

Measuring and Monitoring Technical Debt

CAROLYN SEAMAN

Department of Information Systems, University of Maryland Baltimore County, Baltimore, Maryland, USA

YUEPU GUO

Department of Information Systems, University of Maryland Baltimore County, Baltimore, Maryland, USA

Abstract

Technical debt is a metaphor for immature, incomplete, or inadequate artifacts in the software development lifecycle that cause higher costs and lower quality in the long run. These artifacts remaining in a system 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 may speed up software development in the short run, but such benefit is achieved at the cost of extra work in the future, as if paying interest on the debt. In this sense, the technical debt metaphor characterizes the relationship between the short-term benefits of delaying certain software maintenance tasks or doing them quickly and less carefully, and the long-term cost of those delays. However, managing technical debt is more complicated than managing financial debt because of the uncertainty involved. In this chapter, the authors review the main issues associated with technical debt, and propose a technical debt management framework and a research plan for validation. The objective of our research agenda is to develop and validate a comprehensive technical debt theory that formalizes the relationship between the cost and benefit sides of the concept. Further, we propose to use the theory to propose mechanisms (processes and tools) for measuring and managing technical debt in software product maintenance. The theory and management mechanisms are intended ultimately to contribute to the improved quality of software and facilitate decision making in software maintenance.

- 1. Introduction 26
- 2. The Technical Debt Metaphor 27
- 3. Related Work 30
 - 3.1. Software Project Risk Management 31
 - 3.2. Software Quality Assessment and Program Analysis 33
 - 3.3. Software Development Effort Estimation 34
- 4. Managing Technical Debt 35
 - 4.1. Identifying Technical Debt 35
 - 4.2. Measuring Technical Debt 36
 - 4.3. Monitoring Technical Debt 38
- 5. Open Research Issues 42
- 6. Conclusion 45
- References 45

1. Introduction

A major obstacle to fulfilling society’s demand for increasingly complex software systems is the resources that many organizations must devote to maintaining existing systems that cannot, for business reasons, be abandoned. The quality of maintained software often diminishes over time, with respect to its internal system structure, adherence to standards, documentation, understandability, and so on. This eventually impacts maintenance productivity, as modification of low-quality software is generally harder than of high-quality software. The magnitude of the maintenance burden is well documented [1–3] and not likely to decrease. Therefore, advances that address maintenance productivity have a twofold benefit: cost savings in the maintenance of existing systems and newly available resources for development of systems to address emerging problems.

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. For business reasons, it does not always make sense to completely close this gap, as much of it may not pay off in the end. 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. The difficulty, of course, is that it is rarely known if a particular portion of the system has defects or not, or if a particular module is ever going to need modification. So the problem is really one of managing risk, and of making informed decisions about which delayed tasks need to be accomplished, and when.

Currently, managers and leaders of software maintenance efforts 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 in completing required modifications and compromised quality. Articulating this issue in terms of “technical debt” has helped some practitioners to discuss this issue. This metaphor frames the problem of delayed maintenance tasks as a type of “debt,” which brings a short-term benefit (usually in terms of higher productivity or shorter release time) but which might have to be paid back, with “interest,” later. The “principal” on the debt is the amount of effort required to “pay off” the debt (i.e., complete the task), 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 these tasks in the present. Many practitioners find this metaphor intuitively appealing and helpful in thinking about the issues. What is missing, however, is an underlying theoretical basis upon which management mechanisms can be built to support decision making.

The technical debt metaphor [4] has captured the interest of practitioners because it so effectively describes a deep intuitive understanding of the dynamics of software maintenance projects. Discussion and enhancement of the concept is transforming the way that long-term software maintenance is *viewed*. But its lack of a sound theoretical basis (or even any scholarly examination), empirically based models, and practical implementation hinder its ability to transform how maintenance is *done*.

2. The Technical Debt Metaphor

Technical debt refers to the effect of any incomplete, immature, or inadequate artifact in the software development lifecycle, for example, immature design, incomplete documentation, and so on. These artifacts remaining in a system affect subsequent development and maintenance activities, and so can be seen as a type of debt that the system developers owe the system, whose repayment may be demanded sooner or later. It is common that a software project incurs some debt in the development process because small amounts of debt can increase productivity. However, technical debt brings risks to the project, for example, compromised

system architecture or less maintainable code. In addition, the existence of technical debt complicates software management as managers have to decide whether and how much debt should be paid and when. In order to make informed decisions, it is necessary to have thorough knowledge of the present and future value of the technical debt currently held by the project. Therefore, identifying, measuring, and monitoring technical debt would help managers make informed decisions, resulting in higher quality of maintained software and greater maintenance productivity.

Although the term may be fairly recent, the concept behind technical debt is not new. It is related to Lehman and Belady's [5] notion of *software decay* (increasing complexity due to change) and Parnas' [6] *software aging* phenomenon (the failure of a product to continue to meet changing needs). These issues have been studied for decades by software maintenance and quality researchers. The introduction of the technical debt metaphor provides a new way to talk about, manage, and measure these related concepts.

The term "technical debt" was first coined by Cunningham [7], in which 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 [7]. This metaphor has been extended to refer to any imperfect artifact in the software lifecycle. Although technical debt has not been formally investigated, 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 identification of technical debt to solutions for controlling it.

Technical debt can be described as either unintentional or intentional [8]. Unintentional debt occurs due to a lack of attention, for example, lack of adherence to development standards or missed test cases. Intentional debt is incurred proactively for tactical or strategic reasons such as to meet a delivery deadline. Technical debt can also be classified in terms of the phase in which it occurs in the software lifecycle, for example, design debt or testing debt [9]. Design debt refers to the integrity of the design as reflected in the source code itself. Testing debt refers to tests that were not developed or run against the code. From this perspective, a whole range of phenomena is related to technical debt. 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 is too difficult to keep different portions of the system architecturally consistent.

These phenomena can be seen as symptoms of technical debt.

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.

Since technical debt is closely tied to software quality, metrics for software quality can also be used to measure technical debt. For example, if the software has defects or does not satisfy all the system requirements, then the defects will eventually have to be fixed and the requirements eventually implemented. Therefore, the number of defects currently in the system, or the number of pending requirements, is an indicator of technical debt. 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 [10]. Ultimately, a good design is judged by how well it deals with changes [10], so time and effort required for changes (in particular the trend over time) is also an indicator of technical debt.

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 technical debt currently in the system.

Strategies for paying off technical debt have also been discussed. One strategy is to pay the highest interest debt items first. In other words, this strategy focuses on the debt items that have severe potential future impact on the project. If the debt items with high interest are paid as soon as they are identified, the project avoids that high interest, which in some cases can lead to project failure or “bankruptcy.” In terms of the overall cost—the total amount of interest that the project needs to pay—this strategy is not necessarily the optimal one, but it could lower the risk level of the project and keep technical debt under control. Another solution is to determine a breakeven point [4] along the timeline of the project at which technical debt should be paid. This strategy is based on the rationale that technical debt remaining in the system hurts the productivity of the project, which can be measured by the amount of functionality that the development team can implement per time unit. For example, suppose a software project manager, in planning the implementation of an enhancement, allocated relatively little time to analyzing and documenting the design upfront. As a result, the project delivered more functionality to market in the release in which the enhancement was implemented. However, the consequence is that the modules added and changed to implement that enhancement are more difficult to

work with in the long run. This, then, is accumulated technical debt, which may slow down productivity later. A good design does require upfront time and effort, but it makes software flexible and easier to maintain for longer periods of time.

To extend this example, suppose we could estimate that the productivity (P) for a project in which enhancements are all designed and documented carefully is 10 requirements per day (assuming, in this context, that requirements per day is a reasonable measure of productivity) and it remains steady in the first 3 months. Thus the cumulative number of requirements that P can implement is 300 in the first month, 600 in the first 2 months, and 900 in the first 3 months. By contrast, the same project using a quick and dirty approach to designing enhancements (P') has higher productivity in the first month, but it decreases as the development proceeds. P' delivered 500 requirements in the first month, 700 in the first 2 months, and 900 in the first 3 months. With this trend, P' can no longer prevail in terms of the cumulative functionality after the third month. Therefore, in this example, the end of the third month is the breakeven point. Prior to this point, incurring technical debt yields benefit (early time to market) for P' , while beyond this point the negative effect of technical debt invalidates such benefit and hence the debt should be paid (which, in this case, would involve reengineering and/or redocumenting the design, and possibly refactoring the code). This strategy stresses the need to determine whether it is cost-effective to incur debt, when it should be paid, and how much will be paid each time in order to minimize cost or maximize profit.

3. Related Work

The characteristics of technical debt warrant a strong relationship to several existing areas of active research, including software quality assessment, software effort estimation, and software risk management. A number of metrics and program analysis approaches have been proposed to assess, monitor, and predict software quality. Research results in effort estimation facilitate software project planning, investment analyses, and product pricing. Software risk management has been integrated into software development processes to improve software quality and productivity. Although research achievements are fruitful in each area, there is no work that has leveraged the capability of these three areas to address the technical debt management problem. However, any new work in the technical debt area must be informed by existing approaches to assessing software quality, estimating effort, and managing risk.

3.1 Software Project Risk Management

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. But this is usually not the case for technical debt. For example, the debt incurred by not completely testing a certain portion of the system never has to be paid back if that portion in fact has no defects. Similarly, if a particular module is never going to be modified in the future, then the debt incurred by failing to update its related documentation will save some time during modification of the module, but will not cause any problems if it is never repaid. Therefore, technical debt involves uncertainty that is not normally present with the analogous notion of financial debt.

Ideally, software managers can choose to incur technical debt only on those artifacts without interest, or delay payment on these artifacts while focusing on those that are subject to penalty. Nonetheless, the difficulty is, using the above examples, 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. 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 is closely related to managing risk and making informed decisions about which delayed tasks need to be accomplished, and when.

Our approach to managing technical debt draws inspiration from approaches to software risk management proposed in the literature. Risk management approaches can generally be classified into methods for identifying risks, analyzing risks, or managing risks over time.

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. For example, Carr et al. proposed a risk identification approach based on the SEI taxonomy of software development risks [11]. The types of risks implied by technical debt fall into the product engineering category in the SEI taxonomy. Table I lists the major classes in this taxonomy and the elements in each class.

Risk analysis, or assessment, attempts to describe the identified risks in a way that will help stakeholders decide what actions to take. A common set of attributes to describe risks, as listed in Table II, is also proposed in the literature [12,13]. Our approach to technical debt management makes use of these attributes, with special emphasis on the attributes impact and probability. Other attributes of risks such as scope, class, and causes are included as well. The simplest metric scale for both probability and impact is an ordinal rating scale consisting of low, medium, and high [14]. Since the

TABLE I
SEI TAXONOMY OF SOFTWARE DEVELOPMENT RISKS [11]

Risk class	Product engineering	Development environment	Program constraints
Risk element	Requirements	Development process	Resources
	Design	Development system	Contract
	Code and unit test	Management process	Programming interfaces
	Integration and test	Management methods	
	Engineering specialties	Work environment	

TABLE II
RISK ATTRIBUTES [12,13]

Risk attribute	Description
Class	The type of the risk
Cause	The events that lead to the risk
Scope	The range in which the risk is considered
Impact	The severity of potential loss of the risk
Probability	The likelihood that the risk gets instantiated
Valid time period	The time frame in which the risk is valid

assignment of such values is highly subjective, some methods have been proposed to reduce the potential bias [15]. Better estimation of the impact and probability can be achieved using historical data that match current project characteristics. Based on the estimation of their impact and probability of occurrence, the risks can then be prioritized using a risk analysis matrix [16], tree-based techniques [17], or other risk models. We adopt a similar approach to quantifying technical debt, starting from rough estimation, refining the estimation using historical data, and finally prioritizing the technical debt items based on a combination of impact and probability.

Managing risks over time follows an iterative process that continues throughout the project [18]. This includes updating information about risks as they are handled and reevaluating risks as more information becomes available or context factors change. There are various strategies for risk management, generally classified into strategies for avoiding a risk, mitigating a risk, or transferring a risk to others. The decision is based on the results of risk assessment. It is to be noted that assessing risks for a project is not a one-time task because risks will change as a result of environmental change and actions taken to manage them. Therefore, risks should be continuously monitored to decide whether the risk is becoming more or less probable and whether the effects of the risk have changed [19]. The same is true for technical debt items, which can change with the context of the project.

3.2 Software Quality Assessment and Program Analysis

As mentioned in the introduction, technical debt refers to immature artifacts that fail to meet certain software quality criteria. Therefore, technical debt can in some cases 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 [20]. On the other hand, some attributes of quality, for example, 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 is inevitably subjective and difficult to quantify. Another approach to software quality assessment is to use software quality metrics. For example, lines of code can be used to measure the size of software, while cyclomatic complexity and fan-in/fan-out are used to measure software complexity. No matter what metrics are used, the relationship between the metrics and software quality attribute must be determined so that it is clear that software quality can be properly assessed using these metrics. Research regarding software quality metric evaluation [21–23] can be used to help select the appropriate metrics for a particular quality measurement goal.

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” [24], are shown to cause maintainability problems over time because they make the software less understandable, more complex, and harder to modify. Thus, code smells can be considered as a type of technical debt. Code smells can be categorized depending on their detection methods.

- *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 and require more complex computations and extractions of software quality metrics values from source code. They are computed based on Boolean expressions that include code metrics and thresholds. Based on established quality metrics, Marinescu and

Lanza [25] studied the disharmonies in software source code and proposed a set of criteria including metrics and threshold values to detect code smells. An example of a derived smell is a class that implements too much responsibility (known as a “God class”). The following expression evaluates if the “God class” smell is present:

$$\text{WMC} > 47, \quad \text{ATFD} > 5, \quad \text{and} \quad \text{TCC} < 0.33,$$

where WMC is the weighted methods per class, ATFD is the number of accesses to foreign class data, and TCC is tight class cohesion. These kinds of code smells require, on one hand, a rigorous definition of the metrics used (e.g., how are methods weighted when computing WMC?) and a set of base-lines to define the included thresholds. Although this approach facilitates automated code smell detection, the metrics and thresholds should be tailored to improve accuracy in a particular domain [26].

- The last category of smells includes those that are not automatically detectable. These 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 [24]. 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 [27].

3.3 Software Development Effort Estimation

The currency of choice in software project management is often effort, rather than dollars or euros or pounds or any other national currency. Effort is typically the largest component of cost in a software project, and is the resource that requires the most management. Thus, our approach to management of technical debt focuses on representing such concepts as principal and interest in terms of effort. This implies that methods for estimating the effort required for accomplishing various tasks are central to managing technical debt. Since the 1960s, various estimation techniques based on either formal estimation or expert estimation have been proposed, such as COCOMO [28], function point analysis [29], and Wideband Delphi [30].

The goal of effort estimation research is primarily to improve estimation accuracy. However, the accuracy of the existing approaches heavily depends on the context where the approaches are applied [31]. In addition, some approaches are too complicated to be applied though promising in their estimation accuracy. Hence, the dominant estimation approach in practice is still expert estimation [32]. Managing technical debt does not require more accurate or applicable effort estimation techniques than other areas of software project management, but the factors that go into choosing an estimation technique are similar. That is, estimation techniques that are appropriate for the domain, and that can be applied usefully in the context, are the most effective in any particular case.

4. Managing Technical Debt

In this section, we propose an initial technical debt management mechanism as a framework within which to develop theory and validate the approach. This mechanism is described below, and 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 theories emerging from future work in this area.

4.1 Identifying Technical Debt

The proposed approach to technical debt management centers around 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. 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 or updated, architectural compliance issues, known latent defects that need to be removed, and so on. Each item includes a description of where (in the system) the debt item is 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 effort required to complete the task. The interest is composed of two parts. The “interest probability” is the probability that the debt, if 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 if this debt item is 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, over and

above what it would have cost if they had been found during testing. The technical debt list must be reviewed and updated after each release, when items should be added as well as removed. Table III shows an example of a technical debt item.

Maintainers and managers can add items to the technical debt list at different points in the release process, depending on the type of debt (adapted from Ref. [9]):

- *Testing debt* items are added each time a decision is made to skip some set of tests (or reviews) for a particular release or module.
- *Defect debt* items are added each time a defect is found but not fixed in the current release.
- *Documentation debt* items are added each time a piece of software is modified but there is no corresponding modification to the associated documentation.
- *Design debt* items are generated when deficiencies in the code are identified, such as poor programming practices or violations of the system architecture.

4.2 Measuring 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 they are required to do fine-grained planning 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

TABLE III
TECHNICAL DEBT ITEM

ID	37
Date	3/31/2007
Responsible	Joe Blow
Type	Design
Location	Component X, especially function Y
Description	In the last release, function Y was added quickly and the implementation violates some elements of the architectural design
Estimated principal	Medium (medium level of effort to modify)
Estimated interest amount	High (if X has to be modified, this could cause lots of problems)
Estimated interest probability	Low (X not likely to be modified any time soon)

available, but general procedures 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 are used to achieve a more accurate estimation beyond the initial high/medium/low assessment. If an organization has very accurate, rich, detailed data on many projects, the estimation is very reliable. But even if the historical data are limited, an estimate can still be derived that is helpful in the technical debt decision-making process. Even an expert opinion-based estimate is useful in this context.

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, and so on. 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. The specific procedures for doing these estimations will vary depending on the form and nature of the historical data available, but such procedures can be established for any given maintenance environment. 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 it 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

$$\text{Interest amount} = W \times C$$

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.

4.3 Monitoring Technical Debt

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 much, and which technical debt items should be paid 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: 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.

- Step 1. Extract all technical debt items associated with component X .
- Step 2. Reevaluate high/medium/low estimates for these items based on current plans for the upcoming release (e.g., some items may have a greatly reduced principal if they overlap with the planned work in the release).
- Step 3. Do numeric estimates for all items with high interest probability and high interest amount.
- Step 4. For each item considered in Step 3, compare cost (principal) with benefit (interest probability \times interest amount) and eliminate any item for which the benefit does not outweigh the cost.
- Step 5. Add up the estimated principal for all items left after Step 4. Decide if this cost can be reasonably absorbed into the next release. If not, use this analysis to justify the cost to management. If so, can more debt repayment be put into this release? If so, repeat Steps 3–5 with items with high interest probability and medium interest amount, and vice versa, then with medium for both probability and interest, and so on, until no more debt repayment can be absorbed by the release.

Example

System S has had two major releases: R1 and R2. The next major release, R3, is currently being planned. Since the first version, all technical debt items in S have been tracked and recorded in the technical debt list L, which is updated whenever S is modified. In the plan for R3, Module M will have significant changes. The budget for this release is 600 person-days, 2% of which can be used for paying off technical debt associated with M. According to L, there are five technical debt items in M. Table IV shows these items and their initial estimates of principal, interest probability, and interest amount.

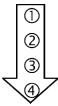
Since most functions related with A are already planned for refactoring in release R3, we first update the estimated principle of A to low (signifying that paying off technical debt item A will impose very little effort beyond what is already planned for the release). After examining the release plan for R3, we decide that no other technical debt items are affected, and the coarse-grained estimates in Table IV do not need to be adjusted, with the exception of the principal for item A, as mentioned previously. However, in order to use this information for decision making, we need to be more precise and quantitative at this point. So, first, we assign 0.2, 0.5, and 0.8 to the interest probability levels—low, medium, and high—respectively. We also derive estimates for principal and interest amounts based on historical effort data. For example, we use data on past refactoring effort to estimate the effort to refactor the methods included in technical debt item A that are not already included in the release plan for R3. Note that these estimates are neither easy to create nor highly accurate. However, they are not created until needed in the analysis, and they provide as much information as is possible given the historical data available, which in any case will provide more insight than not tracking the debt at all. Table V shows the adjusted technical debt items and their numeric estimates.

We first examine item A because A has both high interest amount (6) and high interest probability (0.8). The benefit of paying off A is 4.8 (6×0.8), while the cost, that is, the principal, of paying off A is 4. Since the benefit of paying off A outweighs

TABLE IV
TECHNICAL DEBT ITEMS OF MODULE M IN SYSTEM R

Name	Type	Principal	Interest probability	Interest amount
A	Design debt	Medium	High	High
B	Design debt	High	Medium	High
C	Defect debt	Medium	Medium	Medium
D	Testing debt	Low	Low	Medium
E	Documentation debt	Low	Low	Low

TABLE V
RELEASE PLANNING PROCESS FOR TECHNICAL DEBT

Technical debt item	Principal/cost	Interest probability	Interest amount	Benefit	Decision	Total cost	Step order
A	4 (low)	0.8	6	4.8	Pay off	4	
B	10	0.5	4	2	Delay	4	
C	6	0.5	13	6.5	Pay off	10	
D	2	0.2	11	2.2	Pay off	12	
E	Low	Low	Low	Low	Delay	Low	

the cost, we keep it on the list. The next item we consider is B, which has medium interest probability and high interest amount. Since the benefit of paying off B is less than the cost, we remove it from the list. Following the same procedure, we calculate the cost and benefit of C and D. According to the decisions we have made so far, A, C, and D will be paid off and the total cost is 12 person-days, which has reached the budget limit. Therefore, this process terminates. Although E is not considered because of the budget limit, we are confident on this decision because E is a debt item with low interest probability and low interest amount and hence will not cause big problems to the next release. Table V illustrated the decision-making process.

Scenario 2: 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 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 (TP), which is calculated by summing up over entire list (set three points for high, two for medium, one for low), and
 - (5) weighted total interest (TI) (add points for probability and amount).

Example

System X was first released 2 years ago. Since then, numerous modifications have been made. X currently has three minor releases, v1, v2, and v3. Since the first version, all technical debt items in S have been tracked and recorded in a technical debt list, which is updated whenever X is modified. The weighted TP and the weighted TI are calculated monthly, as shown in Table VI.

TABLE VI
TECHNICAL DEBT HISTORY IN SYSTEM X

	October 2009	November 2009	December 2009	January 2010	February 2010	March 2010	April 2010	May 2010	June 2010	July 2010	August 2010	September 2010
TP	5	8	13	18	30	34	7	8	12	15	20	22
TI	3	6	9	15	25	39	1	3	4	6	10	18
	October 2010	November 2010	December 2010	January 2011	February 2011	March 2011	April 2011	May 2011	June 2011	July 2011	August 2011	September 2011
TP	3	6	8	10	11	16	10	15	22	30	35	37
TI	1	4	9	14	17	19	4	6	15	26	40	50

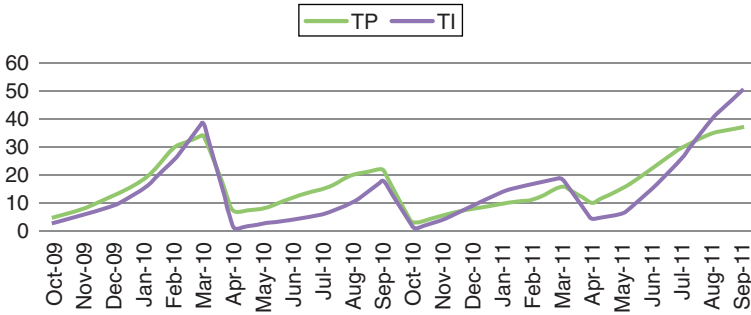


FIG. 1. Trends of technical debt in system X.

Based on the data in [Table VI](#), we can plot charts to monitor the trends of technical debt remaining in X. [Figure 1](#) is an example of the charts. From this chart, we can see that the amount of technical debt in X has an increasing tendency along the timeline, but it dropped to a lower level after each minor release, indicating that effort was devoted to paying down the debt in each release plan. Currently, technical debt has increased to a very high level. In addition, the amount of interest is higher than the amount of principle, which means maintenance of X is currently crippled by a high amount of technical debt. Therefore, it is time to pay down the debt.

This initial proposed approach to technical debt management (both identification and monitoring) relies heavily on the analysis and visualization of existing data from prior software development and maintenance efforts. The reliability and usefulness of the decision-making support that this approach provides increases 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, or even no 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. A benefit of the approach is that it utilizes software metrics data, to the extent that it is available, in new ways, further motivating the collection of this data.

5. Open Research Issues

The technical debt concept, while intuitively appealing and helpful in facilitating discourse, currently lacks an underlying theory, models to aid its analysis, and mechanisms to facilitate its management. Thus, future work in this area needs to progress toward two related aims:

- A comprehensive technical debt theory must be developed that formalizes the relationship between the cost and benefit sides of the concept.
- Practical mechanisms must be developed and validated for exploiting the theory in management decision making.

The authors of this chapter are involved in a research agenda designed to address the above two aims. Specifically, two studies are currently underway, a retrospective study and a case study of a pilot of the technical debt management approach. The retrospective study will determine what types of benefits might have been incurred by a project had they tracked technical debt. The study is based on in-depth analysis of historical effort, defect, testing, and change data over several recent releases of an ongoing development project at one of our industrial partners. With help from project personnel in interpreting the data and adding historical context, we are analyzing all data available to reconstruct a technical debt list as it would have been built during a selected release if the project team had used the proposed technical debt management mechanism. Based on the reconstructed list, decisions will be simulated about the content of the subsequent release. Where the simulated decisions differ from the decisions actually made, the estimates for the affected technical debt items are used to determine the principal that would have been paid back and the interest that would have been avoided if the decisions had been made based on a technical debt list. This analysis will show whether a benefit would have been gained by using a technical debt list, and what factors contributed to that benefit. This analysis will be repeated with data from multiple releases.

While the retrospective study will provide evidence of a benefit from applying the technical debt management approach, it will not shed any light on the costs of the approach. This insight will come from a pilot of the approach on the same project, going forward. Before implementation of the technical debt mechanism in the chosen projects, baseline data will be collected in order to provide a basis for comparison. Such data will include basic effort, cost, and productivity data as well as qualitative data about the project's history of technical debt. The latter data will include the gathering of "stories" [33] about unexpected debt repayment, crises caused by unpaid debt (or at least recognized as such *post hoc*), and decisions that had been made based upon an implicit understanding of technical debt.

Once the baseline data are collected, the technical debt mechanism will be tailored to the specific work practices in the projects. Details of the procedures for using and maintaining the technical debt list will depend on the types and formats of the software process data available, and the tools used to manage that data. A simple tool will be implemented to aid in managing the list and to collect data on its use. The list will be kept under version control, along with other project documents. Project personnel will be trained on using the technical debt list. The process of implementation will be closely monitored, including all issues raised and their resolutions.

In the first release cycle, during which the projects will be tracking technical debt for the first time, detailed qualitative and quantitative data will be collected concerning the amount of time project personnel spend using the technical debt list, the kinds of problems and questions that come up, and the usefulness of the approach from the point of view of the project personnel. After the first release, detailed focus groups will be conducted with each 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. This will also provide an opportunity to modify the approach based on feedback. Data will then be collected over successive releases to determine the benefits of the approach over time. These data will include documentation of decisions made concerning technical debt, the principal and interest paid and/or avoided, 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. Much of this data will be qualitative in nature, and will contribute to an open, mixed-methods case study design. All data, both qualitative and quantitative, will be analyzed using an iterative approach involving coding and comparing pieces of evidence in order to build well-grounded propositions. This approach is similar to grounded theory; however, the process will start with well-defined research questions and will not be completely open. The questions guiding the data analysis are:

- What factors contributed to the costs of measuring and monitoring technical debt?
- In what ways did technical debt information contribute to decision making?

A natural outcome of the retrospective and pilot studies will be a set of suggestions for improvement to the management mechanism. However, the challenge here will be to evolve the management mechanism in a way that is general and flexible enough to be valuable in any environment (with proper tailoring, of course). The key to meeting this challenge is to triangulate findings between multiple study settings; that is beyond the single case study that is currently underway. Comparing the evolution of the mechanism in the two settings will allow identification of modifications that are essential to support the core concepts of technical debt, and of those that are needed to support technical debt management in a particular setting. Both types of modifications are necessary, but it is essential to be able to differentiate between the two.

The findings from the retrospective and case studies will be the primary input to the evolution of the management mechanism. As new factors and relationships are identified in the cost–benefit models, these factors and relationships must be incorporated into the management mechanism. Incorporating new concepts could

take the form of additions to the technical debt item template, modifications to the decision procedures, new types of technical debt, modified estimation procedures, or changes to any other part of the management mechanism.

6. Conclusion

The problem of managing the aging of software systems during maintenance is not new. Typically, tight maintenance budgets and schedules inevitably result in modifications that do not fully take quality requirements into account. Using the technical debt metaphor to reason about this phenomenon allows practitioners and managers to fully account for it in planning and decision making. However, there are many obstacles to doing this easily. We have attempted in this chapter to propose a way forward toward using a technical debt lens to facilitate maintenance management. More work lies ahead in validating and refining this approach in practice, as well as broadening the metaphor to include all sources of technical debt.

REFERENCES

- [1] S.M. Dekleva, Software maintenance: 1990 status, *Software Maintenance Res. Pract.* 4 (1992) 15.
- [2] M.J.C. Sousa, H.M. Moreira, A survey on the software maintenance process, in: *International Conference on Software Maintenance*, Bethesda, MD, USA, 1998, pp. 265–274.
- [3] B.P. Lientz, E.B. Swanson, G.E. Tompkins, Characteristics of application software maintenance, *Commun. ACM* 21 (1978) 6.
- [4] M. Fowler, Technical Debt. Available: <http://www.martinfowler.com/bliki/TechnicalDebt.html>, 2003.
- [5] M.M. Lehman, L.A. Belady (Eds.), *Program Evolution: Processes of Software Change*, Academic Press Professional, Inc, San Diego, CA, USA, 1985.
- [6] D.L. Parnas, Software aging, in: *16th International Conference on Software Engineering*, Sorrento, Italy, 1994.
- [7] W. Cunningham, The WyCash Portfolio management system, in: *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, 1992, pp. 29–30.
- [8] S. McConnell, 10x Software Development. Available: <http://forums.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx>, 2007.
- [9] J. Rothman, 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>, 2006.
- [10] J. Shore, Quality with a Name. Available: <http://jamesshore.com/Articles/Quality-With-a-Name.html>, 2006.
- [11] M.J. Carr, S. Konda, I. Monarch, C.F. Walker, F.C. Ulrich, *Taxonomy-Based Risk Identification*, Software Engineering Institute, Pittsburgh, 1993, CMU/SEI-93-TR-6.
- [12] R. Fairley, Risk management for software projects, *IEEE Software* 11 (1994) 57–67.
- [13] C. Chapman, S. Ward, *Project Risk Management: Processes, Techniques and Insights*, first ed., John Wiley & Sons, West Sussex, UK, 1996.

- [14] Airmic, A risk management standard, available at http://www.theirm.org/publications/documents/Risk_Management_Standard_030820.pdf, 2002.
- [15] D. Hillson, D. Hulett, Assessing risk probability: alternative approaches, in: Proceedings of PMI Global Congress, Prague, Czech Republic, 2004.
- [16] P.E. Engert, Z.F. Lansdowne, Risk Matrix User's Guide, The Mitre Corporation, Bedford, MA, USA, 1999. Available at <http://www.mitre.org/work/sepo/toolkits/risk/ToolsTechniques/files/UserGuide220.pdf>.
- [17] J. Ansell, F. Wharton, Risk: Analysis, Assessment and Management, John Wiley & Sons, Chichester, 1992.
- [18] I. Sommerville, Software Engineering, eighth ed., Addison-Wesley, Harlow, UK, 2007.
- [19] S. McConnell, Software Project Survival Guide, Microsoft Press, Redmond, WA, USA, 1998.
- [20] B. Kitchenham, L. Pfleeger, Software quality: the elusive target, IEEE Software 13 (1996) 12–21.
- [21] J. Tian, M. Zelkowitz, Complexity measure evaluation and selection, IEEE Trans. Software Eng. 21 (1995) 641–650.
- [22] K. El Emam, et al., The confounding effect of class size on the validity of object-oriented metrics, IEEE Trans. Software Eng. 27 (2001) 630–650.
- [23] G. Koru, T. Jeff, Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products, IEEE Trans. Software Eng. 31 (2005) 625–642.
- [24] M. Fowler, et al., Refactoring. Improving the Design of Existing Code, Addison Wesley, Boston, MA, USA, 1999.
- [25] M. Lanza, et al., Object-Oriented Metrics in Practice, Springer, Berlin, 2006.
- [26] Y. Guo, et al., Domain-specific tailoring of code smells: an empirical study, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, 2010, Vancouver, Canada, pp. 167–170.
- [27] E. van Emden, L. Moonen, Java quality assurance by detecting code smells, in: 9th Working Conference on Reverse Engineering, Richmond, VA, USA, 2002.
- [28] B.W. Boehm, Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [29] I.I. Standard, ISO-20926 Software Engineering—Unadjusted Functional Size Measurement Method—Counting Practices Manual, 2003.
- [30] A. Stellman, J. Greene, Applied software project management, 2005.
- [31] M. Jørgensen, Estimation of software development work effort: evidence on expert judgment and formal models, Int. J. Forecasting 23 (2007) 449–462.
- [32] M. Jørgensen, A review of studies on expert estimation of software development effort, J. Systems and Software, 70 (2004) 37–60.
- [33] W.G. Lutters, C.B. Seaman, The value of war stories in debunking the myths of documentation in software maintenance, Inform. Software Technol. 49 (2007) 12.