likely to be considered as candidates for refactoring. These previous findings helped us identify classes from Hadoop whose change proneness appeared to be affected by the technical debt in them. Then we used these classes to simulate the decision of whether and when to refactor the classes to reveal what difference could have been made if our technical debt management approach had been applied.

5.1 Process

This study began with identifying classes whose change proneness fluctuated with some technical debt indicator, which likely is evidence that the indicator points to real

technical debt in those classes. After the classes were identified, we quantified the technical debt in them, that is, we estimated principal, interest amount and interest probability for the technical debt in each version of the classes. Then we simulated the decisions by applying our technical debt management approach to the past versions of these classes, to determine when the technical debt would have been paid off. We also determined when the technical debt was actually paid off (intentionally or not) by observing the point at which the technical debt indicator was no longer present in the class. Finally we compared the simulated decisions with the actual ones to evaluate the effectiveness of our approach in terms of the effort that could have been saved.

5.2 The Subject Project and Classes

The selected project was Apache Hadoop (<http://hadoop.apache.org>), as described in Section 3.2.1.4. In this study we used the results yielded by Zazworka et al. [153] as a starting point. In particular, the change likelihood of the classes and the number of technical debt instances in them from Zazworka et al.'s study were used. The change likelihood of a class with a particular version was defined as the number of repository changes affecting the class in the version divided by the total number of changes in the repository included in that version.

The criteria we used to identify the classes for decision simulation are as follows. Since we wanted to simulate the effect of the technical debt management approach on classes with technical debt, we chose classes that contained potential technical debt (as evidenced by some indicator) in at least one version of its lifetime (criterion #1).

In order to compare simulated decisions with actual ones, at least one decision to pay off (or pay down) technical debt had to be made in the life time of the selected classes. Thus, we searched for decreases in the number of suspected technical debt instances as evidence that such a decision (explicit or implicit) was made. In other words, the number of technical debt instances in the class must have at some point decreased, which indicates refactoring was carried out (intentionally or not) (criterion #2). In this study we used change likelihood as the indicator of maintainability and of technical debt interest, and we used the number of dispersed coupling code smells as the indicator of technical debt. To narrow down the area that is more likely to have technical debt, the change likelihood of the class over time should observe a similar trend to the number of code smells in the class (criterion #3). Again, such a trend could indicate the effect of technical debt.

According to the results from Zazworka et al.'s study, modularity violation has the highest association with change proneness [153]. Therefore we first tried to identify classes using modularity violation as the type of technical debt indicator. However, no class in the project satisfied the above criteria. Then we chose the code smell “dispersed coupling”, referring to the issue of one method calling too many methods from other classes, as the type of technical debt indicator because this had the second strongest association with change proneness in Zazworka et al. [153]. Based on dispersed coupling, we identified 4 classes from the project satisfying our criteria, as shown in Figure 16. By looking into the source code of these classes, we confirmed these classes did have the coupling problem, which led to higher maintenance cost than the versions not having the problem, as their change proneness indicated. A core

part of our technical debt definition (section 1.1) is that technical debt has a short-term benefit and a long-term cost. In this case, the short-term benefit is that leaving the coupling problem in the class can save the refactoring effort, while the long-term cost is that maintenance effort for the class will stay on a higher level until the problem is addressed. Therefore, carrying the coupling problem, or delaying refactoring the class to address the coupling problem is technical debt. In other words, technical debt was incurred in the evolution of the classes, as the code smell they contained indicated.

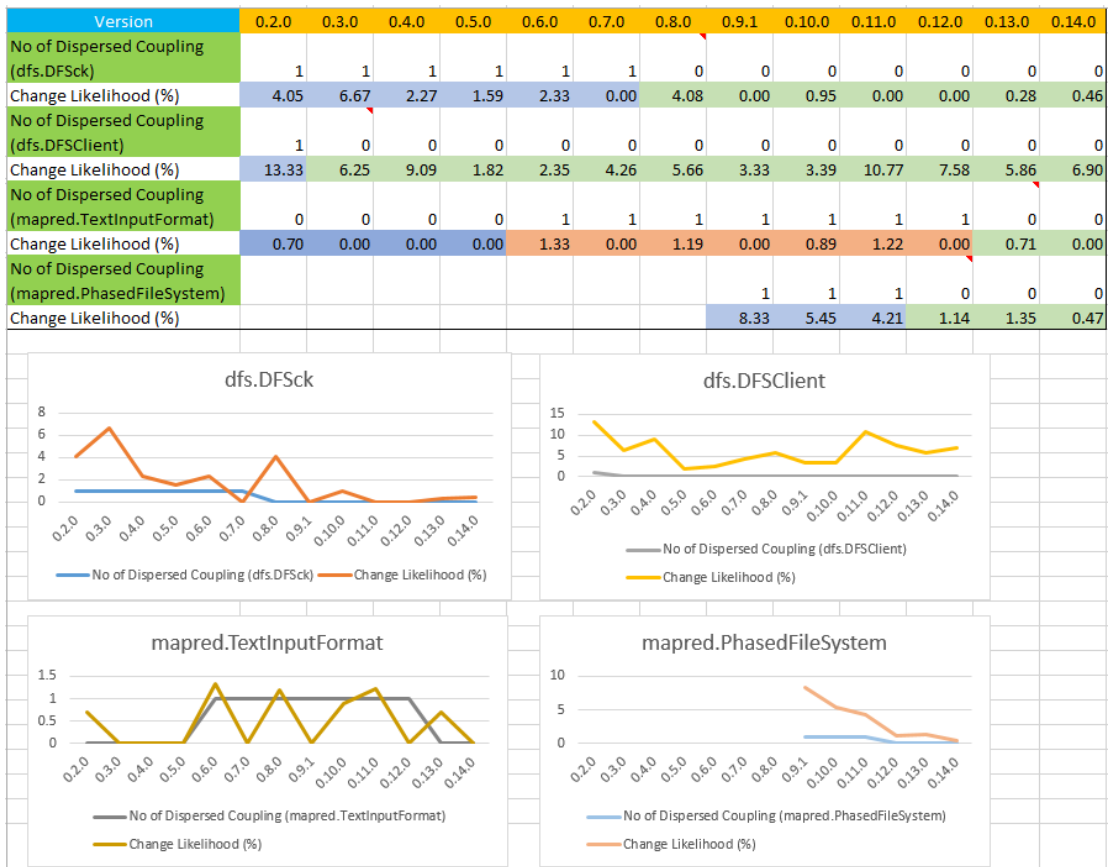


Figure 16. Change Likelihood and No of Potential Technical Debt Instances of the Subject Classes

5.3 Measurement

Informed by the “debt” metaphor, we estimate principal, interest amount and interest probability to characterize the technical debt items in the selected classes. In a scenario in which a decision is being made about whether or not to pay off a technical debt item, the principal is the cost of paying off that item, while the expected interest that would be avoided by paying it off is the benefit. The effectiveness of our approach is measured by the effort that would have been saved if the technical debt management approach had been used to make this decision, as calculated by our simulation approach.

Due to the limited availability of data, especially effort data, proxies had to be used in this study with the following assumptions:

- (1) The effort estimation approach used by the subject project, which they would have used as part of technical debt management, is effective and thus yields accurate results. This assumption may be less valid for new software development organizations, but we consider it achievable for mature software maintenance projects, which our approach is mainly targeted to. This assumption simplifies technical debt principal estimation in the decision simulation for this study by allowing us to use the actual refactoring effort as the estimate.
- (2) The code churn, which is the sum of the lines of code that are added, deleted or modified, is positively related to change effort. The relationship between change effort and code churn has been studied by many researchers. We found supporting evidence for this relationship, as well as negative cases in the

literature [127-129], which were discussed in Section 2.4.1. However, given the limitation of the information availability in the subject project, this was the best way for us to estimate the change effort. With this assumption, we can use code churn as a proxy for change effort.

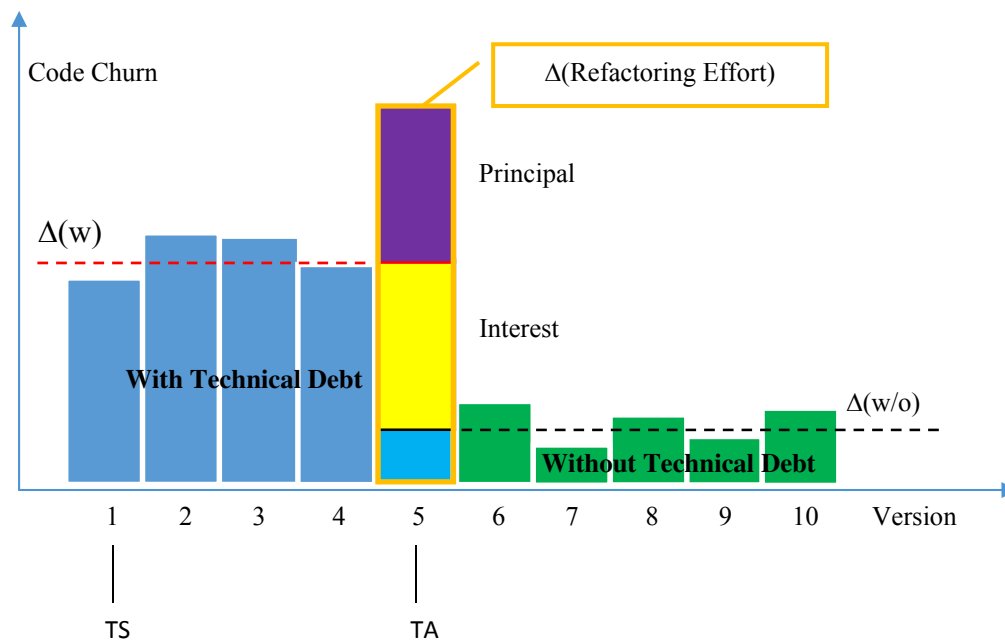


Figure 17. Code Churn History and Refactoring Decision

To help explain the measures defined below, we drew an illustration diagram (Figure 17). Figure 17 shows an idealized case of how technical debt affects maintenance effort. In Figure 17 the class shown carried technical debt beginning from the first version, resulting in higher code churn. The debt was paid off in version 5, requiring extra effort for refactoring. While we don't know anything about how the decision (if any) was actually made to pay off the debt in version 5, we know what the outcome was (it was decided to pay off the debt). Also, we can simulate what decision would have been made using our proposed technical debt management approach.

Our approach requires estimates for principal and interest at the point in time at which the decision is being made (in this case, during planning for version 5). Those estimates must be made, to the extent possible, as they would have been made at the decision time. With Assumption (1), we can assume that whatever estimates that would have been made at that time would have been very accurate, which allows us to use actual values for effort (i.e. churn, by way of Assumption 2) in place of the estimates that would have been made.

The principal of technical debt (P) can be thought of as equal to the effort to refactor the module in the same version where the technical debt was incurred (and hence before any interest has been incurred). This means that P is not equal to all the effort expended in version 5 in Figure 17, but only that part beyond what would have been expended in a typical maintenance cycle, including interest from the existing technical debt. Thus, we calculate the principal, P, using the following equation:

$$P = \text{Max}(\Delta(\text{Refactoring}) - \Delta(w), 0) \quad (1),$$

where $\Delta(\text{Refactoring})$ is the total code churn in the version where the refactoring takes place, i.e. when the technical debt is paid off (version 5 in Figure 17).

$\Delta(\text{Refactoring})$ reflects the total effort spent in this version, which includes the effort to pay off the debt and the routine maintenance effort in a version when the technical debt was present. Note that the routine maintenance effort here includes the interest incurred by the technical debt because the technical debt was present in the version.

$\Delta(w)$ is the average code churn of the class over the versions leading up to the refactoring version, and thus reflects the routine maintenance effort in the versions

where the technical debt was present. Therefore the principal of technical debt is the difference between $\Delta(\text{Refactoring})$ and $\Delta(w)$, as shown in Figure 17. In case $\Delta(\text{Refactoring})$ is smaller than $\Delta(w)$, we consider refactoring is almost effortless and thus can be ignored. To handle such a case, we prevent the calculation of P to drop below 0 in equation (1). The interest amount (IA) is estimated using the following equation:

$$IA = \Delta(w) - \Delta(w/o) \quad (2),$$

where $\Delta(w/o)$ is the average code churn of the class after version 5 and reflects the routine maintenance effort in the versions where there was no technical debt.

Therefore the difference between $\Delta(w)$ and $\Delta(w/o)$ reflects the cost of carrying technical debt, that is, the interest of technical debt, in each version. The interest probability (IP) is estimated using the following equation:

$$IP = CL(w) * N \quad (3),$$

where $CL(w)$ is the average change likelihood of the class in the versions where the technical debt was present. Since the cost of carrying technical debt is incurred when a change is applied, how likely the cost is incurred is determined by how likely a change is required. Therefore, we used the change likelihood of a class as the estimate of interest probability. Because the probability varies with different time frames, a time element has been attached. In equation (3) N is the number of release cycles for which the probability is estimated.

As elaborated in Chapter 3, the principal of the debt, P, is the cost to pay off the debt, while the expected value of the interest, $IA * IP$, represents the benefit, or the cost

that can be saved from not carrying the debt. To simulate the decision on whether or not to pay off the technical debt, for each version, the cost and the benefit are compared. If the cost outweighs the benefit, the debt will be carried over, otherwise the class is refactored and the debt is paid off.

After the decision simulation, we compare the actual decisions with the simulated ones to examine whether and how much benefit, i.e. saved effort, could have been gained from explicit technical debt management. In cases where the actual decision differed with the simulated decision, we performed the simulation again for each version in our data set, in order to determine which version would be the earliest version in which the simulated decision would indicate paying off the debt. This version we denoted TS, and the version in which the debt was actually paid off we denoted TA. This allowed us, then, to calculate the cost savings that would have occurred if the debt had been paid off at TS rather than TA. With Assumption (1), this saved effort (E) is estimated using the following equation:

$$E = \begin{cases} P - IA \times (R_{TS} - R_{TA}) & \text{if } TS > TA \\ IA \times (R_{TA} - R_{TS}) - P & \text{if } TS < TA \\ 0 & \text{if } TS = TA \end{cases} \quad (4),$$

In the first clause, where “TS >TA”, the simulated decision suggests paying off the technical debt later than was actually decided. When “TS > TA”, the technical debt would have been carried longer if the simulated decision had been followed. In this case more interest has to be paid, but the effort to pay off the debt, i.e. the principal, can be saved. The difference between TS and TA represents the number of versions between TS and TA, as shown in Figure 17. Because IA represents the interest

amount incurred in one version, the product of IA and $(TS - TA)$ is the total amount of additional interest that would have been incurred if the simulated decision had been followed. By contrast, “ $TS < TA$ ” means the simulated decision suggests paying off the technical debt earlier than the actual decision did, as was the case in the example in Figure 17. Because the interest incurred in one version with technical debt is almost as high as the principal, the simulated decision suggests paying off the debt in the first version. In this case, when $TS < TA$, the interest would be saved at the cost of paying off the principal. In both cases in equation (4), the difference between the principal and the interest that was paid or saved is the effort that could have been saved if the simulated decision had been made. When the simulated decision is same as the actual decision, i.e. $TS = TA$, no extra effort is saved, as shown by in the third clause of equation (4).

5.4 Data Collection

In addition to the data obtained from Zazworka et al.’s study [153], we collected size information, i.e. Lines of Code (LOC) and code churn for the selected classes in each version. We also reviewed the source code of the selected classes. The code churn information was used to estimate the change effort for decision simulation. Review of the source code helped us understand the causes of the changes and thus facilitated explaining the findings from this study.

5.5 Data Analysis

After the data collection, we calculated principal, interest amount and interest probability for the selected classes in each version. The first version of the classes

was excluded from this analysis to avoid skewing the results with the initial overhead for creating a new class. Then we simulated the decision of whether and when to pay off the technical debt using the proposed technical debt management approach. We considered different time frames (1-3 release cycles) for interest probability estimation and carried out decision simulation accordingly, as shown in Tables 9-12.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
0	Version	0.2.0	0.3.0	0.4.0	0.5.0	0.6.0	0.7.0	0.8.0	0.9.1	0.10.0	0.11.0	0.12.0	0.13.0	0.14.0
2	No of technical debt items	1	1	1	1	1	1	0	0	0	0	0	0	0
3	Change Likelihood	4.05	6.67	2.27	1.59	2.33	0.00	4.08	0.00	0.95	0.00	0.00	0.28	0.46
4	LOC	449	449	458	459	460	460	71	71	69	69	69	69	69
5	Code Churn	16	6	30	3	2	0	538	0	6	0	0	3	1
6	Principal (P)	529	529	529	529	529	529	529	529	529	529	529	529	529
7	Interest Amount (IA)	7.83	7.83	7.83	7.83	7.83	7.83	7.83	7.83	7.83	7.83	7.83	7.83	7.83
8	Interest Probability (1 release)	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00
9	Interest Probability (2 releases)	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00
10	Interest Probability (3 releases)	8.99	8.99	8.99	8.99	8.99	8.99	8.99	8.99	8.99	8.99	8.99	8.99	8.99
11	Actual Decision							✓						
12	Simulated Decision (1 release)	X	X	X	X	X	X	X	X	X	X	X	X	X
13	Simulated Decision (2 releases)	X	X	X	X	X	X	X	X	X	X	X	X	X
14	Simulated Decision (3 releases)	X	X	X	X	X	X	X	X	X	X	X	X	X
15	Effort Saved (E)	474												

Table 9. Class dfs.DFSck for Decision Simulation

The first class, dfs.DFSck, carried technical debt from the Version 0.2.0 through Version 0.7.0 (see row 2 in Table 9). The actual refactoring took place in version 0.8.0, where the code churn had a significant increase compared to the previous versions. The refactoring effort ($\Delta(\text{Refactoring})$) was 538 (LOC), as shown in cell 5H of Table 9. $\Delta(w)$ is calculated as the average value of the code churn (row 5) in versions where the technical debt was present (columns B through G), 9.5 (LOC). Likewise, $\Delta(w/o)$ is calculated using the average values of the code churn after the technical debt was removed, i.e. columns I through N, 1.7 (LOC). Then the principal ($P=529$) and interest amount ($IA=7.83$) are calculated using formulas (1) and (2) in Section 5.3, as shown in Row 6 and Row 7 in Table 9. The interest probability (IP) with different time frames is calculated using formula (3) in Section 5.3, as shown

from Row 8 to Row 10 of Table 9. Then the decision was simulated for each version by comparing P with $IA * IP$. If P is less than the product of IA and IP , the class should be refactored in this version, otherwise, the technical debt is carried through this version, as shown from Row 12 to Row 14 of Table 9. For this class, the simulated decision suggests it would be more cost-efficient to carry the technical debt, i.e. not refactor it ever. This is clearly because the principal of the technical debt (i.e. the cost of refactoring) is too high to justify in comparison with the interest avoided, even considering the interest over several versions. As shown in Table 9, there have been 7 new versions since the technical debt was actually paid off. Thus the saved effort, as calculated using formula (4) in Section 5.3, is 474, as shown in cell 15B of Table 9. In this case, effort equal to changing 474 lines of code would have been saved if the simulated decision had been followed. This holds for decisions with consideration of different time frames. As shown from row 8 to row 10 of Table 9, it is more likely that the interest will be incurred within a longer time frame, but the benefit of paying off the debt, i.e. $IA * IP$, is still much less than the cost, i.e. the principal, for the longest period of time (3 versions). Therefore, the decision on whether and when to pay off the debt remains same for different time frames within which the interest probability is estimated.

By looking into the source code of this class, we found that a big chunk was deleted and several methods were modified in version 0.8.0. As mentioned in Section 5.2, dispersed coupling indicates that a method calls many methods from too many other classes. The modification in version 0.8.0 reduced the calls to methods from other classes. Thus the dispersed coupling issue was gone after the refactoring in version

0.8.0. There were comments in the source code of this class that shed light on the reasons behind the refactoring in version 0.8.0. A large method in this class had actually been copied from another class originally. Because the developers considered duplication to be bad programming practice and a threat to long term maintainability, they decided to refactor the class. The comments indicate that dispersed coupling was actually an issue in the method in the original class from which it was copied, and then this problem was introduced to this class through duplication. Therefore, the main reason for the refactoring was to address the duplication issue, not dispersed coupling. Regardless of the reasons for the refactoring, this class is a case in which the developer was aware of the technical debt in the class and took actions to address the problem. However, due to the lack of technical debt measurement and cost-benefit analysis, the refactoring didn't take place at the optimal point of time.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Version	0.2.0	0.3.0	0.4.0	0.5.0	0.6.0	0.7.0	0.8.0	0.9.1	0.10.0	0.11.0	0.12.0	0.13.0	0.14.0
2	No of technical debt items	1	0	0	0	0	0	0	0	0	0	0	0	0
3	Change Likelihood	13.33	6.25	9.09	1.82	2.35	4.26	5.66	3.33	3.39	10.77	7.58	5.86	6.90
4	LOC	736	766	798	800	800	902	904	914	918	961	972	1007	1219
5	Code Churn	256	64	67	4	3	157	13	16	21	113	71	1269	710
6	Principal (P)	0	0	0	0	0	0	0	0	0	0	0	0	0
7	Interest Amount (IA)	152	152	152	152	152	152	152	152	152	152	152	152	152
8	Interest Probability (1 release)	13.33	13.33	13.33	13.33	13.33	13.33	13.33	13.33	13.33	13.33	13.33	13.33	13.33
9	Interest Probability (2 releases)	26.67	26.67	26.67	26.67	26.67	26.67	26.67	26.67	26.67	26.67	26.67	26.67	26.67
10	Interest Probability (3 releases)	40.00	40.00	40.00	40.00	40.00	40.00	40.00	40.00	40.00	40.00	40.00	40.00	40.00
11	Actual Decision		✓											
12	Simulated Decision (1 release)		✓											
13	Simulated Decision (2 releases)		✓											
14	Simulated Decision (3 releases)		✓											
15	Effort Saved (E)	0												

Table 10. Class dfs.DFSClient for Decision Simulation

The second class, dfs.DFSClient, had a quite different situation. The refactoring, which took place in version 0.3.0, barely required any effort, while carrying the technical debt incurred relatively high interest, as shown in row 6 and row 7 of Table 10. According to the simulated decision, this class should have been refactored

immediately to get rid of the technical debt. Fortunately it was actually refactored in the same version as the simulated decision suggests. Therefore no additional costs (i.e. interest) were incurred. The change included adding to this class two methods that it previously had to call from other classes. The addition of the two methods in this class reduced calls to other classes and thus addressed the coupling problem in this class. It is unclear, based on the available information, whether there were other reasons to refactor this class at that particular point of time. However, it was a right decision as confirmed by the simulation.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1 Version	0.2.0	0.3.0	0.4.0	0.5.0	0.6.0	0.7.0	0.8.0	0.9.1	0.10.0	0.11.0	0.12.0	0.13.0	0.14.0	
2 No of technical debt items	0	0	0	0	1	1	1	1	1	1	1	1	0	0
3 Change Likelihood	0.70	0.00	0.00	0.00	1.33	0.00	1.19	0.00	0.89	1.22	0.00	0.71	0.00	
4 LOC	54	54	54	54	111	111	111	111	122	35	35	18	18	
5 Code Churn	1	0	0	0	117	0	7	0	17	113	0	26	0	
6 Principal (P)	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7 Interest Amount (IA)						36.09	36.09	36.09	36.09	36.09	36.09	36.09	36.09	36.09
8 Interest Probability (1 release)						0.66	0.66	0.66	0.66	0.66	0.66	0.66	0.66	0.66
9 Interest Probability (2 releases)						1.32	1.32	1.32	1.32	1.32	1.32	1.32	1.32	1.32
10 Interest Probability (3 releases)						1.99	1.99	1.99	1.99	1.99	1.99	1.99	1.99	1.99
11 Actual Decision														✓
12 Simulated Decision (1 release)							✓							
13 Simulated Decision (2 releases)							✓							
14 Simulated Decision (3 releases)							✓							
15 Effort Saved (E)		180												

Table 11. Class mapped.TextInputFormat for Decision Simulation

The third class (mapped.TextInputFormat) in Table 11 had a similar situation as dfs.DFSClient, described in Table 10, with respect to the relationship between its principal and interest. The technical debt in this third class could be almost effortlessly paid off, but it had a significant impact on the change likelihood/effort of the class when it remained, as shown in row 6 and row 7 of Table 11. By looking into the source code, we found out that in version 0.6.0, the class was modified to import a new package and add several methods to call the methods in the new package. The modification increased the coupling level between this class and the classes in the

newly imported package and resulted in the coupling issue in this class. In version 0.12.0, the class inherited a new base class, which helped reduce the burden of implementing functions in this class. As a result, the class no longer had the coupling issue beginning with version 0.13.0. It's unknown whether the modification was actually refactoring practice, but it indeed paid off the technical debt in this class. As with the second class, the simulated decision suggests paying off the technical debt immediately, as shown in the cells G12 to G14 in Table 11, but actual refactoring happened after 6 release cycles, which incurred extra interest equal to the effort of modifying 180 lines of code, as shown in the cell B15 of Table 11.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Version	0.2.0	0.3.0	0.4.0	0.5.0	0.6.0	0.7.0	0.8.0	0.9.1	0.10.0	0.11.0	0.12.0	0.13.0	0.14.0
2	No of technical debt items								1	1	1	0	0	0
3	Change Likelihood								8.33	5.45	4.21	1.14	0	0
4	LOC								333	342	344	245	245	245
5	Code Churn								466	14	10	167	0	0
6	Principal (P)									155	155	155	155	155
7	Interest Amount (IA)									12.00	12.00	12.00	12.00	12.00
8	Interest Probability (1 release)									4.83	4.83	4.83	0	0
9	Interest Probability (2 releases)									9.67	9.67	4.83	0	0
10	Interest Probability (3 releases)									14.50	9.67	4.83	0	0
11	Actual Decision											✓		
12	Simulated Decision (1 release)									X	X	X	X	X
13	Simulated Decision (2 releases)									X	X	X	X	X
14	Simulated Decision (3 releases)									X	X	X	X	X
15	Effort Saved (E)		131											

Table 12. Class mapped.PhasedFileSystem for Decision Simulation

The fourth class, mapped.PhasedFileSystem was created in version 0.9.1 with many calls to the methods of the file stream classes that are located in two other packages. Because of these calls, the class had the dispersed coupling issue. The refactoring of this class took place in version 0.12.0, where the code churn surged to 167 lines of code, indicating high principal of this technical debt, as shown in cell 5L of Table 12. The two file steam classes called by this class were replaced by one single class and thus reduced the coupling of this class with other classes. After the refactoring, the

code churn went down to 0 in version 0.13.0 and 0.14.0, as shown in cells 5M and 5N of Table 12. In terms of change proneness, the refactoring was effective. However, the refactoring cost was too high to justify this technical debt repayment decision. The simulated decision, by contrast, suggests carrying this technical debt all the way through. Although more interest would have to be paid for the simulated decision, it is still more cost-effective than the actual decision. A total of the effort equal to changing 131 lines of code would have been saved if the simulated decision had been followed, as shown in cell 15B of Table 12.

5.6 Discussion

In this study we tracked four technical debt instances located in four classes of the subject project. As mentioned at the beginning of this chapter, the main goal of this study is to uncover the potential benefits gained from explicit technical debt management. We achieved this goal by simulating the decisions on when to pay off technical debt for the past versions of a class and then comparing them with the actual decisions. Therefore, the candidate classes must have at some point contained technical debt and at least one decision on paying off the technical debt had to have been made in the lifetime of the classes.

The study began with identifying candidate classes that can be used for decision simulation. Using the analysis from Zazworka et al. [153] as a starting point, we searched for classes that exhibited technical debt at some point (as indicated by the presence of the dispersed coupling code smell) and that eliminated the technical debt

at some point (as indicated by the absence of the dispersed coupling code smell).

Finally we chose four classes for the decision simulation.

The results of the data analysis, i.e. decision simulation, show that our proposed debt management approach suggests different technical debt repayment strategies in most cases (3 of the 4 classes) compared to the actual decisions. Where the simulated decisions differed from the actual ones, significant effort could have been saved if the simulated decisions had been followed. For example, a total of the effort equal to 474 lines of code could have been saved for the first class if it had not been refactored. Similarly, the third and fourth classes could have also saved effort comparable to the effort of constructing the classes, i.e. the size of the classes. Among these classes, only one actually implemented the same technical debt repayment strategy as the simulated decision suggests. Thus, most of the refactoring didn't take place at the optimal point of time from the perspective of cost saving. The differences between the simulated decisions and the actual decisions indicate that technical debt information was not fully taken advantage of to reduce overall project cost.

For classes 2 and 3 (see Tables 10 and 11), the simulated decisions are to pay off the debt in the same version in which it was incurred. Class 2 was indeed refactored immediately after the technical debt was incurred. This raises the question: why was it incurred in the first place? As mentioned in Chapter 2, there are various causes for technical debt. For example, the project may lack the development resource at the particular point of time, or an inexperienced developer may create some low quality work. We might not be able to conclude how the technical debt in these two classes

was incurred, based on the results of this study, but it is clear that, in the case of class 2, incurring the debt was not based on an informed decision, given that it was paid off almost immediately. In these cases the debt itself could have been avoided if a careful cost-benefit analysis had been performed, as our technical debt management approach proposes, at the point of deciding whether or not to incur the debt.

For classes 1 and 4, the analysis results show that the technical debt should have been carried throughout the period analyzed rather than being paid off because the benefit, i.e. interest that was saved, was much lower than the cost of refactoring the classes.

The simulated decision seems to contradict the common sense of technical debt management and problem solving as “refactoring a module early to avoid debt accumulation” and “fixing a problem before it gets big” are generally considered good practice in software development. But we would argue that technical debt management should go beyond the heuristic rules that common sense suggests because different technical debt instances may have quite different situations. The idea of explicit technical debt management is not to develop general strategies that treat technical debt as a whole, but to propose managing technical debt in a scientific manner. Decisions on technical debt management should be based on analysis and measurement, as our proposed technical debt management approach suggests.

Through this study, we have observed that explicit technical debt management could bring significant benefit, in the form of cost saving, to the project in comparison with the implicit technical debt management practiced by the project team, if at all.

At the same time, we also realize there are validity threats to this study. The main threat to the internal and construct validity of this study lies in the measurement aspect because of the assumptions we made. Due to the limited data availability, we used the actual refactoring effort as the estimate of the technical debt principal for decision simulation with the assumption that the estimation approach used by the project (and which would have been used in the cost-benefit analysis if technical debt had been managed explicitly) was mature enough to yield accurate results. If this assumption doesn't hold, then the decisions made may not be optimal, and the benefits we observed in this study would be higher than what the project could actually gain from explicit technical debt management. For the same reason, we had to use code churn as the proxy of change effort with the assumption that the lines of code that are changed reflect the effort to change them. Besides the risk of overstating the benefits of technical debt management, this assumption risks measurement error of the technical debt and hence the benefit from explicit technical debt management. To evaluate the potential measurement error, we studied the evolution history of the four classes, i.e. the changes to the source code in the involved versions. This helped us understand what the changes were and hence helped us determine if the code churn in these classes reflected the effort to make the changes. We were able to determine, subjectively, based on the author's programming experience, that code churn is largely proportional to change effort for these classes. Therefore, we do not believe that using code churn as the proxy for change effort for this study introduces large measurement errors.

Besides the above assumptions, we also simplified the estimation process of technical debt in the decision simulation. Instead of estimating the principal and interest amount of technical debt in each version, we used average code churn over the related versions, as prescribed by formulas (1) and (2) in Section 5.3. Thus the estimated principal and interest amount of the technical debt in each class remains constant over the versions involved in the decision simulation. This simplification resulted in extreme cases – the technical debt in a class should be either paid off in the same version where it was incurred, or carried indefinitely. In reality, it is likely that some cases fall in between the two ends because estimates of principal and interest need to be adjusted to reflect changes in the situation over time, and to be refined as more information becomes available. For example, we may decide not to refactor a class with technical debt right away due to the high principal and low interest of the debt, but several versions later the cost of carrying this technical debt, i.e. the interest, might significantly increase because of a change in another class that it is coupled with. In this situation it is no longer economical to keep the technical debt in the class. Therefore, the decision should change from carrying debt to paying it off. This example shows there may be cases in which the time span between the two points of paying off technical debt for the simulated decision and the actual decision, i.e. the distance between TA and TS, is shorter than those in the selected classes for this study. According to formula (4), a smaller difference between TA and TS leads to less saved effort, which means the actual benefit from explicit technical debt management may not be as high as the result from the decision simulation in this study.

Another threat to the internal validity of this study comes from selection of the subject classes. In this study the technical debt instances in the selected 4 classes are independent of each other, which allowed the decision simulation approach to be applied to the classes individually. Given the small number of selected classes, the technical debt instances in them may not be representative and the relationship between technical debt instances in the project could be much more complicated. In such situations the optimal decisions on technical debt repayment may be different from the simulated decisions in this study and hence the effort that can be saved by explicit technical debt management could be different as well.

In spite of the limitations, this study still sheds light on the effect of explicit technical debt management on software projects. The compromise we took on measurement and simplification of the management approach would only impact the magnitude of the results from this study, but not change the way in which explicit technical debt management affects software projects, i.e. using technical debt information to help software managers make better decisions. In the worst case, no extra benefit comes from explicit technical debt management when the decisions made on all technical debt items in a software project by the explicit technical debt management approach are the same as those decisions made through implicit technical debt management, which is highly unlikely, as demonstrated by the cases in this study.

Chapter 6: The Tranship Study

This research aims at technical debt theory construction through the study of how technical debt can be measured and monitored. The emerging theory addresses the cost and benefit characteristics of explicitly managing technical debt in comparison with implicit management. To investigate these cost-benefit characteristics, we designed two types of studies – retrospective study and case study, as elaborated in Chapter 3. The study described in this chapter falls into the second type, that is, we use the selected software project to investigate the costs of explicit technical debt management. In this study we explored the costs of using our proposed approach by tracking the technical debt management activities in an on-going software project.

The research questions for this study are:

- (1) What are the costs of managing technical debt using the proposed approach?
- (2) How does technical debt information contribute to decision making?

This study has been published as a peer-reviewed article in an academic journal [154].

6.1 Subject Project

The subject project was a software application for water vessel management from a Brazilian software company that provides enterprise-level software development, consulting and training services, as described in Section 3.2.2.3.

The project began in late 2011. The first release of the application was approved by the customer in May 2012. Since then, a new customer release was delivered every sprint. The case study began with baseline data collection in March 2012. Afterwards,

the project started managing technical debt explicitly using our proposed approach. We observed and collected data on the technical debt management process until the middle of November 2012. The case study ended in the middle of December, 2012 with the follow-up interview being completed.

6.2 Study Process

Due to the distance and language, we did not have direct contact with the subject project. Therefore, this study involved a company contact, who was also part of the research team and was in charge of translation, data collection (e.g., interviewing the project personnel), and communication between the project team and the research team. All communication with the project team was in Portuguese, while the research team communicated in English.

We followed the general process described in Section 3.2.2.1 to carry out this study, but we present here the steps with more details and particulars to this study.

- (0) Preparation: the company contact collected baseline data in terms of effort, cost and productivity of the project. We designed a simple spreadsheet to aid in managing the technical debt list and to collect data on its use. The list was kept under version control, along with other project documents.
- (1) The company contact trained the project team on how to manage technical debt using the proposed approach and how to document and report information required by this study. Training materials were prepared by the research team in English and translated to Portuguese by the company contact.

- (2) The project team identified technical debt items in their system to prepare the initial technical debt list. In the first release cycle, during which the project was tracking technical debt for the first time, the team members provided data to the research team regarding the amount of time they spent using the technical debt list and the problems and questions that came up.
- (3) After the first release, the research team reviewed the technical debt list, the changes made by the project team and other data collected in the first release cycle. Then the research team asked questions about the project team's experience, the data they provided, and the process they followed. This began a back-and-forth discussion, which continued until all the questions were addressed, the research team was satisfied with the types of data provided by the project team, and the project team was clear and comfortable with the proposed approach.
- (4) The project team continued tracking the technical debt items in the successive three sprints using the approach. In the process, they documented the decisions made concerning technical debt, the principal and interest paid and/or avoided, and the costs (i.e. effort) of using the technical debt list. In the end of this study, the company contact interviewed the project leader, who was in charge of release planning, about his decision making process and the effect of the approach on the project.

6.3 Data Collection

In step 1 of the case study process, we needed to collect baseline data about the effort, cost, and productivity of the project, which was to be used later for comparison. The

baseline data we collected is summarized in Table 13. It gives an overall picture of the size and current productivity of the project. This data was gained via interview with the project leader (conducted by the company contact).

No. of Software Modules	8
No. of Use Cases	95
Hours/Month	120
Average Person-hours/Use Case	37
Average Person-hours/sprint planning	5

Table 13. Baseline Size and Productivity of the Project

At the same time, the project team created the initial technical debt list. This activity began with a training on the concept of technical debt. This training was performed in Portuguese by the company contact and took about 30-40 minutes. After that, the team members had a chance to ask questions. The definition of technical debt presented to the team was: incomplete, immature, or inadequate artifact in the software development lifecycle which in turn adds more constraints on future maintenance tasks and makes modification more difficult, costly and unpredictable [17]. Please note that the definition of technical debt for the project team took a different perspective from the one we use, which was presented in Section 1.1 where we discuss the technical debt concept, but they are not inconsistent. In Section 1.1 we define technical debt in terms of the behavior associated with it (i.e. short-term benefit and long-term cost) because this perspective facilitates managing technical debt, which is the target of this dissertation research. But in the training of the project team, we used another technical debt definition, which focuses on its appearance, as this perspective is relevant when identifying technical debt, which was

the first step in using technical debt information for release planning in their project. For example, one technical debt item in the project was that the class `FaturaControlle` includes some hard-coding on the service type, which should be retrieved from a database. If this technical debt item is described in terms of behavior, it is that hard coding on the service type in this class reduces the effort to construct this class, but it will cause data inconsistency issue in case the service type is updated in the database. A core part of the technical debt definition (section 1.1) is that technical debt has a short-term benefit and a long-term cost. In this case, the short-term benefit is the reduced effort to construct this class, while the long term cost is the potential data inconsistency problem that needs extra effort to fix. Therefore, delaying refactoring this class to remove the hard-coding is technical debt. As mentioned in Chapter 2, there are multiple dimensions to categorize technical debt and several classifications have been proposed. Among these classifications, the one Rothman proposed [26] makes it clear that technical debt has different sources and forms, and hence may need different measures for identification, and approaches for management, which is the center of this dissertation research. In addition, categorizing technical debt by development phase is natural to software developers. It would be easier for them to understand the technical debt concept and thus fulfill their tasks for this study more effectively. Therefore, we decided to use Rothman's classification as part of our technical debt training for the project team. Specifically, we explicitly defined the following types of technical debt:

- **Design debt:** any kind of anomaly or imperfection that can be identified by examining source code and/or related documentation, that leads to decreased maintainability if not remedied;
- **Testing debt:** tests that were planned but not executed;
- **Documentation debt:** documentation that is not kept up-to-date;
- **Defect debt:** known defects that are not yet fixed.

Finally, the development team was also trained on how to report technical debt items using the technical debt list. After that, the development team started to identify technical debt items. Basically, for this task, each member of the team reported technical debt items based on their role and activities on the project. Some of the developers looked manually for technical debt items in the parts of the source code with which they were working and familiar, others just recalled instances of incomplete, immature, or inadequate project artifacts that they had encountered. Some of the testers examined test reports or requirements specifications to identify debt items. No analysis tools were used. For each reported item, the team member assigned its type (design, testing, documentation, or defect).

The team members were also asked to estimate rough values of principal, interest amount, and interest probability. To estimate these values, the development team was instructed during the training to use their own experience and expertise to make their best guess, and to use high, medium, and low for the initial estimates. After that, they were instructed to refine those values numerically, based on their own experience, during sprint planning. Thus, initially, all technical debt items had high, medium, and low estimates for principal and interest. Later, the team analyzed each technical debt

item considered during sprint planning and estimated numerically these values.

Information about each technical debt item was collected into a single spreadsheet (example shown in Figure 18).

Given the objective of this study, the cost of using the proposed technical debt management approach is the core category of data. The cost data includes costs of meetings, effort spent managing the technical debt list, and time spent gathering data to feed into technical debt management. These data were collected directly by the project leader during sprint planning over each of the 4 sprints included in the case study. For example, for each relevant meeting, the project leader recorded the number of people present and the length of the meeting, thus yielding the overall cost of the meeting. During the sprint planning meeting, the project leader asked team members to report time spent gathering data relevant to technical debt management. The project leader also recorded his own time spent managing the technical debt list and in other technical debt related activities.

Data was also collected concerning the changes made to the technical debt list, including when a technical debt item was paid off or incurred, the updated estimates of the principal and interest, and the actual effort of paying off an item. Collecting this change data allowed us to monitor the implementation of the technical debt management approach. The change data was recorded in the same spreadsheet as the technical debt list, which was kept under version control.

Another type of data that was collected was about the technical debt decisions made in the process. We were interested in the decision making process, and the ways in

which it was influenced by technical debt information. Although this study focuses on the cost side of technical debt management, we also expected that information about decisions might reveal some of the short-term benefits that could be observed from using the approach, as well as problems and suggestions regarding the usefulness of the approach. The decisions themselves were recorded as changes to the technical debt list, and more in-depth information on decisions was collected through the final interview with the project leader.

ID	Identification Date	Owner	Type	Location	Description	Affected Scope	Coupled Scope	Principal	Interest Amount	Interest Probability
1	3/15/2012	Marcos, Edmilson	Design	Lingada.java, ItemGuarnicao.java, FuncionarioAccess.java, ItemTreinamento.java, RestricaoFuncionario.java	Name, but this data is stored in an external database. Thus, it is not possible to access the employer information as we can do with the other information stored in the system database.	Lingada.java, ItemGuarnicao.java, FuncionarioAccess.java, ItemTreinamento.java, RestricaoFuncionario.java		25h	5h	25%
2	3/15/2012	Marcos, Vagner	Documents	MALO_SGI_TRANSHIP_Especificacao_Requisitos.doc	The Module of Allocation doesn't have a requirements specification document.			25h	7h	40%
3	3/15/2012	Edmilson	Design	AlocacaoController.java alocacar[...]method	This method checks the data about employers in Access database and, after that, sort the data according to a set of criteria. However, the method is currently very large and need to be entering a zip code in the inclusion of a	com.kalisoftware.tranship.controller.AlocacaoController		8h	3h	10%
4	3/15/2012	Fabricio	Defect	customer inclusion (ClienteController.java) company inclusion (EmpresaController.java)	company or customer, the system returns some districts as cities, but those districts don't exist on IBGE's table. This fact can cause errors when an invoice is created.	com.kalisoftware.tranship.controller.empresaController		12h	5h	50%
5	3/15/2012	Marcos, Fabricio	Design	FaturaController.java Nfe.java	I need to make a verification with the activity name when I need to identify a service type or bill. This information is fixed in the code, and can bring errors when some update is	com.kalisoftware.tranship.controller.FaturaController com.kalisoftware.tranship.servico.notaFiscal.Nfe		8h	4h	75%
6	3/15/2012	Marcos, Fabricio	Design	DespesaController.java	There is a need to locate the cost codes according to their numbering. However, this information is fixed in the code and any update or error on it can bring bad side effects to the system.	r.conciliacaoController, com.kalisoftware.tranship.controller.despesaController, com.kalisoftware.tranship.controller.componenteController		40h	20h	100%
7	3/15/2012	Fabricio	Design	AutoTracJob.java	High consumption of memory causing stack overflow. This method needs to run every 2 minutes, but because of this problem was only running once a day to keep the data updated.	com.kalisoftware.tranship.servico.AutoTracJob		15h	4h	15%
8	3/15/2012	Marcos, Fabricio	Defect	Parser.java	In the import of "Machine Daily" is necessary to associate the "Machine Daily 1" to "Machine Daily 2" (this mapping is one-to-one), but sometimes, we can have more than one Machine Daily 2. In this case, the system	com.kalisoftware.tranship.autotrac.parser		8h	0h	0%
# Technical Debt Management Activity (Sprint I) Effort (person-min)										
1	Evaluate the current level of interest amount and probability of the technical debt items									120
2	Create a traceability matrix to help manage the technical debt items									30
3	Look for technical debt items in the project's artifacts									10
4	Meet with developers									10
5	Understand the technical debt items on the technical debt list									10
6	Fill out the technical debt list									40
Total									220	

Figure 18. Data Collection Spreadsheet

In short, we collected 6 categories of data: (1) baseline data of the project, (2) costs of using the technical debt management approach, (3) changes in the technical debt list (4) decisions on when and what technical debt items should be paid off, (5) benefits

of using the technical debt management approach, and (6) problems and suggestions for the proposed technical debt management approach.

Data (1) were collected as described above. Data (2) – (4) were collected from the project team when they implemented the approach. These data were collected using the simple spreadsheet (shown in Figure 18) we designed in step 2 of the case study process (Section 6.2). Data (4) - (6) were collected through the final interview, after the project team finished tracking the technical debt items.

The spreadsheet we used for data collection had two parts, one for the technical debt list and one for the effort data. The technical debt list had columns for technical debt attributes and rows for technical debt instances, as shown in Figure 18. The table for recording technical debt management activities and effort was very simple to allow the project leader to define the activities and categories of effort. For example, the first entry in the table shown in Figure 18 represents the project leader and two developers spending 40 minutes (so 120 person-minutes in total) in evaluating the principal and interest of the technical debt items in the list. The initial form of the spreadsheet was designed by the research team. When it was sent to the development team, they revised it according to the information availability and used it to log the data they collected. The spreadsheet was updated and version controlled to reflect changes in the technical debt items and effort spent in technical debt management at the beginning of each sprint.

The set of questions used to interview the project leader at the end of the case study consisted of four sections – the profile of the interviewee, the original management

practices, the experience of using the proposed technical debt management approach and technical debt measurement. The questions in the first section elicited the project leader's experience level in software development and role in the subject project. We used the questions in the second section to check if the participant had knowledge or past experience on managing technical debt, which would help us determine the effectiveness of the proposed approach. The third section was the main section of the interview. In this section we asked the project leader to elaborate on how he performed technical debt management, the factors he considered when he planned a sprint, how the technical debt information was used in the sprint planning, the impact of the proposed approach on the project, and the problems and the benefits he observed. The questions in the last section were to elicit how effective the current cost measures of managing technical debt are.

The questionnaire was developed by the research team in English. Then the company contact translated it to Portuguese and conducted the interview. After the interview, the company contact translated the results into English for analysis by the research team.

6.4 Data Analysis

The cost data gathered from the sprint planning meetings was compared with the project's baseline to determine the effect of the proposed technical debt management approach on the project. Besides the cost data, we collected a significant amount of qualitative data through the interview and the technical debt list. These qualitative data were coded according to a coding scheme developed based on the research

questions. The coding was performed using a text editor, by highlighting chunks of text and annotating those chunks with the name of the associated code. The main codes of the scheme are shown in Table 14. With these codes, the data were grouped to form the chain of evidence to explain the changes that explicit technical management brought to the project and to reveal the rationale behind the decisions made in the sprint planning. It also helped us understand the benefits and problems of the proposed approach.

Main Code	Definition
Cost	Information regarding the cost incurred by technical debt management activities
Benefit	Information regarding the benefit related to the application of the technical debt approach
Decision	Decisions in the course of the study regarding sprint planning and technical debt repayment
Approach	Application process and user feedback of the technical debt approach
Measure	Information regarding technical debt evaluation method and activities

Table 14. Main Codes of the Coding Scheme

6.5 Results and Findings

The case study began in March 2012. First the project baseline was collected and the initial technical debt list was created. By the end of September 2012, the project team finished documenting the initial 31 technical debt items. Among these items design debt accounted for the majority, but there were also other types of technical debt. We can see from Table 15 that a new type of debt (different from those in the initial list of types from the literature) was suggested and reported by the development team: process debt. The team members described process debt as variations in how some activities were performed because the process, as originally designed, was no longer appropriate for the project. Thus process debt consists of changes that the team has

made to the defined process, in an ad hoc manner, to deal with the fact that the process was not updated to reflect the needs of the current project.

The initial rough estimates for principal and interest ranged from low to high. Later numeric estimates for principal of these items ranged from 4 to 40 person-hours, while the estimated interest amount ranged from 0 to 20 person-hours. The interest probability also varied greatly (0%-100%).

Type	Design	Defect	Documentation	Testing	Process	Total
No. of items	18	7	4	1	1	31

Table 15. Technical Debt Items by Type

The project team started tracking technical debt from the beginning of October 2012. Since some of the originally identified technical debt items had already been paid off at that time, the initial technical debt list consisted of 22 items. Then the research team interacted with the project team to review the collected data and the process they followed to manage the technical debt during the first sprint. After that, these technical debt items were tracked over the subsequent 3 sprints with their principal and interest being re-estimated as the situation of the project changed. Meanwhile the project leader kept the technical debt list updated as the existing items were paid off and new items were identified. Table 16 shows the changes in the technical debt list over the four sprints. While it can be seen from Table 16 that the number of technical debt items decreased from 22 to 15 (i.e. by 32%), we cannot say that the amount of technical debt was reduced by 32%, or that any reduction was due to explicitly managing technical debt. Capturing all the technical debt in the project was not a goal

of this study, nor was reducing technical debt, thus we cannot draw any conclusions from the decrease in the number of debt items being tracked.

Items	Sprint I	Sprint II	Sprint III	Sprint IV
Total (before the sprint started)	22	20	15	15
New	0	0	0	1
Paid-off	2	5	0	1
Deferred to the next sprint	20	15	15	15

Table 16. Changes in the Technical Debt List over the Sprints

6.5.1 Costs of Technical Debt Management

In the four sprints where the proposed technical debt management approach was applied, the project leader documented and reported all the work related to technical debt management, including the responsible person and the time spent on it.

According to these reports, the costs of managing technical debt came from several different activities, including identification of technical debt items, analysis and evaluation, communication and documentation. The project leader identified these activities in vivo, i.e. they were not predefined as categories of effort. Among these activities, identification (which in this case only happened once, in the fourth sprint) refers to developers finding low-quality software artifacts deemed to be technical debt instances from the modules they were in charge of. This was not done during the sprint planning meeting, but ahead of time in a fairly ad hoc way. Analysis and evaluation refers to understanding the technical debt items, estimating their principal and interest, creating a traceability matrix and mapping them to project artifacts. Communication refers to meetings of the project leader with the developers to gather

information for technical debt management. Documentation refers to updating the technical debt list according to the analysis and evaluation results, documenting the decisions related to the technical debt items and the implementation results, including the date and the actual effort of paying them off. Table 17 shows the cost of each activity in the sprints.

Items	Sprint I	Sprint II	Sprint III	Sprint IV
Identification	0	0	0	0.5
Analysis and evaluation	2.8	0.3	0.4	0.3
Communication	0.2	0	0	0
Documentation	0.7	0	0.1	0
Total	3.7	0.3	0.5	0.8

Table 17. Costs (in person-hours) of Technical Debt Management Activities over the Sprints

From Table 17 we can observe that the major cost came from analysis and evaluation work. This was confirmed, through the final interview with the project leader, that analyzing and evaluating technical debt items was the most time-consuming work. Technical debt Identification seems to be high-cost work as well according to the figures for Sprint IV, but this is not conclusive because only one technical debt item was identified in this sprint. In the second and third sprints, the project leader focused on the items already on the technical debt list and didn't identify new technical debt items. Hence there was no identification effort in those two sprints, but there was in Sprint IV. The initial technical debt items for this case study were identified in the preparation phase, which also involved training/learning of the proposed approach. Since it would have been difficult to capture the effort involved in preparing the initial debt list, and because such effort is not particularly relevant to technical debt

management in general, we excluded the identification effort from the first sprint. The time required for identification likely varies widely depending on how the debt is identified, e.g. as part of a code review vs. using a static analysis tool. In the case study, technical debt items were identified ad hoc, by suggestion of developers, not through the use of any tools or formal processes. One could speculate that technical debt identification might take less time when using code analysis tools or as part of an existing process like a code review.

The cost numbers in Table 17 also indicate the proposed approach has a significant initial overhead. When the technical debt list was used for the first time, the project team had to go over each item to understand it and re-estimate its value. But in the subsequent sprints they only needed to examine those items that had changed. Therefore, there was a significant drop in the cost from the first sprint to the subsequent ones. But even in the first sprint, the cost of managing technical debt was only about 10% of the cost to implement an average use case.

According to the baseline data, it took 5 person-hours on average for the project leader to plan a sprint before the case study, i.e. before starting to manage technical debt explicitly. To determine how much effort was added to the sprint planning activity by explicit technical debt management, we look to Table 17 to calculate the average effort in managing technical debt per sprint – 1.325 person-hours. Since most of this effort was expended while planning each sprint, we can conclude that, after applying the technical debt management approach, the average effort devoted to the planning phase of a sprint increased to 6.325 person-hours, or a 26% increase.

Although a 26% increase seems to be significant, 1.3 person-hours is a small number compared to the cost of implementing a use case, which is 37 person-hours on average. Moreover, this increase was largely due to the aforementioned “initial overhead”. The first release cycle had a 70% increase in sprint planning effort (3.7 hours) due to technical debt management, but then the extra cost went down to just over half a person-hour per sprint, or about a 10% overhead.

6.5.2 Planning Process and Decision Factors

As mentioned in Section 6.3, we collected decision data from the technical debt list as well as the interview. By analyzing these decision data, we have obtained a clear picture of how a new sprint is planned and what factors could affect the decisions about technical debt. When the project leader plans a new sprint, he first “takes into consideration the current delivery (a set of use cases to be deployed to the customer)”. Besides the delivery, “there are also improvement points requested by the customer”. “With this in mind”, the project leader will then check other factors such as the technical debt currently in the project. In other words, the project leader always gives the highest priority to the features and enhancements agreed to by the customer. The project leader’s top concern is on-time delivery. If the project leader cannot make the delivery on time, he “runs the risk of losing the customer”. In this situation he would delay some work to save time even if he has to pay high interest later because “paying high interest is still better than losing the customer”. This rationale accounts for the major cause of technical debt incurrence in the project where most technical debt items were given an explanation similar to “this should be done, but we do not have

time in this sprint”. Therefore, “time and resource availability are crucial in deciding whether to delay or pay off the debt.”

If the schedule and resources allow, the project leader will “check the value of the debt items, i.e. the principal and interest.” The items with high interest amount and high interest probability will be more likely to be chosen to be paid off. But the final decision is also based on a “look for the tasks in the backlog.” The project leader considers “a good opportunity to pay off a debt item if it is related to some backlog item or currently being maintained by a developer” because they can hit two birds with one stone. He also gave an example in which he decided to pay off a debt item associated with the function of generating financial reports, which was being developed at that time. This strategy is consistent with the proposed technical debt management approach in that these items have reduced principal. Besides these factors, the project leader also considered the impact of debt on other features of the system. The debt items that may affect more features will be given higher priority to be paid off.

In short, our results show that the actual process of deciding whether or not to pay off technical debt was affected by many factors. Among these factors, customer expectations have the top priority, followed by availability of the development resource, the interest of the technical debt items, the current status of the debt-infected modules and the impact of the debt on other features.

This is the first time we observed a real software project attempt to explicitly manage technical debt. While we provided a process for technical debt management to the

project team, we expected that the team would have to modify it, or use it in a slightly different way than we intended, in order to serve their needs. This turned out to be true. In the proposed approach, we assumed that all the factors that may affect the decision on paying off or deferring technical debt could be reflected in the values associated with technical debt, i.e., principal and interest. For example, if a technical debt item needs to be paid off immediately due to its criticality, the interest estimate of this debt item should be increased because if it is not paid, it will cause serious problems, and thus higher cost in future. So in our proposed approach, our intent was that the priority of each debt item could be determined using only its principal and interest. By contrast, the actual decision making process, as observed in the case study, worked differently. The values of principal and interest were first estimated without considering other factors such as team availability and module criticality. In other words, principal and interest estimation was simple and rough, and was strictly focused on effort. During decision making, then, these estimates of principal and interest were combined with other factors to determine the final priority of a debt item. This meant that, sometimes, a debt item that should have had high priority because of its principal and interest estimates was deferred for reasons that could have, but were not incorporated into the principal and interest estimates, e.g. resource availability. Thus, although we had designed our proposed approach so that all factors could be incorporated into the notions of principal and interest, and so principal and interest could then be used as the sole criteria in decision making, the development team found it preferable to simplify the estimation of principal and interest

(concentrating strictly on effort), but to be more holistic in actual decision making, by taking other factors into consideration at that time.

In summary, sprint planning could be affected by many factors, and not all these factors are easily incorporated into the notions of principal and interest. Effort (either expended or saved) is generally not the most important factor. The actual process, when technical debt management is incorporated, may deviate from the one we prescribed. In spite of these differences from our proposed technical debt management approach, the approach actually used in the case study is still an example of explicit technical debt management, and thus our goal of observing this process in an actual project was achieved. The ways in which the team tailored the process provides important insight into how technical debt management should be integrated into a real software development process. In addition, the process differences provided us with the opportunity to consider more factors and improve our approach, which is one of the objectives of our larger research agenda. Eventually we expect to provide a way for managers to prioritize technical debt items along with enhancements and bug fixes.

6.5.3 Benefits and Impact

Although this study focuses on the costs of explicit technical debt management, it was natural that the benefits observed in this study, especially from the interview, were also collected with the cost information as they are two sides of the same issue – technical debt management.

During the entire time period of the case study, there were no big changes in the development process, the staff and the project management, except the application of the technical debt management approach. Moreover, the project leader confirmed that he didn't have any experience in technical debt management before applying our approach. Therefore we are confident that the impact and benefits observed can be mainly attributed to the application of the approach.

An important benefit of the technical debt management approach is the increased awareness of the problems that may be overlooked otherwise. When we asked the project leader about his general impression of the technical debt management approach, he stressed that it was the technical debt list that helped him realize “many things were being overlooked in the project.” He also admitted that in the technical debt list “there were so many points that had not been identified early in the project and could have been better handled earlier.” When we asked the project leader if any of the decisions he made would have been different without using the approach, he gave a positive answer and further explained that he “would be restricted to the problems reported by the customer without the [technical debt management] approach and would not consider code fragments that should be improved, which can often be treated together with bug fixes identified by the user”. Thus the problems could have been addressed earlier to avoid serious impact. This statement is consistent with the finding from the SMB study (elaborated in Chapter 4) in which the subject project suffered from high change cost because their decisions in the early stage didn't consider the impact of the technical debt in their project. Moreover, the estimates of the principal and interest “make it easy to understand the gravity of the problem, thus

allowing a quick decision” because he had to know “how much effort a debt would cost before allocating developers to pay it off”. When we asked the project leader whether the integration of the approach impacted on his sprint planning, he responded that “just the fact of thinking about using it already impacted my activities because I started, in my decisions, to consider the impacts of not paying off a debt in a sprint.”

6.6 Discussion

This study focused on the cost side of technical debt management. The objective was to uncover the costs that could be incurred by the proposed technical debt management approach. To achieve this objective, we selected a live software project and applied the technical debt management approach to it. In the course of using the approach during sprint planning, technical debt items were identified, the effort spent in managing technical debt was logged, and the decisions were also documented. By tracking technical debt in the subject project, we have identified a set of technical debt management activities and measured their costs. From the cost information we have gained insight into the feasibility of the technical debt management approach. In addition, this study also helped us understand how technical debt information contributes to decision making in release planning. Along with the benefits exhibited by applying the approach, we have been able to characterize both costs (quantitatively) and benefits (qualitatively) in a way that can help future projects and organizations decide if managing technical debt is feasible for them. Meanwhile, we also noticed some limitations of this study and hence will address them in our future work.

6.6.1 Costs and Benefits of Explicit Technical Debt Management

According to the findings of this study, technical debt management could incur different types of costs. Among these costs, analysis and evaluation took the top position, accounting for the majority of the total management cost. Analysis and evaluation was closely tied to the difficulties of quantifying and estimating the technical debt parameters, principal and interest. Thus, future advances in technical debt measurement [155] are likely to have an impact on the overall cost of technical debt management. Identification of new technical debt items also incurred high cost, and many current research efforts are aimed in this area [23, 119, 156].

In the subject project the technical debt items were identified manually, a time-consuming task performed by the developers who had to look into the code, compare it with the criteria, and recall what they have done in the past. This situation would be similar in projects even with different characteristics unless an effective technical debt identification approach is applied.

Likewise, evaluation of technical debt was largely based on the developers' experience. A great amount of effort is required to reach a certain level of estimation accuracy even for experienced developers, who have to take into consideration many factors such as the impact from other changes, the future change of the debt-infected software artifacts and the change likelihood. When the developer lacks experience, the evaluation will be subject to large estimation errors. This remains true for any software project, so this area requires a breakthrough in research results.

Our results show that the overall cost of technical debt management is small compared to the release planning cost. However, we believe it is still worthwhile to improve the performance of technical debt identification and measurement because all types of management costs are overhead expenses, with an only indirect effect on the value of the software being developed. Thus advances in these areas will benefit overall efficiency for all software projects, not just those that have more significant technical debt management overhead.

We didn't observe any unusual characteristics that distinguish our study setting as highly unusual, other than the basic project characteristics that we have reported, nor in the conduct of the project during the case study. Thus we can, to some extent, be assured of the generalizability of our results to similar contexts.

The change in the total cost over sprints shows there was a significant initial overhead for technical debt management, with a much reduced cost over subsequent sprints. Because of the initial overhead, technical debt management effort in the first sprint reached 3.7 person-hours, a 70% increase in sprint planning effort, after incorporating technical debt management. In the subsequent sprints, the average increase dropped to 10%, which has a very minor impact on the project in terms of management cost. Even in the first sprint, the cost of managing technical debt was only about 10% of the cost to implement an average use case. Therefore, the amount of extra project cost added by adopting technical debt management could be considered reasonable.

Release planning in practice often follows a prescribed process with factors and criteria. When technical debt management was incorporated into the process, it

became a decision factor and thus affected the process and the final decision. Through this study we identified a set of decision factors and revealed their relations. For example, customer requests and availability of development resources are given higher priority than the amount of principal or interest associated with a technical debt item. Other factors include the current status of the debt-infected modules and the impact of the debt on other features. These factors and priorities helped us understand how technical debt management works in release planning and where the possible benefits of technical debt management may come from.

Although this study centers on investigating the costs of managing technical debt, we also gained insights into the benefits of explicit technical debt management. The main benefit of technical debt management is the increased awareness of the problems that are as important as those raised by the customer, but may be overlooked otherwise. Thus these problems could be identified and handled earlier to avoid turning into bigger problems in the future. The technical debt list also facilitates comparing different technical debt items, thus decision making. Because of these benefits, the project leader in our subject project decided to continue to explicitly manage technical debt, even after the case study concluded. These benefit insights gained from this study were combined with results and findings from other studies in this research to address the questions for this dissertation research (elaborated in Chapter 8).

6.6.2 Limitations and Validity Threats

The main limitation of this study is actually the limitation of any case study. The generalization power or external validity [157], of the study is restricted by the case we selected. In this study we selected a relatively small and young software project, which may have a different cost pattern and management style than bigger mature projects. For example, the proportion of the communication cost in a big project is usually higher than a small project. Having more people involved in meetings in which technical debt is being discussed would make those discussions more expensive. Managing the technical debt list might also involve more people, and thus be more expensive. On the other hand, a more mature organization might very well have more mature cost estimation procedures, which would make technical debt management more effective. In other words, the findings from this study reflect experience in just one particular case and hence may not apply to other contexts (e.g. larger more mature projects), but the general lessons learned should be instructive to those applying and studying this approach in any situation.

As mentioned in Section 6.1, the study was carried out on the project site, which is located outside of U.S., where the principal researchers were located. The physical distance and language barrier prevented the principal researcher from having direct contact with the project team. All qualitative data had to come through and be translated by our company contact. Therefore, there might be some subtle points that were lost in translation. This is a threat to internal validity [157], as it affects our ability to accurately explain the phenomena that we observed. The language barrier is also a minor threat to construct validity, in that we cannot be sure that the study

participants completely understood technical debt and our proposed management process in the way that we intended.

Another threat to internal validity (i.e. the extent to which we can claim to be accurately characterizing the effects of technical debt management) comes from the inaccuracy of technical debt estimation. Since the value of a technical debt item (i.e. its principal and interest) determines its payoff priority, inaccurate estimation may lead to a wrong decision. In this study there were technical debt items whose estimated values deviated far from their actual values. In particular, in two cases, the estimated principal was more than three times the actual effort expended when the technical debt item was actually paid off. The members of the development team attributed these deviations to a lack of experience with effort estimation. The decisions related to these items would also have been different if the estimated values had been closer to their actual values. Although this difference didn't affect addressing the research questions, more benefits might have been observed if they had used a more effective effort estimation approach.

To overcome the major limitation of this study, the subject project we selected for another case study (elaborated in Chapter 7) is bigger and more mature. It has a very different profile in terms of project cost, development cycle, etc. Combining the results from the two case studies not only yielded valuable findings from which more solid conclusions can be drawn, but also helped us gain insights into obstacles to the application of technical debt management in practice, which is presented in Chapter 7 and discussed in Chapter 8.

Chapter 7: The EducationHub Study

The study presented in this chapter is the second case study focusing on the cost side of technical debt management. In particular, this study was originally designed to address the following research questions:

(1) What are the costs of explicitly managing technical debt using the proposed approach?

(2) How does technical debt information contribute to decision-making?

In this study, we applied our proposed technical debt management approach to an ongoing software project, then collected and analyzed data about various costs incurred in the course of technical debt management. Other information such as the technical debt items identified from the project and the decisions on carrying or paying off the technical debt, were also collected to help provide context.

As the study proceeded, especially in the data collection phase, we found that the project team we were studying encountered obstacles to applying the proposed approach to their project and the actual process deviated from what our approach prescribed. This was not surprising, as the proposed approach was designed to be simple and basic, allowing for future refinement as it was evaluated in practice.

However, to some extent, this hindered us from fully addressing the original research questions above, especially the role of technical debt management in decision making (RQ2). We were still able to collect and analyze some data addressing costs, and we interpret our results along with results from the Tranship study (presented in Chapter 6). Therefore, we decided in the end to keep the first research question, but

drop the second one regarding how technical debt information contributes to decision-making (the numbering of the research questions hereafter is consistent with this decision). On the other hand, the process deviation we observed also raised new and interesting questions on managing technical debt. We believe that revealing the causes behind the obstacles to applying explicit technical debt management in a software project could smooth technology transfer from academic research to industrial practice and thus improve adoption of technical debt management in industry. Moreover, the exploratory nature of this study design allowed us to direct the study to new targets. Therefore, we added the following research questions to the study:

(2) What are the obstacles to explicit technical debt management?

(3) What contributes to the deviation of the proposed technical debt management process?

The opportunity to address these two questions allowed us to expand our findings to further facilitate future efforts at explicit technical debt management. This study has been published as a peer-reviewed article in an academic journal [158].

7.1 Subject Project

The subject project was selected by convenience by our industry collaborator in Brazil. It was a software application for supporting teachers and students, running on tablet computers on the Android operating system, as described in Section 3.2.2.4.

At the beginning of the study, the development team consisted of 1 project manager, 1 project leader and 7 developers. This remained fairly constant, although the person

occupying the role of project leader changed once, and some developers came and went from one sprint to the next. The timeline of important project and case study dates is shown in Figure 19. The project began on Dec 20, 2011. The case study began in the middle of February, 2012, with collection of basic project information. Afterwards, the project team started to identify technical debt items. For various reasons, the technical debt identification phase lasted until the middle of November 2012. The project team started tracking technical debt on March 20, 2013. We observed and collected data on the technical debt management process until the end of June, 2013. The case study ended in September, 2013 with a follow-up interview.



Figure 19. Important Project and Case Study Dates

7.2 Study Process

From the beginning of the study until the end of technical debt identification, the research team, which included the faculty advisor, the student investigator and a third researcher, directly communicated with the Developer contact via email and teleconference (three persons were involved due to project personnel changes). When

the project team started tracking technical debt for release planning, the faculty advisor from the research team was working on-site with the project team. So she was in charge of observing and supervising the project team in performing technical debt management. Then the student investigator conducted a follow-up interview with the project leader at the end of the study. The third researcher was in charge of facilitating participation of the subject project in the study, helped with communication, and overall supervision of the study. The project personnel were all native Portuguese speakers, with varying levels of English proficiency. All communication between the student investigator and the company contact was in English. The majority of the communication between the faculty advisor and the project team was also in English, with some Portuguese as necessary. The third researcher communicated with the project team primarily in Portuguese. All communication within the research team was in English.

We followed the general process described in Section 3.2.2.1 to carry out this study, but we present here the steps with more details and particulars to this study.

- (0) Preparation: the company contact collected basic project information such as project description, requirements and project schedule. We designed a simple spreadsheet as the technical debt list to ease management of technical debt items and to collect data on its use. The list was kept under version control, along with other project documents.
- (1) The project team were trained on how to manage technical debt using the proposed approach and how to document and report information required by this study. Training materials were prepared by the two researchers in

English, and then was translated to Portuguese and presented by the Developer contact.

- (2) The project team identified technical debt items in the subject system to prepare the initial technical debt list. This effort involved surveys of the development team, application of some source code analysis tools, and examination of the existing project backlog.
- (3) In the 13th release cycle, during which the project was tracking technical debt for the first time, the research member working on-site attended the sprint planning meeting, observed and took notes on the time spent by the project team on managing technical debt, the project members involved, and the decisions they made in the course of analyzing technical debt. After the meeting, the project team sent an updated technical debt list to the researchers.
- (4) After each of the planning meetings for sprints 14 and 15, we followed up with the project leader and other team members to collect data on the meeting content related to technical debt management.
 - a) We asked for information on the process followed during the planning meeting, as it deviated from the proposed technical debt management process.
 - b) We elicited the reasons for the deviation.
 - c) We interviewed the project leader after the completion of the 15th sprint, which was not only the last sprint for our study, but also the last one of the project (the project was cancelled soon after).

7.3 Data Collection

The first type of data we collected was project documentation, which included a general project description, project requirements and high-level design, and project schedule. This type of data was collected at the beginning of the study.

The second type of data was regarding the existing technical debt instances in the subject project. It was collected in the form of a technical debt list, as shown in Figure 20. The technical debt list contained the description, responsible persons, location, principal, interest amount, and interest probability. The items were grouped by their current status (all the items that appear in Figure 20 are “currently active” items). Most of the technical debt items identified from the project are design debt, which is defined as the effect of shortcutting programming practices that result in low quality, especially low maintainability, source code. A core part of our technical debt definition (section 1.1) is that technical debt has a short-term benefit and a long-term cost. For design debt, the short-term benefit is reduced programming effort resulted from shortcutting practices, while the long term cost is the potential increase of the maintenance effort due to the low quality source code. For example, one of the technical debt items was that there were multiple independent classes in the project with very similar functionality. They should have been designed to be extended from a generic parent class. Although having all the similar independent classes might have saved time in the design phase (short term benefit), maintaining multiple independent classes with similar behavior requires more effort than using a class hierarchy in the maintenance phase (long term cost) when the functionality needs to be changed. Over the course of this study, we collected 6 versions of the technical debt list.

ID	Responsible	Type	Location	Description	Principal	Interest Amount	Interest Probability
Currently Active Items							
10	Developer	Design	All of ListViews Adapters used in the project. All classes that extend ThemedArrayAdapter	We could use a parent class for generic adapters. In the current project there are several classes with similar behavior.	Medium	Medium	Low
15	Tech Leader	Documentation	Architecture document	Better documentation of the internal architecture of EducationHub.	Medium	Low	Low
19	Developer	Design	Widget, WidgetsController	Refactor the creation and control of widgets.	High	Medium	High
20	Designer	Design	All project screens	The interface design is ugly and poorly designed. Serious usability problems.	High	High	Low
6	Developer	Design	Ui.messages	Refactor the part wondering messaging to improve code readability.	Medium	Medium	Medium

Figure 20. Technical Debt List Extract

Field notes are the largest category of data we collected. They contain the project team’s activities regarding technical debt management, e.g. estimating principal and interest, time spent on these activities, decisions made on the technical debt items and other situations related to the study such as project members’ opinions, obstacles they encountered and deviations we observed.

Interviews were another type of data we collected. We conducted two interviews. The first one was conducted by one research member, who once worked on-site, in the middle of the study with the project leader. This interview was semi-structured and followed an interview guide that included questions about the details of the previous sprint planning meeting, activities related to technical debt management outside the sprint planning meeting, and in general how technical debt management was going. This interview was brief (about 20 minutes) and was not recorded, but was immediately transcribed into English by the research member. Afterwards we expanded the interview guide with follow-up questions and sent it to the project leader via email (this version is in Appendix A), and the project leader responded via email. The focus of that interview was how the decisions on paying off technical debt

were made (given that the proposed technical debt management process was not yet followed). The second interview was conducted with the same person, i.e. the project leader, at the end of the study (it is in the Appendix B). All of the questions and the response are in English. The questions were sent out and answered via email. It covered the interviewee's profile, the project's original technical debt management practices, obstacles of incorporating the proposed technical debt management approach and his opinion on the usefulness of the approach. The main goal of this interview was to identify obstacles and find out how to improve the adoption of technical debt management.

7.4 Data Analysis

Since most of the data we collected are qualitative in nature, e.g. open-ended questions, correspondence and field notes, coding was the primary approach for data analysis. The initial codes and categories were pre-formed based on the research questions. There are four main categories: cost, benefit, obstacle and process deviation, as shown in Table 18. Cost refers to the costs incurred in the project that can be attributed to explicit technical debt management (Research Question 1). Benefit refers to benefits gained from explicit technical debt management, including the benefits observed in the project and the benefits perceived by the project team. Benefits of technical debt management were not a focus of this study or its research questions, but they are relevant data of this dissertation research. Therefore, the benefits were collected and so were coded. In this study we observed that the project team encountered obstacles to using the proposed technical debt management approach. Therefore, one of the main codes is obstacle, which captures the factors

that prevent the project from using explicit technical debt management (Research Question 2). Another code is process deviation, which refers to the differences between the proposed technical debt management approach and the actual technical debt management practice carried out by the project team (Research Question 3). Process deviations included deviations from the way we expected technical debt information to be used in decision making.

The data were then open coded using the categories defined in Table 18. In this step we identified and added new subcategories and codes from the data in vivo. For example, “initial overhead” was mentioned specifically by different subjects. So we used two specific codes, initial overhead and other costs, to code the cost category. Likewise, we created specific codes under the category “process deviation” to reflect what part of the process deviated, as we found specific references to these parts of the process in the data (“task prioritization” was primarily concerned with technical debt decision making). The data about obstacles was primarily concerned with three aspects – difficulty in technical debt identification, time pressure and process integration – and so these became the codes in this category. Since data about the benefits of applying the technical debt management approach were collected not only from observation of the implementation of the approach, but also from interviews with the project members, we found it useful to differentiate between observed benefit and perceived usefulness. The complete coding scheme is presented in Table 18.

Category	Code
Cost	Initial Overhead
	Other Costs
Benefit	Observed Benefit
	Perceived Usefulness
Obstacle	Difficulty in Technical Debt Identification
	Time Pressure
	Process Integration
Process Deviation	Technical Debt Estimation
	Task Prioritization

Table 18. Coding Scheme

The next step of the coding, axial coding, was to re-assemble these codes and categories and identify the relationships between them. Figure 21 represents the results from axial coding. By analyzing the data, we found out there are different types of obstacles to explicit technical debt management, one of which is cost-related. The perceived usefulness (or lack thereof) of explicit technical debt management also hinders the application of the technical debt management approach. These obstacles eventually caused the process deviation. The detailed results are presented in the next section.

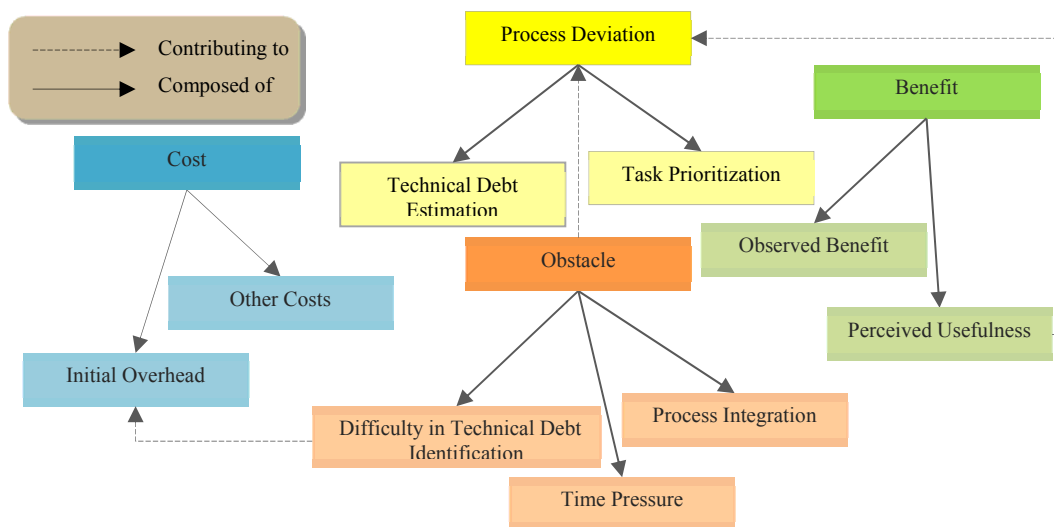


Figure 21. Axial Coding

7.5 Results and Findings

Through data analysis, we identified three major themes regarding technical debt management – costs of technical debt management, obstacles to applying explicit technical debt management to the project, and deviation of the actual technical debt management process from the proposed one. These three themes easily map to Research Questions 1, 2, and 3, respectively, and are discussed from that perspective in section 7.6. By digging into the details of the data, we found various aspects within each theme. The axial coding of the data further revealed the relationships between the themes and the aspects, as shown in Figure 21. The themes, aspects and relationships together helped us understand the rationales behind the technical debt management practice and hence address the research questions of this study.

7.5.1 Costs of Managing Technical Debt

There are different costs tied to technical debt management. We identified these cost categories mainly through observation of how the project team dealt with the technical debt in the subject project. In particular, the field notes taken around the first sprint planning meeting record how many persons spent how much time in discussing technical debt related issues, estimating the values of technical debt items and updating the technical debt list before, during and after the meeting. The field notes taken around the subsequent sprint planning meetings also contain cost information, but it was not as detailed and exhaustive as the first one. In addition, the project leader, through an interview, provided some cost information such as the participants and the time they spent on technical debt management. Time and effort

associated with the initial phase of technical debt identification, described below, was communicated to the research team via email from our Developer organization contact.

Technical debt management started with identification of the technical debt instances and initial construction of the technical debt list. Technical debt identification began in the middle of March, 2012 and continued, on and off, until November 2012 (see Figure 19). This included periods in which the project was without a permanent project lead, or was suspended for business reasons, or the personnel were reassigned to other projects temporarily, and thus they did not engage in technical debt identification. Technical debt identification actually took about 3 months of activity, but was spread over an 8-month period. Project members reported technical debt items in an ad hoc manner at first, but were aided by several other techniques. We analyzed the bug repository and change log of the project to help narrow down the areas of the source code where it was more likely to have technical debt than other areas. We also conducted a survey among the project members to elicit technical debt information from their past experience in a more systematic way.

In terms of time span and involved effort, technical debt identification was the most costly step in the technical debt management process. One reason is that the concept of technical debt involved a *learning curve* for the project team. The learning curve was evident when the first set of technical debt items were identified and reported to the research team, in April 2012. For example, one technical debt item they identified was regarding the lack of a feature. This was actually a missing

requirement, not technical debt. Another example is that they reported a defect as a technical debt item. Based on our definition of technical debt as the effect of immature artifacts, delaying defect fixing could have effects on other development and maintenance activities, and these effects could be considered technical debt, but a defect in and of itself is not technical debt. From these and other technical debt items in the initial technical debt list, it appeared that the developers had difficulty in associating the abstract concept of technical debt to concrete forms of technical debt in reality. It took more than three months for the project team to finish identifying all technical debt items in the project. This learning curve was confirmed by the project leader in the interview we conducted with him. When we asked him about the obstacles to technical debt management in their project, he thought that “it was not always very clear what was a technical debt item and what was not. What I saw as a technical debt item, sometimes the team thought it was not, and vice versa. Also sometimes I had seen a technical debt as having been paid, but they thought it was not.” Resolving these different interpretations, both within the project team as well as with the research team, took considerable time.

Besides the learning curve, another reason is that **identifying technical debt is still difficult**. Although various technical debt identification approaches have been proposed or implemented for software development practitioners, most of the approaches still need to be adapted to be effective for a project in a particular domain [159]. During technical debt identification, our research team took various actions, such as analyzing change logs of the project and running a survey, to help the project team identify technical debt, but it still took more than three months for them to finish

this step. When we asked the project leader what the most difficult/time-consuming part of the technical debt management approach was, he gave a very clear answer – identifying and recording technical debt. Because technical debt identification is the first step of technical debt management, from the cost perspective we consider this difficult and time-consuming step a big initial overhead for technical debt management.

Once the technical debt items were identified, the project team constructed a technical debt list with information such as description, identification date, principal and interest. Then the project team used the technical debt list for sprint planning. The costs incurred in this process can be categorized into “updating technical debt list”, “sprint planning” and “communication”. “*Updating technical debt list*” represents the time spent in understanding technical debt items in the list and determining their current status. In Sprint 13, this occurred during the sprint planning meeting, but in other sprints it occurred outside of the sprint planning meeting. “*Sprint planning*” represents time during sprint planning meetings spent on using the technical debt list to plan a sprint, primarily deciding what to include in the next sprint.

“*Communication*” captures other time, outside the sprint planning meeting, for discussing strategies to manage technical debt among the project team members.

As shown in Table 19, updating the technical debt list in Sprint 13 took much more time and involved many more people (highlighted with red color) than in the other two sprints. This was mainly because the technical debt list was used for the first time

and the project team had to go over the entire list more than once to determine the status of each technical debt item.

There were three sprint planning meetings that we tracked for this study. They included all project team members and ranged from 90 to 375 minutes, but the technical debt information was barely used for sprint planning (3% or less of the total meeting time). This was where the major process deviation occurred, which we will discuss in more detail in the next section. Communication among project members regarding technical debt management happened occasionally outside the sprint planning meeting and involved 2-3 persons. Among these types of costs, **updating the technical debt list for the first time accounted for the majority of the technical debt management cost.** It could be viewed as part of the initial overhead we mentioned earlier. **The other types of costs were relatively small and hence didn't have significant impact on the project,** in comparison with the initial overhead. Therefore the initial overhead was the main cost category of technical debt management.

	Sprint 13		Sprint 14		Sprint 15	
	O/M	I/M {11, 130} ¹	O/M	I/M {8, 375}	O/M	I/M {N/A, 90}
Updating Technical Debt List	{1, 30}	35 minutes	{2, 10}	0 minutes	{0, 0}	17 minutes
Sprint Planning (percentage)²	{2, 5}	2 minutes (2%)	{2, 5}	0 minutes (0%)	{0, 0}	3 minutes (3%)
Communication	{2, 15}		{3, 10}		{2, 10}	

O/M: outside of meeting; I/M: during meeting

Table 19. Costs of Technical Debt Management

¹ The pair in the braces represents the number of involved persons and the minutes of time spent accordingly.

² The percentage of sprint planning time that was spent in managing technical debt.

7.5.2 Process Deviation

The original goal of this study was to explore the costs of explicit technical debt management. Therefore, the study design required that the project team apply the proposed technical debt management approach to the subject project, allowing us to track related costs and activities. However, the project team didn't strictly follow the proposed technical debt management approach, which gave us the opportunity to observe deviations as the project proceeded.

According to the proposed approach, the principal and interest of the items on the technical debt list should be estimated and the prioritization decision, i.e. which items have higher priority than others for repayment, should be based on cost/benefit analysis of these items. The project team, by contrast, didn't always do estimation and when they did, the estimates were not used for their decisions. For example, when the first sprint planning meeting was held the principal and interest of the items on the technical debt list hadn't been estimated. They didn't do any estimation in the meeting, but a decision was made not to pay any technical debt items in next sprint. This may be because many of the technical debt items had already been paid off before the first sprint (during the project suspension), but time constraint could be another reason because they had a long list of features that needed to be implemented in the next sprint. A follow-up interview after the meeting confirmed this. The decision to not pay off any technical debt items in this sprint was actually made before the sprint planning meeting for the same reason. The project team didn't even have enough time to fulfill the requirements. Therefore, they would not pay off any technical debt items in this sprint unless it overlapped with the requirements.

Instead of estimating and using principal and interest in decisions about paying off technical debt, other criteria were used to drive these decisions. For example, after the second sprint planning meeting, a senior developer and the project leader discussed implementing more test automation in the next sprint, which was already on the technical debt list. They also considered restructuring the system architecture. So they updated the technical debt item about test automation and added restructuring the architecture as a new item to technical debt list with coarse estimates of its principal and interest. Then they negotiated with the Client to include test automation in the next sprint. For test automation, they did not do numeric estimation for the principal and interest as our approach requires. A follow-up interview with the team leader afterwards revealed that the decision to implement more automated tests had to do with the difficulty (the “pain”) of testing, but had more to do with improving the quality of testing. In other words, at least in this case, the priority of a technical debt item has more to do with quality than cost. While test efficiency might have been fairly straightforward to represent as a type of “interest”, the team found it hard to do this with test quality. Thus, our approach of using principal and interest fell by the wayside in making this decision. Similarly, there was no discussion of principal or interest on any technical debt item during the third sprint planning meeting. The team leader reviewed the entire technical debt list, added some new items, and made some decisions about paying off items in the meeting, but he didn’t follow our approach to prioritize the items and make decisions accordingly. The decision to pay off one item, which was proposed by a developer, was made based on the availability of the development resource, not economic concerns.

In summary, the proposed technical debt management process was not strictly followed by the project team, which was not completely unexpected. The deviation was primarily in the form of the use of criteria other than a quantitative cost-benefit analysis involving principal and interest. Rather than estimating principal and interest on individual technical debt items, and then using those estimates to do a cost-benefit analysis for each one, the team chose technical debt items based on such criteria as current pain points in the project and availability of specific resources. In other words, information about existing technical debt (location, principal, interest, etc.) was barely used for decision making in release planning.

It was mentioned several times and on different occasions that the compromises the team took in sprint planning, including their approach to technical debt management, was a result of lacking time. To confirm this point as well as identify other causes of process deviation, we looked into the obstacles that the project team had in managing technical debt, which are presented in the next subsection.

7.5.3 Obstacles

In this study the project team encountered several obstacles to managing technical debt using the proposed approach. As we saw in Section 7.5.1, the difficulty of technical debt identification was the first major obstacle for the project team. The project leader confirmed this in an interview, explaining that technical debt still seemed to be an abstract concept although he thought the training was sufficient. He suggested that “Maybe there should be harder rules for what should be classified as a technical debt and how to identify it.” The difficulty surrounding technical debt

identification not only created a big obstacle for the project team, but also incurred high cost at the beginning of their technical debt management experience. Meanwhile the project team had low perceived usefulness of technical debt management to their project, as indicated in an interview with the project leader. When we asked the project leader about his general impression on the technical debt management approach, his answer was “it’s not very impressive.” The difficulty and the low expectations of usefulness hindered application of technical debt management in their project. Three project members, including the leader, were asked about the usefulness of the proposed technical debt management approach. They all thought that the approach was generally useful, but the leader stressed that it’s still hard to apply, especially when facing time pressure, such as “delivering many features in a short time frame”. This leads to the second major obstacle – time pressure.

According to the project leader, requirements/features were always given the highest priority in their sprint planning. In the situation where the time was barely sufficient to deliver requirements, “other processes tend to be completely ignored” and “technical debt management is very vulnerable as a candidate to be cast away”. Such situations did occur in this study when the second sprint planning meeting was about to start. The project leader described the situation as a “very long” meeting with “a lot of complicated features.” Therefore they did not spend time going through the technical debt list or discussing any technical debt items during the sprint planning meeting. Under time pressure, it was natural for the project team to consider technical debt management an optional process on top of, rather than incorporated into, their original sprint planning process. It also then makes sense that the team would revert

back to their original process for prioritizing tasks because the original sprint planning process ensures the requirements gain highest priority and are always treated first even if they do not have enough time. When the project leader was asked about the impact of technical debt management on the subject project, he explained this tendency in another way: “we had not used the recorded values to make the decisions and we usually paid the debts when the demand was low.”

Because we have realized that our technical debt management approach was not strictly followed by the project team, we framed some interview questions for the project leader as to what changes to the management approach would have made it easier for the project team to adopt. The answer by the project leader brought up another major obstacle – lack of process integration. The project leader suggested that the integration of our approach with their project management tool would have helped them in using the approach.

Actually the sprint planning process that the team was already following was very similar to our technical debt management approach – walking through tasks, estimating effort, prioritizing them accordingly, deferring some low priority tasks if the required effort is more than what is available for the next sprint. Therefore, our approach could have been organically integrated into their sprint planning process although it was not presented that way to the team originally. So the project team treated technical debt management as an addition onto their original process. It became another, separate task, with a separate document (the technical debt list) to deal with. They started trying to fit in discussion of technical debt items only after

they finished all the items in their backlog, during the sprint planning meetings. For example, during the first sprint planning meeting, the project team first spent 1.5 hours discussing each task in the backlog and assigning priorities. Then they moved to technical debt management and spent another 35 minutes on discussion of updating the technical debt list, which might have been saved, at least partially, if technical debt management had been well integrated into their sprint planning process. The same approach was taken during the second sprint planning meeting, but because there was not enough time left over, technical debt management was only briefly covered at the very end of the meeting and no other actions or decisions on technical debt were made. In an interview with the team leader after the meeting, he stressed that they didn't have enough time to follow our approach to manage technical debt before and during the meeting. This indicates that technical debt management was very much viewed as an add-on, not an integrated part of the sprint planning process.

In summary, there were three major obstacles in this study for the project team. The first obstacle, i.e. the difficulty of technical debt identification, together with the low perceived usefulness by the project team, hindered motivation to apply the proposed technical debt management approach and thus caused the actual technical debt management process to deviate. Time pressure was an external factor that resulted in the intended prioritization mechanism of our approach (i.e. principal and interest) to be abandoned under some circumstances. Lack of process integration, another obstacle, also aggravated the process deviation.

7.6 Discussion

The original goal of this study was to uncover the costs of explicit technical debt management. In the process of implementation of this study, we observed that the project team encountered some obstacles and the actual process of technical debt management they followed was different from the one we proposed. This deviation provided us with the opportunity to investigate the obstacles to application of technical debt management in software projects and reveal the reasons behind the process deviation. In the end, we investigated three research questions, presented in Section 7.1. We discuss the findings of our study in light of these three research questions in this section, followed by a discussion of the validity of this work, and its limitations.

7.6.1 Costs and Obstacles

The costs we have identified largely fall in four categories: cost of technical debt identification, cost of updating the technical debt list, cost of using the technical debt list for sprint planning, and cost of communication. Among these costs, technical debt identification accounted for the majority of the total technical debt management cost. The cost to update the technical debt list for the first time was also high, but it dropped significantly in the subsequent sprints. Because technical debt identification is the first step of technical debt management, and constructing and updating a technical debt list immediately follows technical debt identification, the high cost incurred by these first two steps created a large, almost prohibitive, overhead for technical debt management, as illustrated in Figure 22. Other costs of technical debt management are insignificant compared to this initial overhead. It should be noted

though that the low effort of updating the technical debt list in Sprint 14 and 15 compared to Sprint 13 may result from the project leader’s decision of not managing technical debt for sprint planning in the two release cycles. Therefore, the cost pattern may have been different, or even the cost may have not formed a pattern, if the technical debt management had been practiced during the two Sprints.

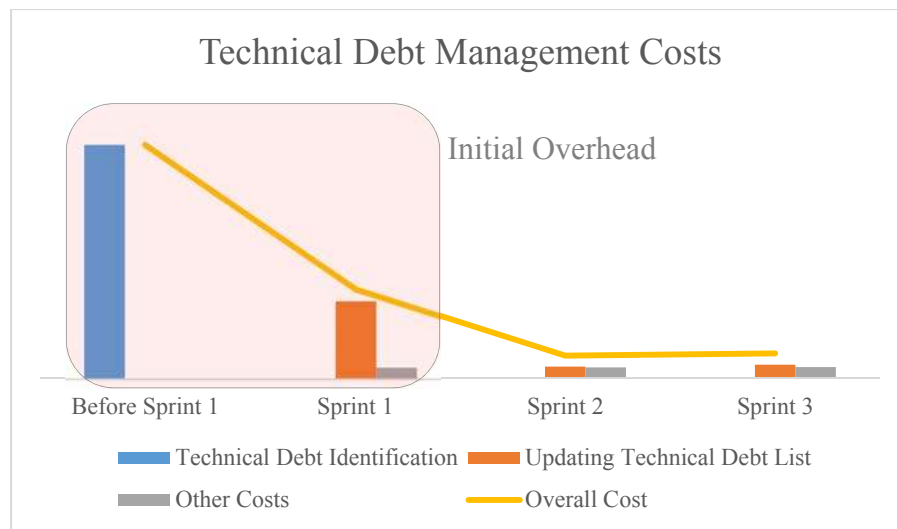


Figure 22. Cost Pattern of technical debt Management

However, we tend to believe this cost pattern is real because we found a similar one (high initial overhead and subsequent minimal costs) in the Tranship study (presented in Chapter 6). Because the actual technical debt management process deviated from our proposed approach in this study, the cost pattern we identified may be different from a case in which the proposed approach is strictly followed. But given the consistency of this finding with that of the Tranship study, and the fact that this finding is further elaborated and explained through interview data, we are confident that the application of technical debt management has a significant initial overhead.

In this study the actual technical debt management process followed by the project team deviated from the one we proposed. The main deviation lay in the criteria used to decide if an instance of technical debt would be paid off in the next sprint. Rather than using technical debt principal and interest, in a cost-benefit analysis, to make this decision (as proposed in the technical debt management approach), the project team used a variety of other criteria, relying on their previous sprint planning process. In theory, any decision-making criteria could have been incorporated into the process of estimating principal and interest. For example, the availability of specific personnel is often considered in these decisions. If the personnel best suited to paying off a technical debt instance are not available, then this could be incorporated into that instance's estimate of principal, making the principal high enough that it would not be considered cost-effective to pay off. Similarly, if a technical debt instance is particularly important to a customer, that instance's interest would increase, reflecting the future cost (in terms of good customer relations) of not paying off the debt. However, incorporating such decision criteria into the notions of principal and interest is not intuitive, so the project team in this study preferred to use criteria other than principal and interest in their decision making.

The deviation resulted from the significant obstacles that the project team encountered in the course of managing technical debt in their software project. Among the obstacles, those associated with technical debt identification, including the learning curve of understanding technical debt and the technical difficulty of technical debt identification, were major. In the entire period of the study, the project team always faced the challenge of implementing a heavy load of requirements in an

“unreasonably” short time frame. As a result, some tasks had to be postponed or even cancelled if the tasks were considered non-critical. Because the technical debt management approach was not organically integrated to the project release planning process (which we learned from the interview with the project in the end of this study) and thus was only treated as an optional task, it’s natural that the proposed technical debt management approach became a candidate to be abandoned. Therefore, time pressure and lack of process integration were obstacles to application of technical debt management in the project.

7.6.2 Evaluation of Validity

The actual implementation deviated from the original study design in that the proposed technical debt management approach was not followed in their sprint planning process. This is a threat to the internal validity of this study as there might have been other types of costs that were not captured due to the process deviation. In addition, the cost pattern we identified, i.e. high cost of updating technical debt list for the first time and insignificance of other types of costs for technical debt management, may not accurately represent the cost pattern for other projects that more fully used technical debt information in the sprint planning process. Therefore, the reliability of our findings related to Research Question 1 is weak.

Another threat to both the internal and construct validity came from the shifted focus of this study. The original study design was to explore the costs of explicit technical debt management. Therefore the study instruments, including the observation/documentation guidelines and the interview questionnaire, didn’t focus

on obstacles and process deviation until the second sprint that we tracked. This means some details about the obstacles and process deviation that happened in the early stage of this study may be missed or captured by indirect means. The threat is also reflected in the data volume and level of detail, as most of the information regarding obstacles and process deviations was collected in the later stage of the study.

However, because it was collected during the critical phase, i.e. sprint planning, we do not believe this has a significant impact on the validity of this study, especially on the conclusion regarding the obstacles and how they contributed to the process deviations.

There is a threat to reliability in that the data collection techniques were not as robust as they could have been. This is primarily due to the remote location of the researchers during some phases of the study and the mix of languages used in conducting the research. The cultural difference between some of the researchers and the project team may have had an effect as well. These factors may have led to some misunderstandings or to missing some context factors that might have been relevant. However, we believe these effects are at least partially mitigated by the fact that two research members were on-site during most of the data collection (one of them was on-site for the entire study), and there was a mix of cultural backgrounds represented in the research team.

7.6.3 Limitations

To overcome the limitations of the Tranship study, the subject project used for this study has a different profile. The software project described in this paper was

developed in an organization belonging to a multinational company. It was a larger and more mature software project compared to the one used in the previous study. We expected that this study could complement the previous one and improve the generalizability of the conclusions drawn from them. However, the limitation of generalizability couldn't be fully addressed by this single case. From the perspective of case selection, the project we used for this study was not the best choice as the project was not obviously feeling the "pain" of technical debt. The project personnel might have been more motivated, and thus offered less resistance, if the project had had some obvious technical debt issues and wanted very much a solution to the technical debt problem. This implies that our results, in particular the impact of the obstacles we encountered, might not apply in contexts where the need for technical debt management is recognized by all stakeholders.

The case used for this study was selected based on convenience and availability. Due to business reasons, the subject project was interrupted several times during the study, and was ultimately canceled right at the end of our study. As a result, we were not able to conduct a second-round follow-up interview with the project leader or other personnel. Thus, we were limited in the depth of insight we could elicit about the project's experience with technical debt management. This raises the possibility that we are misunderstanding some of the data that we have collected. We have attempted to mitigate this risk through careful analysis, direct observation where possible, and peer checking within and outside our research team.

Chapter 8: Discussion

This dissertation research started with exploring the nature of the problem that the technical debt metaphor signifies. By investigating the phenomenon in software development practice and reviewing academic work in related areas, we have learned that the essence of technical debt lies in the two sides – short term benefit with long term cost – and the uncertainty of interest payment. Technical debt could save effort in the short run for a software project if it is properly leveraged, but a software project may end up with higher maintenance cost in the long run because of the technical debt it carries. Technical debt is a common problem in the software industry and the impact is often serious enough to concern software project managers. Moreover, the involved uncertainty further complicates the effect of technical debt on software projects. Therefore, software managers need to balance the benefit with the cost when they make decisions on when and what technical debt should be incurred or paid off, which is the central problem of technical debt management.

Aside from the short term benefit and long term cost of technical debt, there are costs and benefits associated with explicitly managing technical debt, which may affect the overall cost of a software project. In other words, leveraging technical debt information for better decision making will incur management cost, but may bring benefit that could offset or even exceed the management cost, thus contributing to reduction of the project cost. To help software managers make informed decisions on technical debt management, we developed a general technical debt management framework and carried out this research to refine and validate it. This research

characterized the costs and benefits of explicit technical debt management. To be specific, the research addressed the following questions. (1) What factors contribute to the costs and benefits of measuring and monitoring technical debt? (2) How does technical debt information contribute to decision making in software project management?

To tackle the problem, we designed two types of studies, each of which focused on one side, i.e. benefit or cost, of technical debt management. The first type of study investigates the potential benefit gained from explicit technical debt management. The study design is retrospective and requires that the subject project contain technical debt in its past releases and that there were decisions made, either intentionally or unintentionally, to pay off the technical debt during the releases studied. Firstly, our proposed technical debt management approach is applied retroactively from the beginning of the selected releases to simulate the decisions that would have been made about what technical debt should be paid and when. Then the results from the decision simulation are compared with the actual decisions that were made to determine whether and how much cost could have been saved if the explicit technical debt management approach had been used. The two types of studies are all case studies per se, but given the nature of the first type of studies, i.e. decision simulation on past releases of a software project, we call it a “retrospective” study.

By contrast, the second type of study, which we call a case study, is targeted to future releases of a live software project. In these studies, the technical debt in the subject projects is explicitly managed in real time using our proposed approach. Then the

costs of explicit technical debt management are uncovered by tracking technical debt and the related management activities continuously over several releases of the subject projects. This type of study could also reveal how technical debt information is used in decision making and thus help us understand how to refine the proposed technical debt management approach to improve its adoption by software practitioners. Finally, with the results from the two types of studies, we can compare the benefits of explicit technical debt management with its costs to determine the effectiveness of explicit technical debt management and thus answer the research questions.

We have carried out two retrospective studies and two case studies. Because of the nature of these studies, we gathered a rich body of qualitative data regarding technical debt management, which helped us gain insights into the technical debt management problem and allowed us to go beyond the research questions to discuss the effects of technical debt on software maintenance from a wider perspective. In the following subsections we discuss the results from the two types of studies separately. Then we discuss the contributions of this line of research and future direction of technical debt management research by combining the results of these studies.

8.1 Retrospective Studies

The two retrospective studies we carried out differ in profile of both the subject project and the technical debt items. In the first study (SMB, elaborated in Chapter 4) we chose a software application provided by our industrial partner as the subject project, while the subject project for the second study (Hadoop, elaborated in

Chapter 5) came from the open-source software community. The subject project for the SMB study was much larger than the one for the Hadoop study in terms of total lines of code, although both of them had a long evolution history, as required by our study design. In the SMB study only one technical debt instance was used for decision simulation, but it was a huge debt item involving modification of the entire communication layer and other layers of the application, and hence had serious impact on the project. By contrast, the technical debt items used for the Hadoop study were relatively small and located in four classes that are independent of each other. In general, the SMB study had a large subject project with one outstanding technical debt instance, while the subject project used in the Hadoop study has a smaller profile with many small technical debt items spreading over it. We believe that these differences improved the generalizability of the conclusions drawn from the two studies.

For most of the technical debt items in the two studies, the simulated decisions are different from their actual counterparts, which indicates that the technical debt information in the projects was not fully taken advantage of to help the software managers reach the optimal decisions. The results of the two studies show that explicit technical debt management could bring significant benefit to software projects through optimizing the timing of incurring or paying off certain technical debt items. In both cases, the benefits were significantly large compared to the effort of constructing the artifacts that contained the technical debt items.

In the Hadoop case, most of the technical debt items were small issues and any individual item may not be serious enough to worry the project manager. Therefore, it is highly likely that the impact of technical debt on the project would be overlooked or underestimated when technical debt is managed in an implicit way, if at all. For this type of case, the main value of explicit technical debt management lies in the raised awareness of the impact of technical debt. Through explicit technical debt management, software managers could have a holistic view of the technical debt in their projects and thus avoid misunderstanding its impact. Among the four technical debt items analyzed in this study, two of them would have turned out differently. The modification effort that could have been saved is largely comparable to, or even bigger than, the effort of constructing the classes containing the items although this effort is not huge given the small size of the classes. Therefore, even if the benefit gained from an individual technical debt item is limited, managing all technical debt items collectively in an explicit way could still lead to big cost saving on the project level.

Compared to the technical debt items in the Hadoop study, the one used in the SMB case was much larger with respect to both principal and impact. It would take a great amount of effort to modify the communication layer of the SMB application, i.e. the principal of the technical debt. The persistence layer had to be changed, too, as it is coupled with the communication layer, which means the modification would have significant impact on other components of the application. Given its size and impact, this type of technical debt item can hardly be overlooked. Instead, it is most likely to be incurred intentionally. In the SMB case we can see that delaying upgrading of the

communication protocol was indeed a serious decision made by the project team. However, the actual decisions on this technical debt item did not lead to the best outcome, due to a lack of cost-benefit analysis as our proposed technical debt management approach suggests. Actually the extra cost of carrying the technical debt, i.e. delaying upgrading the communication protocol, was two times as high as the cost of upgrading and it could have been avoided if the simulated decisions had been followed. Therefore, in this type of situation, a cost benefit analysis based on measurement of technical debt, the core of our proposed approach, is essential for software managers to avoid big losses and reach optimal decisions.

Through the two studies, we have demonstrated that explicit technical debt management could bring significant benefit to software projects in the situations where there are either one technical debt item with great value or many independent debt items with limited impact. In reality, it is possible that a software project falls into both of the situations, that is, the project has many technical debt items and only a few of them have great impact. We believe our proposed approach is able to handle such cases because it provides a prioritization mechanism to determine the order of paying off technical debt items, in addition to the cost-benefit analysis of individual debt items.

8.2 Case Studies

The two case studies were hosted by two different organizations with the help from our industrial partners. They began almost at the same time. In the first case study (Tranship, elaborated in Chapter 6), we identified and tracked technical debt items in

a web system for enterprise management, while the subject project used in the second case study (EducationHub, elaborated in Chapter 7) was to develop a software application running on a mobile platform. The two projects are similar in the size of development team, but they are significantly different in project history. The EducationHub project had gone through more than ten sprints and had many versions when the study began, while the Tranship project largely started from scratch. In other words, the Tranship project was young compared to the EducationHub project. Both of the projects followed the Scum development process, but the development cycle, i.e. sprint, of the EducationHub project was much longer than that of the Tranship project, which was one reason that the EducationHub study ended later than the other one. The two projects were developed at two organizations with different profiles. The company hosting the Tranship project was a start-up software development and service provider, while SMB, where the EducationHub project was developed, is a large software development and research institute. The technical debt items used for the two studies largely conform to their organization profiles. In the Tranship study no individual technical debt item had significant value, hence impact, on the project, while some technical debt items, e.g. test automation, used for the EducationHub study involved huge effort to change and a great amount of time to make a decision for. These differences contributed to the generalizability of our conclusion on the technical debt management cost. They also provided the opportunity to compare the implementation processes of technical debt management under different organizations and thus identify factors affecting the adoption of technical debt management approach in the software industry. In fact, we indeed

gained insights in this aspect from the EducationHub study, where the actual technical debt management process significantly deviated from the one we proposed. By digging deeper in the results, we found out the reasons behind this deviation, which is another contribution of the study.

The results from the two studies show that managing technical debt could incur various types of costs. The cost identified through the Tranship study falls into 4 categories: identification of technical debt items, analysis and evaluation, communication and documentation, while in the EducationHub study the cost was categorized slightly differently due to the actual process of technical debt management implemented by the project. The cost categories used in the EducationHub study are technical debt identification, updating the technical debt list, sprint planning using technical debt, and communication. However, the difference in the cost categorization of the two studies can be easily reconciled by comparing similar categories in the two studies. According to the definition of the cost categories, analysis and evaluation cost in the Tranship study refers to the effort of understanding technical debt items and estimating their values, which subsumes the major part of the cost for updating the technical debt list, i.e. the effort of understanding technical debt and its status, defined in the EducationHub study. While the minor part of the cost, i.e. updating the documentation according to the technical debt analysis result, can be moved to documentation category defined in the Tranship Study. Since using the technical debt analysis result to make decisions is an added small step to the original release planning process and thus barely takes effort, we expand the analysis and evaluation category to include the sprint planning category

defined in the EducationHub study and name it “analysis” category. Figure 23 shows the mapping of the cost categories used in the two studies.

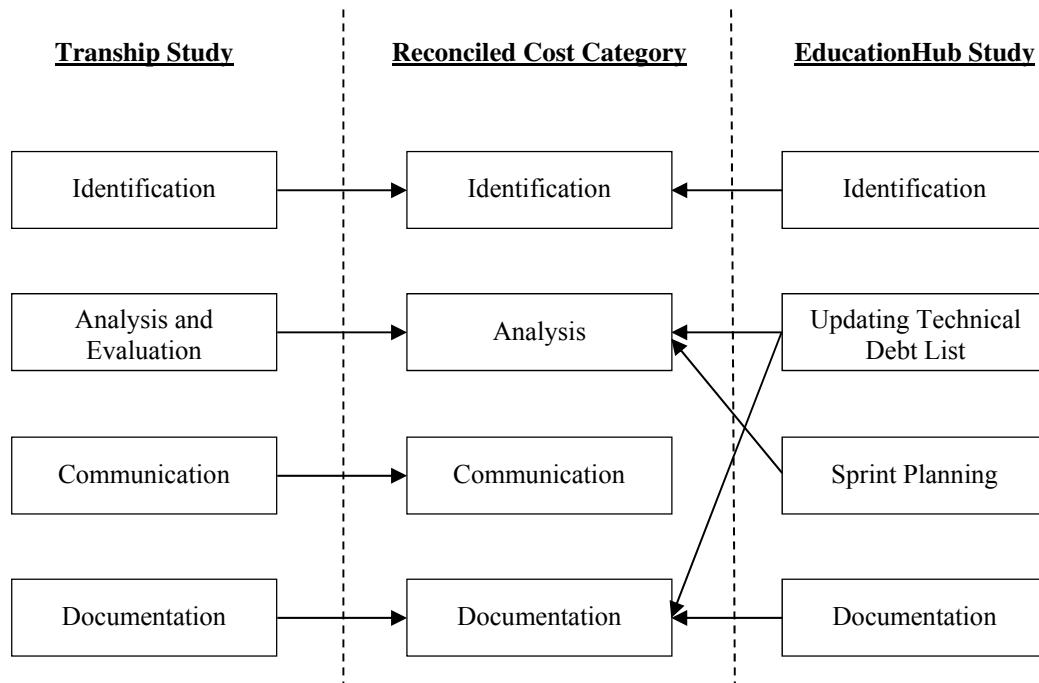


Figure 23. Mapping of Cost Categories

If we view these costs in chronological order, rather than by category, the first one is the initial technical debt identification cost. In the EducationHub study, the initial technical debt identification cost was huge and accounted for the majority of the technical debt management cost. Further analysis of the EducationHub data revealed the reasons behind it – the learning curve of technical debt management and the technical difficulty made technical debt identification the most time-consuming and costly part of technical debt management. Due to resource constraints, the initial technical debt identification in the Tranship study was not tracked and thus the cost information was not collected.

Coming after the initial technical debt identification was the initial analysis, which incurred high cost in both studies. The result from the Tranship study shows that understanding technical debt items and estimating their values accounted for the majority of technical debt management effort. In particular, the cost of technical debt analysis was high in the first sprint and dropped significantly in the subsequent sprints. We observed a similar trend from the EducationHub study in which technical debt analysis for the first time was the major part of the analysis cost because the technical debt list was used for the first time and the project team had to go over the entire list multiple times to determine their status and values.

Another technical debt management activity that could incur high cost is identification of new technical debt items after initial technical debt identification. In the Tranship study, identifying a new technical debt item took great effort, but we may not conclude that it incurs high cost in general, given that only one new technical debt item was identified in one of the three sprints. In the EducationHub study, identification of new technical debt items after initial technical debt identification did occur, but the cost could not be extracted from other efforts.

Compared to technical debt identification and analysis for the first time, other types of cost in the subsequent release cycles in both studies were insignificant. Thus, we can conclude that technical debt management is subject to a big initial overhead, which consists of initial technical debt identification and the initial analysis of the technical debt list in the first sprint in which technical debt is explicitly managed. This overhead has significant impact on the overall cost of technical debt

management. Under some conditions, such as that of the EducationHub study, the overhead may hinder the project from carrying out technical debt management. Therefore, reducing the overhead is essential to make technical debt management a cost-effective approach, and advances in technical debt identification and measurement are likely to improve the adoption of technical debt management.

Besides characterizing the cost of technical debt management, another goal of the case studies was to understand how technical debt information contributes to decision making in the process of software development, especially in release planning.

Through the Tranship study we have identified a set of factors that affect the decisions on release planning. Technical debt information was used by the project team in their release planning, but it was treated as one decision factor rather than a process that incorporates other decision factors for release planning, as our proposed approach suggests. The reason was that the project team preferred a simple approach to estimation of technical debt principal and interest while keeping all the factors in sight when they actually make the decision. In spite of the deviation of the actual technical debt management process from the one we proposed, we learned how technical debt is used for release planning in real settings through this study.

Moreover, the deviation helped us gain insights into how technical debt management should be integrated into a software development process in practice.

From the perspective of implementing a prescribed process, the EducationHub study went further off the course as we observed bigger deviation from our proposed technical debt management process than that in Tranship study. In Tranship study

most of the steps of the actual technical debt management process were actually same as their counterparts of the proposed process. It's just the timing of consolidating all decision factors that differentiate the two processes (when estimating technical debt values in the proposed process versus when the final decision is made in the last step of the actual process in Tranship study). In EducationHub study technical debt items were not always evaluated before being used in decision making. The decisions made in the release planning were based on other criteria, not on technical debt value.

Among the decision criteria, some were subjective and qualitative compared to the quantitative cost-benefit analysis approach we proposed. In general, technical debt was managed in a more implicit way, while our approach was treated as an “add-on” to their original release planning process and thus could be abandoned if necessary.

Based on the results of this study, we attributed the process deviation to obstacles that the project team encountered in the course of managing technical debt in their project. Due to its learning curve and technical difficulty, technical debt identification became the first major obstacle to applying the technical debt management process to the project. The difficulty of technical debt identification also incurred high cost at the beginning of technical debt management, which contributed to the aforementioned initial overhead. Another obstacle was rooted in the fact that the project team often had tight schedule for bug fix and feature implementation, which are usually given higher priority than maintainability problems like technical debt. For software managers, a common strategy to deal with time constraints is to delay some non-critical tasks, which is usually the primary reason that technical debt is incurred. Ironically, in the studied case, the proposed technical debt management approach

itself became a candidate to be abandoned. Therefore, time pressure was a significant force that drove the technical debt management off the prescribed course.

By looking deeper into the results of EducationHub study, we also found that a lack of process integration aggravated the process deviation, in that the project team regarded the technical debt management approach as an addition to their original sprint planning process rather than a part of it. In hindsight, there were clear ways that technical debt management could have been better integrated into the tools and processes used for release planning, as the two processes were not that different, but opportunities for integration were not obvious initially. As an add-on, it became easier for the project team to abandon technical debt management under time pressure, as discussed above. Therefore, lack of process integration became another obstacle.

The difficulty surrounding technical debt identification calls for more automatic approaches. Most of the approaches and tools currently available tend to yield a comprehensive set of technical debt items, but they rarely provide accurate interpretation of the importance of individual items. Thus the users often face the situation that they have a large set of technical debt candidates, but barely know which ones are more important or critical in the context of their project. A strategy of narrowing the focus of initial technical debt identification is particularly important to large and mature software projects such as the one chosen in the EducationHub study because the size of the code base and the length of the maintenance phase provide more opportunities for technical debt of a larger variety to be incurred. Certainly it is

not always true that a larger, more mature project will have more technical debt of more types than a smaller and younger software project, but one can intuit that this is likely to be the case often.

In both case studies, the criteria used for decision making about technical debt differed between the proposed technical debt management approach and the actual projects. In theory, principal and interest could incorporate notions of resource availability, customer preferences, and other decision criteria important to projects. But in reality, principal and interest were seen as representing only required effort. So during decision making, technical debt principal and interest needed to be combined (or, in the case of the EducationHub study, supplanted) with other factors to determine the priority of a debt item. This complicated the process of decision making, when the intent of the technical debt management approach was to simplify it. This experience leads us to recommend an approach to adopting explicit technical debt management that allows for a “transition” time, in which a project’s familiar decision making process and criteria are maintained. The notions of principal and interest are introduced, but initially represent simply required effort (e.g. to pay off the debt, in the case of principal). Gradually, ways to incorporate other decision criteria important to the project into the notions of principal and interest could be devised and evaluated. Eventually, guidelines could be developed to deal with any relevant decision criteria by folding it into the estimate of principal and interest, allowing a straightforward cost-benefit analysis to be used for prioritization of technical debt items. In this way the decision makers would be more likely to try technical debt management in their projects without worrying about its effect on their

original decision making process. This would make it more likely to see the benefit of technical debt management earlier, and thus improve the acceptance of technical debt management eventually in software practice.

8.3 Addressing the Research Questions

Through the series of studies, we have uncovered a set of benefits and costs associated with explicit technical debt management. These findings allow us to address the research questions. They also help us evaluate the effectiveness of explicit technical debt management by comparing the benefits with the costs.

8.3.1 RQ1: Costs and Benefits

Our first research question asks:

RQ1: What are the characteristics of the costs and benefits of measuring and monitoring technical debt?

We rely on findings from all four of our studies to address this question.

In the SMB retrospective study, we tracked a single technical debt item that had significant impact on the project. We compared the benefit, i.e. the effort that could have been saved, of explicit technical debt management to the effort of upgrading the communication protocol, which itself is a major maintenance task for the project.

Through the decision simulation we demonstrated that the benefit could be two times as high as the upgrading effort. By contrast, four technical debt items with low principal and interest were used to evaluate the benefit of explicit technical debt management in the Hadoop retrospective study. Although the benefit, i.e. the change

effort that could have been saved through explicit technical debt management, was smaller in number, it is proportional to the development effort of software artifacts containing the technical debt items, which is also the case in the SMB study. Given that the subject projects and technical debt items used in the two studies have very different profiles, we believe the finding about the magnitude of benefit achieved from explicit technical debt management can be generalized to some extent, e.g., to the cases we studied on the cost side of explicit technical debt management. It is in this sense that we consider the two types of studies we conducted are comparable and hence we can combine the results of the studies to address the research questions.

The Tranship case study and the EducationHub case study helped us understand the cost of technical debt management. According to the results from the Tranship study, managing technical debt in an explicit manner costs 1.325 person-hours per sprint on average, a 26% increase of the original release planning cost. Although 26% seems to be significant, 1.3 person-hours is very small compared to the project development cost, which is 37 person-hours on average for implementing a use case. In other words, technical debt management didn't add much burden to the project from the perspective of total project cost.

Through the two case studies we identified various types of costs and a cost pattern associated with technical debt management. As elaborated in Section 8.2, managing technical debt is subject to a large initial overhead, which includes the cost of technical debt identification and analysis for the first time. Analysis cost had a significant drop after the first development cycle, while other types of costs, such as

communication and documentation, remained low during the entire period of technical debt management practice. Therefore, reducing the initial overhead cost is most likely to reduce overall cost significantly. Because technical debt management has a learning curve, and technical debt identification is difficult technically and takes great effort, we consider an exhaustive search for technical debt items to construct an initial technical debt list may not be beneficial. Instead, we suggest that projects start with a small number of known technical debt instances whose impact on the project is relatively obvious or serious enough to warrant immediate attention, for example, the most “painful” technical debt items in the project. It has been demonstrated in the SMB study that a careful analysis of just one technical debt item could pay off, while the results from the EducationHub study show that managing a large technical debt list was not a cost-effective approach. Actually focusing on particular technical debt items rather than dealing with all technical debt issues at the same time not only reduces the management cost, but also helps the users overcome the learning curve and thus improve their confidence on using, or at least their willingness to try, technical debt management in their projects.

According to the Unified Theory of Acceptance and Use of Technology (UTAUT) [160], perceived ease of use and usefulness are two determinants of user acceptance of a new technology. In the case of technical debt management, perceived ease of use and usefulness mean higher expectation on the usability of the management approach and the benefit that explicit technical debt management can bring. Since the strategy of focusing on particular technical debt

items rather than dealing with all technical debt issues have such effect, it would contribute to the acceptance of technical debt management positively.

In summary, the explicit technical debt management approach can bring significant benefit, i.e. cost-saving, without adding much overhead to software projects when it has better process integration, more focused technical debt items or less costly technical debt identification approaches. The effort that can be saved through explicit technical debt management is largely proportional to the effort to construct the software artifacts that contain the technical debt items. Technical debt management incurs various types of cost. Among these costs, technical debt identification cost and the analysis cost for the first time account for the majority of the cost and are much higher than other types of cost and the analysis cost in the subsequent release cycles. The majority cost of technical debt management occurs in its early stage, which we identified as a large initial overhead, while the benefit remains on a constant level over time as long as the technical debt is controlled and paid off on time. Even the initial overhead can be reduced by following the suggested improvements of the technical debt management, such as the non-exhaustive technical debt search strategy discussed in this section. Therefore, we conclude that with the enhancements we suggest, explicit technical debt management can be an affordable approach that benefits software projects in different ways.

8.3.2 RQ2: Decision-making

Besides the cost and benefits insights, the results of the studies also provide a clear view on the use of technical debt information for decision making, which corresponds to our second research question:

RQ2: In what ways does technical debt information contribute to decision making?

There are different ways in which technical debt information can be used for decision making in release planning. In our proposed approach, we assume that technical debt information is used in release planning in a cohesive way, where other decision factors such as customer request and resource availability are incorporated into the notion of technical debt and reflected by the technical debt principal and interest. Then technical debt alone is used for release planning. But in the case studies technical debt information was used in ways different from the way we proposed. In the Tranship study, technical debt information was used as a single decision factor like other decision factors for the release planning. Because technical debt information was still used in release planning, we consider it a minor deviation from the way we proposed. In the EducationHub study, technical debt management was treated as an add-on process. In some situations the project team just followed their original release planning process without considering the technical debt information in decision-making at all, which we consider a major deviation from the way we expected them to use technical debt information for decision making.

In the case studies, we also gathered opinions of the project personnel on the benefits of technical debt management to complement the numeric evidence. The benefits of technical debt management perceived or observed by the project team include increased awareness of the technical debt problem that may be overlooked otherwise and improved understanding of technical debt impact through technical debt measurement. These opinions and observations also shed light on how technical debt information contributes to decision making.

In summary, technical debt information may serve as a comprehensive input, an individual factor, or an add-on part for decision making in software projects, depending on how effective their technical debt identification approaches are and how well the technical debt management is integrated into their decision making process. Technical debt information can be used in providing a holistic view of the potential maintenance problems hidden in software projects, and in analyzing the cost and benefit of project management strategies, thus raising managers' awareness of the impact of the problems and improving their decision making.

8.4 Contributions

This dissertation research is targeted to address the central problem of technical debt management – how technical debt should be leveraged to improve decision making of software development and maintenance projects. A set of research questions were developed accordingly and two types of studies were designed to investigate the cost/benefit side of technical debt management and the way in which technical debt information contributes to decision making. A total of 4 studies were carried out,

involving software projects with different profiles (domain, size, etc.). The exploratory nature of the studies facilitated gathering a rich body of qualitative data regarding technical debt management. These data not only helped us answer the research questions, but also allowed us to understand the rationale behind the answers.

Contribution 1: empirical evidence that explicit technical debt management can be cost-effective.

The results and findings from these studies together provided a chain of empirical evidence for the value of explicit technical debt management in comparison with implicit technical debt management, which is currently the dominant practice. The cost and benefit insights gained from the studies show that significant benefit, such as avoidance of high interest in future and fewer surprises on project outcome, can be achieved through explicit technical debt management without triggering high management cost. In spite of the obstacles to application of technical debt management in software projects, it's clear that explicit technical debt management is the right direction towards better decision making and software project management.

Contribution 2: empirical evidence of why explicit technical debt management is not always cost-effective.

Through the studies we also realized that high management cost could be incurred under some conditions. The cost can be so high that it prevents technical debt management practice, for example, in the case of the EducationHub study. In other words, explicit technical debt management is not always cost-effective. In the studies,

we observed both motivation and resistance towards using the explicit technical debt management approach. On one hand, the software managers have accepted the notion of technical debt and would like to try technical debt management in their projects, while on the other hand, technical debt management was still given a low priority and even becomes a candidate to be abandoned in case it competes for resources with other management plans. Analysis of the results from these studies helped us understand the reasons behind the phenomenon. We have found out that the project team encountered three major obstacles in the process of managing technical debt in their project using our proposed approach. It is these obstacles that resulted in the resistance and high management cost. Therefore, besides the aforementioned supporting evidence for the potential cost-effectiveness of technical debt management, these studies also provided empirical evidence that helps explain why and under what conditions the technical debt management approach may not be cost-effective.

Contribution 3: empirically grounded proposals for improvement that will make explicit technical debt management cost-effective more likely and more often.

Based on the findings and insights gained from the studies, we were able to propose solutions in the form of improvements to address the obstacles to technical debt management, including the non-exhaustive technical debt identification strategy at the beginning of technical debt management and use of more efficient technical debt identification approaches to reduce the initial management overhead and resistance from the users, and addition of a transition period to ease the integration of technical

debt management. We believe that these proposals for improvements will make technical debt management cost-effective more likely and more often.

8.5 Research Implications and Future Work

Through these studies we also identified the areas that more research effort should be devoted to. First, to facilitate incorporating decision factors into the technical debt notions of principal and interest, researchers need to come up with better technical debt principal and interest definitions that are more operational and easier to incorporate more decision factors. Second, since one of the findings of the Tranship case study is that large estimation error on technical debt principal or interest could result in different decisions, more effective technical debt estimation approaches are needed to improve the decision making. Third, the technical difficulty and high cost of technical debt identification suggests that developing cost-effective technical debt identification approaches is still an active area of research.

From the research point of view, we also call for more validation work on technical debt management as more empirical evidence will contribute to both the advance of technical debt management research and the adoption of explicit technical debt management by software practitioners. So far all studies we conducted on technical debt management, were hosted by commercial software companies from the private sector of the software industry. One area we haven't covered is governmental systems, i.e. large contractors for government agencies, which differ from private software businesses in such aspects as project cost control, management style, and even enterprise culture. These aspects may influence technical debt and its

management in their projects. Therefore, we've included in our future work to study software projects in governmental systems, which will help us uncover more factors affecting benefits and costs of technical debt management and thus improve the generalizability of the theory we are building for technical debt management.

Similarly, we also plan to study non-Agile projects (all those we've studied have been Agile), as software development process could affect technical debt management and the related decision making as well.

Bibliography

- [1] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060-1076, 1980.
- [2] S. M. Dekleva, "The Influence of the Information Systems Development Approach on Maintenance," *MIS Quarterly*, vol. 16, no. 3, pp. 355-372, 1992.
- [3] G. Alkhatib, "The maintenance problem of application Software: An empirical analysis," *Journal of Software Maintenance: Research and Practice*, vol. 4, no. 2, pp. 83 - 104, 1992.
- [4] W. Harrison and C. Cook, "Insights on improving the maintenance process through software measurement," in *Software Maintenance*, San Diego, CA , USA, 1990, pp. 37 - 45.
- [5] B. P. Lientz and B. E. Swanson, "Problems in application software maintenance," *Communications of the ACM*, vol. 24, no. 11, pp. 763 - 769, 1981.
- [6] B. P. Lientz and E. B. Swanson, "Software Maintenance Management," 05/01/1980 1980.
- [7] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, "Object-oriented software engineering," 1992.
- [8] B. W. Boehm, *Software engineering economics* (advances in computing science and technology series). Englewood Cliffs, N.J.: Prentice-Hall, 1981.
- [9] *ISO-20926 Software Engineering - Unadjusted Functional Size Measurement Method - Counting Practices Manual*, 2003.

- [10] A. Stellman and J. Greene, *Applied software project management*, Sebastopol, Calif.: O'Reilly, 2005, p. 256 p. [Online]. Available.
- [11] M. Jørgensen, "Estimation of Software Development Work Effort: Evidence on Expert Judgment and Formal Models," *International Journal of Forecasting*, vol. 23, no. 3, pp. 449-462, 2007.
- [12] M. Jørgensen, "A Review of Studies on Expert Estimation of Software Development Effort," vol. 70, no. 1-2, pp. 37-60, 2004.
- [13] B. Kitchenham and S. L. Pfleeger, "Software Quality: The Elusive Target," *IEEE Software*, Article vol. 13, no. 1, p. 12, 1996.
- [14] CAST, "Cast Worldwide Application Software Quality Study: Summary of Key Findings," CAST Report 2010.
- [15] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams, "An enterprise perspective on technical debt," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, 2011, pp. 35-38: ACM.
- [16] E. Lim, N. Taksande, and C. Seaman, "A Balancing Act: What Software Practitioners Have to Say about Technical Debt," *IEEE Software*, Article vol. 29, no. 6, pp. 22-27, 2012.
- [17] W. Cunningham, "The WyCash Portfolio Management System," in *Addendum to the proceedings on Object-oriented programming systems, languages, and applications*, 1992, pp. 29-30.
- [18] S. McConnell. (2007). *10x Software Development*. Available: <http://forums.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx>

- [19] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," (in English), *Journal of Systems and Software*, Article vol. 101, pp. 193-220, MAR 2015 2015.
- [20] C. Izurieta and J. M. Bieman, "Testing Consequences of Grime Buildup in Object Oriented Design Patterns," in *2008 1st International Conference on Software Testing, Verification, and Validation*, 2008, pp. 171-179.
- [21] J. Bohnet and J. Döllner, "Monitoring code quality and development activity by software maps," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, 2011, pp. 9-16: ACM.
- [22] N. Zazworka, M. A. Shaw, F. Shull, and S. Carolyn, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, Honolulu, HI, USA, 2011, pp. 17-23, 1985366: ACM.
- [23] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic, "Mapping architectural decay instances to dependency models," in *Proceedings of the 4th International Workshop on Managing Technical Debt*, San Francisco, CA, USA, 2013, pp. 39-46, 2663304: IEEE Press.
- [24] M. Fowler. (2009). *Technical Debt Quadrant*. Available: <http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html>
- [25] J. Yli-Huumo, A. Maglyas, and K. Smolander, "The Sources and Approaches to Management of Technical Debt: A Case Study of Two Product Lines in a Middle-Size Finnish Software Company," in *Proceedings of the 15th*

International Conference on Product-Focused Software Process Improvement (PROFES 2014), Helsinki, Finland, 2014, pp. 93-107.

- [26] J. Rothman. (2006). *An Incremental Technique to Pay Off Testing Technical Debt*. Available:
[http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL
&ObjectId=11011&tth=DYN&tt=siteemail&iDyn=2](http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=11011&tth=DYN&tt=siteemail&iDyn=2)
- [27] A. Lester. (2008). *Get Out of Technical Debt*. Available:
<http://petdance.com/perl/technical-debt/>
- [28] N. Brown *et al.*, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, Santa Fe, New Mexico, USA, 2010, pp. 47-52, 1882373: ACM.
- [29] K. Schmid, "On the limits of the technical debt metaphor: some guidance on going beyond," in *Proceedings of the 4th International Workshop on Managing Technical Debt*, San Francisco, California, 2013, pp. 63-66, 2663308.
- [30] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," (in English), *IEEE Software*, Editorial Material vol. 29, no. 6, pp. 18-21, 2012.
- [31] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," (in English), *Journal of Systems and Software*, Article vol. 86, no. 6, pp. 1498-1516, 2013.
- [32] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," (in

- English), *Information and Software Technology*, Article vol. 64, pp. 52-73, 2015.
- [33] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist, "Technical Debt in Test Automation," in *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation*, 2012, pp. 887-892.
- [34] C. Izurieta and J. M. Bieman, "A multiple case study of design pattern decay, grime, and rot in evolving software systems," *Software Quality Journal*, journal article vol. 21, no. 2, pp. 289-323, 2013.
- [35] C. Giardino, N. Paternoster, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, "Software Development in Startup Companies: The Greenfield Startup Model," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 585-604, 2016.
- [36] J. Holvitie, V. Leppänen, and S. Hyrynsalmi, "Technical Debt and the Effect of Agile Software Development Practices on It - An Industry Practitioner Survey," in *Proceedings of the 6th IEEE International Workshop on Managing Technical Debt (MTD' 14)*, Victoria, British Columbia, Canada, 2014, pp. 35-42.
- [37] J. Bird, "Technical Debt - How much is it Really Costing you?," vol. 2012, ed, 2012.
- [38] M. Fowler. (2003). *Technical Debt*. Available:
<http://www.martinfowler.com/bliki/TechnicalDebt.html>
- [39] J. L. Letouzey, "The SQALE method for evaluating Technical Debt," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*, 2012, pp. 31-36.

- [40] M. Smit, B. Gergel, J. H. Hoover, and E. Stroulia, "Code convention adherence in evolving software," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM '11)*, 2011, pp. 504-507.
- [41] J. Shore. (2006). *Quality With a Name*. Available:
<http://jamesshore.com/Articles/Quality-With-a-Name.html>
- [42] 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*, 2011, pp. 1-8: ACM.
- [43] M. Fowler. (2007). *Design Stamina Hypothesis*. Available:
<http://www.martinfowler.com/bliki/DesignStaminaHypothesis.html>
- [44] N. Ramasubbu and C. F. Kemerer, "Managing Technical Debt in Enterprise Software Packages," (in English), *IEEE Transactions on Software Engineering*, Article vol. 40, no. 8, pp. 758-772, 2014.
- [45] F. A. Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD' 12)*, 2012, pp. 15-22.
- [46] K. Schmid, "A formal approach to technical debt decision making," in *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, Vancouver, British Columbia, Canada, 2013, pp. 153-162, 2465492: ACM.
- [47] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In Search of a Metric for Managing Architectural Technical Debt," in *Proceedings of Joint*

- Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, 2012, pp. 91-100.
- [48] F. Bachmann, R. L. Nord, and I. Ozkaya. (2012) Architectural Tactics to Support Rapid and Agile Stability. *CrossTalk*. 20-25.
- [49] A. Martini, J. Bosch, and M. Chaudron, "Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study," *Information and Software Technology*, vol. 67, pp. 237–253, 2015.
- [50] R. N. Charette, *Applications strategies for risk analysis* (McGraw-Hill software engineering series). New York: Intertext Publications : McGraw-Hill, 1990, pp. xxiii, 570 p.
- [51] B. W. Boehm, "Software Risk Management: Principles and Practices," *IEEE Software*, Article vol. 8, no. 1, p. 32, 1991.
- [52] S. Alter and M. Ginzberg, "Managing Uncertainty in MIS Implementation," *Sloan Management Review*, Article vol. 20, no. 1, pp. 23-31, 1978.
- [53] G. B. Davis, "Strategies for information requirements determination," *IBM Systems Journal*, vol. 21, no. 1, pp. 4-30, 1982.
- [54] R. N. Charette, *Software engineering, risk analysis and management* (McGraw-Hill software engineering series). New York: Intertext Publications, McGraw-Hill, 1989, pp. xvii, 325 p.
- [55] R. Fairley, "Risk Management for Software Projects," *IEEE Software*, vol. 11, no. 3, pp. 57-67, 1994.
- [56] R. P. Higuera and Y. Y. Haimes, "Software Risk Management," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1996

- [57] J. Kontio, *Software Engineering Risk Management: A Method, Improvement Framework and Empirical Evaluation*. Helsinki, Finland: Suomen Laatu keskus, 2001.
- [58] H. R. Costa, M. d. O. Barros, and G. H. Travassos, "Evaluating software project portfolio risks," *Journal of Systems and Software*, Article vol. 80, no. 1, pp. 16-31, 2007.
- [59] R. L. Kumar, "Managing risks in IT projects: an options perspective," *Information & Management*, Article vol. 40, no. 1, p. 63, 2002.
- [60] G. Stoneburner, A. Goguen, and A. Feringa, "Risk management guide for information technology systems," Nat. Inst. Stand. & Technol., Washington, DC, Gaithersburg, MD, USA, 2002, Available:
<http://search.ebscohost.com/login.aspx?direct=true&db=inh&AN=7624734&site=e=ehost-live>.
- [61] M. J. Carr, S. L. Konda, I. Monarch, F. C. Ulrich, and C. F. Walker, "Taxonomy-Based Risk Identification," Software Eng. Institute, PittsburghCMU/SEI-93-TR-6, 1993.
- [62] (1994). *Software Risk Evaluation Method*.
- [63] C. J. Alberts, A. J. Dorofee, R. Higuera, R. L. Murphy, J. A. Walker, and R. C. Williams, *Continuous Risk Management Guidebook*. Pittsburgh, PA, USA, 1996: Software Engineering Institute, Carnegie Mellon University.
- [64] *Risk Management - Principles and guidelines on implementation*, 2002.
- [65] (2006). *Risk Management Guide for DoD Acquisition*.

- [66] (2002). *Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners*.
- [67] R. H. Hamm, "Selection of Verbal Probabilities: A Solution for Some Problems of Verbal Probability Expression," *Organizational Behavior & Human Decision Processes*, Article vol. 48, no. 2, p. 193, 1991.
- [68] S. Lichtenstein and R. J. Newman, "Empirical Scaling of Common Verbal Phrases Associated With Numerical Probabilities," *Psychonomic Science*, vol. 9, no. 10, pp. 563-564, 1967.
- [69] R. L. Keeney and D. von Winterfeldt, "Eliciting probabilities from experts in complex technical problems," *IEEE Transactions on Engineering Management*, vol. 38, no. 3, pp. 191-201, 1991.
- [70] D. A. Hillson and D. T. Hulett, "Assessing Risk Probability : Alternative Approaches," presented at the PMI Global Congress, Prague, Czech Republic, 2004.
- [71] N. Rescher, *Risk: A philosophical introduction to the theory of risk evaluation and management*. Washington, D.C., USA: University Press of America, 1983.
- [72] C. W. Kirkwood, *Decision Tree Primer*. Department of Supply Chain Management, Arizona State University, 2002.
- [73] J. E. Kelley Jr, "Critical-Path Planning and Scheduling: Mathematical Basis," *Operations Research*, Article vol. 9, no. 3, pp. 296-320, 1961.
- [74] A. Gemmer, "Risk management: Moving beyond process," *Computer*, Article vol. 30, no. 5, p. 33, 1997.

- [75] N. Dalkey and O. Helmer, "An Experimental Application of the Delphi Method to the Use Of Experts," *Management Science*, Article vol. 9, no. 3, pp. 458-467, 1963.
- [76] T. L. Saaty, *The Analytic Hierarchy Process*. McGraw-Hill, Inc., 1980.
- [77] K. Lyytinen and L. Mathiassen, "Attention Shaping and Software Risk--A Categorical Analysis of Four Classical Risk Management Approaches," *Information Systems Research*, Article vol. 9, no. 3, pp. 233-255, 1998.
- [78] T. Addison and S. Vallabh, "Controlling software project risks: an empirical study of methods used by experienced project managers," in *Annual Research Conference of Computer Scientists and Information technologists*, Port Elizabeth, South Africa, 2002, vol. 30, pp. 128 - 140.
- [79] F. J. Heemstra and R. J. Kusters, "Dealing with risk: a practical approach," *Journal of Information Technology*, Article vol. 11, no. 4, pp. 333-346, 1996.
- [80] R. C. Williams, J. A. Walker, and D. A. J, "Putting risk management into practice," *IEEE Software*, Article vol. 14, no. 3, pp. 75-82, 1997.
- [81] L. Wallace, M. Keil, and A. Rai, "Understanding software project risk: a cluster analysis," *Information & Management*, Article vol. 42, no. 1, pp. 115-125, 2004.
- [82] L. Wallace, M. Keil, and A. Rai, "How Software Project Risk Affects Project Performance: An Investigation of the Dimensions of Risk and an Exploratory Model," *Decision Sciences*, Article vol. 35, no. 2, pp. 289-321, 2004.

- [83] W. Han and S. Huang, "An empirical analysis of risk components and performance on software projects," *Journal of Systems and Software*, Article vol. 80, no. 1, pp. 42-50, 2007.
- [84] J. J. Jiang, G. Klein, and R. Discenza, "Information system success as impacted by risks and development strategies," *IEEE Transactions on Engineering Management*, vol. 48, no. 1, pp. 46-55, 2001.
- [85] E. H. Conrow and P. S. Shishido, "Implementing risk management on software intensive projects," *IEEE Software*, Article vol. 14, no. 3, p. 83, 1997.
- [86] B. A. Aubert, M. Patry, and S. Rivard, "Assessing the risk of IT outsourcing," in *Proceedings of the 34th Hawaii International Conference on System Sciences*, Los Alamitos, CA, USA, 1998, vol. 6.
- [87] M. Sumner, "Risk factors in enterprise-wide/ERP projects," *Journal of Information Technology*, Article vol. 15, no. 4, pp. 317-327, 2000.
- [88] S. Huang, C. I. Chang, S. Li, and M. Lin, "Assessing risk in ERP projects: identify and prioritize the factors," *Industrial Management and Data Systems*, Article vol. 104, no. 8, pp. 681-688, 2004.
- [89] I. Jacobson, G. Booch, and J. Rumbaugh, *The unified software development process* (The Addison-Wesley object technology series). Boston, MA, USA: Addison-Wesley, 1999.
- [90] S. H. Kan, *Metrics and models in software quality engineering*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2003, pp. xxvii, 528 p.
- [91] R. S. Pressman, *Software engineering : a practitioner's approach*, 6th ed. Boston, MA, USA: McGraw-Hill, 2005, pp. xxxii, 880 p.

- [92] I. Sommerville, *Software Engineering*, 8th ed. Harlow, England ; New York: Addison-Wesley, 2007, pp. xxiii, 840 p.
- [93] K. Akingbehin, "A quantitative supplement to the definition of software quality," in *Proceedings of the 3rd ACIS International Conference on Software Engineering Research, Management and Applications*, Los Alamitos, CA, 2005, pp. 348-352.
- [94] "IEEE Standard for Software Quality Assurance Processes," *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)*, pp. 1-138, 2014.
- [95] J. L. Letouzey and T. Coq, "The SQALE Analysis Model: An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code," in *Proceedings of the 2nd International Conference on Advances in System Testing and Validation Lifecycle*, 2010, pp. 43-48.
- [96] *Software engineering - Product Quality*, 2011.
- [97] J. McCall, P. Richards, and G. Walters, "Factors in Software Quality," US Department of Commerce, 1977.
- [98] B. W. Boehm, *Characteristics of software quality*. Amsterdam and New York: North-Holland, 1978.
- [99] R. G. Dromey, "A Model for Software Product Quality," *IEEE Transactions on Software Engineering*, Article vol. 21, no. 2, pp. 146-162, 1995.
- [100] *Systems and Software Engineering -- Vocabulary*, 2010.
- [101] J. Bansiya and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering*, Article vol. 28, no. 1, pp. 4-17, 2002.

- [102] B. W. Boehm and P. N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, Article vol. 14, no. 10, pp. 1462-1477, 1988.
- [103] G. A. Koru and H. Liu, "Building Effective Defect-Prediction Models in Practice," *IEEE Software*, Article vol. 22, no. 6, pp. 23-29, 2005.
- [104] N. Nagappan, L. Williams, M. Vouk, and J. Osborne, "Early estimation of software quality using in-process testing metrics: A Controlled Case Study," in *Proceedings of the 3rd workshop on Software quality*, St. Louis, Missouri, USA, 2005, pp. 1 - 7.
- [105] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel, "Early Quality Prediction: A Case Study in Telecommunications," *IEEE Software*, Article vol. 13, no. 1, pp. 65-71, 1996.
- [106] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow, "Comparing design and code metrics for software quality prediction," in *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, Leipzig, Germany, 2008, pp. 11-18.
- [107] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308-320, 1976.
- [108] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, Article vol. 7, no. 5, pp. 510-518, 1981.
- [109] D. N. Card and R. L. Glass, *Measuring software design quality*. Upper Saddle River, NJ, USA: Prentice-Hall, 1990.

- [110] V. R. Basili, L. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751-761, 1996.
- [111] L. H. Rosenberg and L. E. Hyatt. (1997) Software Quality Metrics for Object-Oriented Environments. *CrossTalk*.
- [112] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, Article vol. 20, no. 6, pp. 476-493, 1994.
- [113] M. Lanza, R. Marinescu, and S. Ducasse, *Object-oriented metrics in practice*. Berlin ; London: Springer, 2006, p. a 205 p.
- [114] E. J. Weyuker, "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering*, Article vol. 14, no. 9, pp. 1357-1365, 1988.
- [115] J. Tian and M. V. Zelkowitz, "Complexity Measure Evaluation and Selection," *IEEE Transactions on Software Engineering*, Article vol. 21, no. 8, pp. 641-650, 1995.
- [116] K. E. Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Transactions on Software Engineering*, Article vol. 27, no. 7, pp. 630-650, 2001.
- [117] G. A. Koru and J. Tian, "Comparing High-Change Modules and Modules with the Highest Measurement Values in Two Large-Scale Open-Source Products," *IEEE Transactions on Software Engineering*, Article vol. 31, no. 8, pp. 625-642, 2005.

- [118] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [119] E. V. Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering*, 2002, pp. 97-106.
- [120] H. Leung and Z. Fan, *Software Cost Estimation*. Handbook of Software Engineering and Knowledge Engineering, 2002.
- [121] N. E. Fenton and S. L. Pfleeger, *Software metrics : a rigorous and practical approach*, 2nd ed. Boston, MA, USA: PWS, 1997, pp. xii, 638 p.
- [122] G. Hall and J. Munson, "Software evolution: code delta and code churn," vol. 54, no. 2, pp. 111–118, 2000.
- [123] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111-122, 1993.
- [124] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 390-400: IEEE Computer Society.
- [125] S. Karus and M. Dumas, "Code churn estimation using organisational and code metrics: An experimental comparison," vol. 54, no. 2, pp. 203–211, 2012.
- [126] T. L. Graves and A. Mockus, "Inferring change effort from configuration management databases," in *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, 1998, pp. 267-273.

- [127] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the 16th International Conference on Software Maintenance*, 2000, pp. 120-130.
- [128] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the Effect of Code Smells on Maintenance Effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144-1156, 2013.
- [129] K. E. Emam, "A Methodology for Validating Software Product Metrics," 2000.
- [130] S. G. MacDonell, "Comparative review of functional complexity assessment methods for effort estimation," *Software Engineering Journal*, vol. 9, no. 3, pp. 107 - 116, 1994.
- [131] A. J. Albrecht, "Measuring Application Development Productivity," in *IBM Application Development Symposium.*, 1979, pp. 83 - 92.
- [132] C. R. Symons, "Function Point Analysis: Difficulties and Improvements," *IEEE Transactions on Software Engineering*, Article vol. 14, no. 1, pp. 2-11, 1988.
- [133] R. D. Banker, R. J. Kauffman, and R. Kumar, "An Empirical Test of Object-based Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment," *Journal of Management Information Systems*, Article vol. 8, no. 3, pp. 127-150, 1991.
- [134] L. H. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," *IEEE Transactions on Software Engineering*, Article vol. 4, no. 4, pp. 345-361, 1978.
- [135] C. N. Parkinson, *Parkinson (1957) Parkinson's law, and other studies in administration*. Boston: Houghton Mifflin, 1957.

- [136] A. J. Albrecht and J. E. Gafeney Jr, "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Transactions on Software Engineering*, Article vol. 9, no. 6, pp. 639-648, 1983.
- [137] B. W. Boehm, *Software cost estimation with Cocomo II*. Upper Saddle River, NJ: Prentice Hall, 2000.
- [138] B. W. Boehm, C. Abts, and S. Chulani, "Software Development Cost Estimation Approaches - A Survey," *Annals of Software Engineering*, vol. 10, no. 10, pp. 177-205, 2000.
- [139] C. F. Kemerer, "An Empirical Validation of Software Cost Estimation Models," *Communications of the ACM*, Article vol. 30, no. 5, pp. 416-429, 1987.
- [140] F. J. Heemstra and R. J. Kusters, "Function point analysis: evaluation of a software cost estimation model," *European Journal of Information Systems*, vol. 1, no. 4, pp. 229-237, 1991.
- [141] A. L. Lederer and J. Prasad, "Information systems software cost estimating: a current assessment," *Journal of Information Technology*, Article vol. 8, no. 1, pp. 22-33, 1993.
- [142] R. T. Hughes, A. Cunliffe, and F. Young-Martos, "Evaluating software development effort model-building techniques for application in a real-time telecommunications environment," *IEEE Proceedings-Software*, vol. 145, no. 1, pp. 29-33, 1998.
- [143] I. Myrtveit and E. Stensrud, "A Controlled Experiment to Assess the Benefits of Estimating with Analogy and Regression Models," *IEEE Transactions on Software Engineering*, Article vol. 25, no. 4, pp. 510-525, 1999.

- [144] L. C. Briand, K. E. Emam, D. Surmann, I. Wieczorek, and K. D. Maxwell, "An assessment and comparison of common software cost estimation modeling techniques," in *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, USA, 1999, pp. 313-322.
- [145] M. Shepperd, C. Schofield, and B. Kitchenham, "Effort Estimation Using Analogy," in *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, 1996, pp. 170 - 178.
- [146] S. L. Pfleeger, F. Wu, and R. Lewis, *Software cost estimation and sizing methods : issues, and guidelines*. Santa Monica, CA, USA: Rand Corp., 2005, pp. xxvii, 97 p.
- [147] F. Walkerden and R. Jeffery, "An empirical study of analogy-based software effort estimation," *Empirical Software Engineering*, vol. 4, no. 2, pp. 135-158, 1999.
- [148] M. Jørgensen and M. Shepperd, "A Systematic Review of Software Development Cost Estimation Studies," *IEEE Transactions on Software Engineering*, Article vol. 33, no. 1, pp. 33-53, 2007.
- [149] M. Shepperd and G. Kadoda, "Comparing Software Prediction Techniques Using Simulation," *IEEE Transactions on Software Engineering*, Article vol. 27, no. 11, pp. 1014-1022, 2001.
- [150] C. Hinsman, N. Sangal, and J. Stafford, "Achieving Agility Through Architecture Visibility," in *Proceedings of the 5th International Conference on the Quality of Software Architectures*, Berlin, Germany, 2009, vol. 5581, pp. 116-129.

- [151] B. Flyvbjerg, "Five misunderstandings about case-study research," (in English), *Qualitative Inquiry*, vol. 12, no. 2, pp. 219-245, 2006.
- [152] Y. Guo *et al.*, "Tracking technical debt - an exploratory case study," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM '11)*, 2011, pp. 528-531.
- [153] N. Zazworka *et al.*, "Comparing four approaches for technical debt identification," *Software Quality Journal*, journal article vol. 22, no. 3, pp. 403-426, 2014.
- [154] Y. Guo, R. Spinola, and C. Seaman, "Exploring the costs of technical debt management - a case study," (in English), *Empirical Software Engineering*, Article vol. 21, no. 1, pp. 159-182, 2016.
- [155] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," (in English), *IBM Journal of Research and Development*, Article vol. 56, no. 5, pp. 9:1-9:13, 2012, Art. no. ARTN 9.
- [156] P. Wang, J. Yang, L. Tan, R. Kroeger, and D. J. Morgenthaler, "Generating precise dependencies for large software," in *Proceedings of the 4th International Workshop on Managing Technical Debt (MTD '13)*, 2013, pp. 47-50.
- [157] R. K. Yin, *Case Study Research: design and methods*, 2nd ed. Thousand Oaks: Sage Publications, 1994.
- [158] Y. Guo, C. Seaman, and F. Q.B. da Silva, "Costs and obstacles encountered in technical debt management – A case study," *Journal of Systems and Software*, vol. 120, pp. 156-169, 2016.

- [159] Y. Guo, C. Seaman, N. Zazworka, and F. Shull, "Domain-Specific Tailoring of Code Smells: An Empirical Study," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, vol. 2, pp. 167-170.
- [160] V. Venkatesh, M. G. Morris, G. B. Davis, and F. D. Davis, "User acceptance of information technology: Toward a unified view," (in English), *Mis Quarterly*, Article vol. 27, no. 3, pp. 425-478, 2003.

