# Managing technical debt in software-reliant systems

**Conference Paper** · January 2010

DOI: 10.1145/1882362.1882373 · Source: DBLP



**14 authors**, including:

Rick Kazman
Carnegie Mellon University
**333** PUBLICATIONS **16,904** CITATIONS
SEE PROFILE

Philippe Kruchten
University of British Columbia - Vancouver
**290** PUBLICATIONS **13,738** CITATIONS
SEE PROFILE

Robert L. Nord
Carnegie Mellon University
**133** PUBLICATIONS **5,820** CITATIONS
SEE PROFILE

Ipek Ozkaya
Carnegie Mellon University
**98** PUBLICATIONS **2,647** CITATIONS
SEE PROFILE

# Managing Technical Debt in Software-Reliant Systems

Brown, N.*, Cai, Y.**, Guo, Y.[£], Kazman, R.*[,¥], Kim, M.[+], Kruchten, P.[#], Lim, E.[#], MacCormack, A.[++], Nord, R*., Ozkaya, I.*, Sangwan, R.[π], Seaman, C.[£,β], Sullivan, K.[α*], Zazworka, N.[£,β]

*Carnegie Mellon Software Engineering Institute, **Drexel University, [£]University of Maryland Baltimore County, [¥]University of Hawaii, [+] University of Texas at Austin, [#] University of British Columbia, [++] MIT, [π]Penn State University, [α] University of Virginia, [β] Fraunhofer Center,

## ABSTRACT

Delivering increasingly complex software-reliant systems demands better ways to manage the long-term effects of short-term expedients. The *technical debt* metaphor is gaining significant traction in the agile development community as a way to understand and communicate such issues. The idea is that developers sometimes accept compromises in a system in one dimension (e.g., modularity) to meet an urgent demand in some other dimension (e.g., a deadline), and that such compromises incur a "debt": on which "interest" has to be paid and which the "principal" should be repaid at some point for the long-term health of the project. We argue that the software engineering research community has an opportunity to study and improve this concept. We can offer software engineers a foundation for managing such trade-offs based on models of their economic impacts. Therefore, we propose *managing technical debt* as a part of the future research agenda for the software engineering field.

## Categories and Subject Descriptors

D.2.9 **[Software Engineering]** Management—life cycle, productivity.

## General Terms

Management, Measurement, Design, Economics

## Keywords

Technical debt, large-scale system development, cost-benefit analysis, software metrics, design decision trade-off

## 1. INTRODUCTION

Software developers and corporate managers frequently disagree about important decisions regarding how to invest scarce resources in development projects, especially in relation to internal quality aspects that are crucial to system sustainability, but that are largely invisible to management and customers, and that do not generate short-term revenue. Among these properties are design and code quality and documentation.

Engineers often advocate for such investments, but executives question their value and frequently decline to approve them, to the long-term detriment of software projects. The situation is exacerbated in projects that must balance short deadlines with long-term sustainability.

Cunningham coined the *technical debt* metaphor in his 1992 OOPSLA experience report [4] to describe a situation in which long-term code quality is traded for short-term gain, creating future pressure to remediate the expedient.

> *Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite…The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.*

To date, technical debt has been used as a metaphor and rhetorical device within the agile community with increasingly recognized utility for technical communication and for communication between engineers and executives. The technical debt concept is gaining traction as a way to focus on the long-term management of accidental complexities created by short-term compromises. Effective management of such debt is perceived as critical to achieving and maintaining software quality. Left unmanaged, such debt creates significant long-term problems, such as increased maintenance costs.

The metaphor highlights that, like financial debt, technical debt incurs interest payments in the form of increased future costs owing to earlier *quick and dirty* design and implementation choices. Like financial debt, sometimes technical debt can be necessary. One can continue paying interest, or pay down the principal by re-architecting and refactoring to reduce future interest payments.

Agile practices of refactoring [8], test-driven development [6], iteration management [3], [10], [21] and software craftsmanship [16], along with an intuitive understanding of technical debt, are felt to be sufficient to manage technical debt on small-scale projects; a more rigorous definition and validation of the concept and the heuristic practices it implies has not been undertaken. Agile software development methods, such as XP, Scrum, FDD (Feature Driven Development), Lean Software development, have had significant impact over the last 5-8 years on industrial software development practices. While scaling agile techniques have gained attraction [14], techniques for managing technical debt have not emerged out of such efforts.

The risk is that the needs of the technical community for effective solutions to the managing technical debt problem will lead to the adoption of intuitively attractive but sub-optimal heuristics. Research that illuminates the strengths and weaknesses of the



FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

metaphor and that develops the definitions needed for quantitative modeling and management of technical debt are lacking. In addition, while the term was originally coined in reference to coding practices, today the metaphor is applied more broadly across the project lifecycle and may include architectural [2], testing [20], or documentation debt.

The research community can play a constructive role by helping practitioners to put this resonating idea on firm foundation. The metaphor already communicates critical issues in large-scale, long- term projects:

- There is an optimization problem where optimizing for the short-term puts the long-term into economic and technical jeopardy when debt is unmanaged.
- Design short-cuts can give the perception of success until their consequences start slowing projects down.
- Development decisions, especially architectural ones, need to be actively managed and continuously analyzed quantitatively as they incur cost, value, and debt.

Yet, the concept as developed to date leaves many questions open:

- Is *debt* a sound metaphor for managing expedients and remediative investments in software projects? If not, is there a closely related metaphor that is better?
- Can the debt metaphor or an alternative lead to useful, testable theories about how to use, measure, and pay-off software short-cuts?
- How can one identify debt in a software development project and product, perhaps automatically?
- What are the kinds of debt? What techniques can help projects elicit, communicate, analyze, and manage it?
- How is technical debt related to evolution and maintenance?
- How can information about technical debt be collected empirically for developing conceptual models?
- How can technical debt be visualized and analyzed?

On June 2-3, 2010, the authors of this paper met at the Software Engineering Institute with the purpose of discussing research on managing technical debt in large-scale systems. The goal of the workshop was sharing work-in-progress and early results, developing a common understanding and conceptual model of technical debt, identifying gaps in knowledge, and refining a research agenda. This research agenda statement was a key output of the workshop. We propose that the research community take on the challenge of assessing the metaphor, and developing validated theory and practices for the management of technical and economic tradeoffs with technical and executive decision-makers as stakeholders.

# 2. THE TECHNICAL DEBT METAPHOR

Technical debt is one dimension of a larger valuation process, a lens through which to observe a financial strategy view of software development. This requires a means of assessing opportunities for creating value in software in terms of the cost of those endeavors relative to other endeavors that might be considered (some of which may not be technical).

Here is a typical real-world example. A large web project was developed over more than two years, primarily by one developer. The company had guidelines for deploying a common architecture across all projects and for accessing a company-owned web library (that stored often-used code). Developers were advised to document their work by creating in-code and API style documentation. The visible, external quality of the product was above average (e.g., number of post-delivery defects detected by customers, number of implemented features per iteration). When the main developer left one day, the developers taking over found that the source code had greatly drifted from the common architecture. The result was that the implementation of new features constantly led to defects due to the mismatch between the developers' understood architecture (based on the common architecture) and the actual, implemented architecture of the product. This previously undetected "debt" is currently being paid off by putting enormous amounts of effort into understanding and gradually refactoring the system towards the common architecture (while at the same time implementing new features.)

One way to understand technical debt is as a way to characterize the gap between the current state of a software system and some hypothesized "ideal" state in which the system is optimally successful in a particular environment. This gap includes items that are typically tracked in a software project, such as known defects and unimplemented features. But it also includes less obvious and less visible aspects, such as architectural and code decay and outdated documentation. While the metaphor is broad enough to encompass all of these concepts, the discourse around technical debt has emphasized the latter category, because those issues tend to be ignored or discounted by decision-makers when considering how to invest developer time.

## 2.1 Properties of Technical Debt

For purposes of both defining the concept and characterizing types of technical debt, it is helpful to refer to a number of properties of technical debt. While a complete set of technical debt properties is a subject of research, a number of properties are already becoming clear.

- *Visibility*. Significant problems arise when debt is not visible. In many cases, it is (or was) known to some people (e.g., I know that I broke encapsulation to implement a feature before the deadline) but it is not visible enough to others who eventually have to pay for it. A purpose of research in this area is to find ways to ensure that technical debt achieves adequate visibility so that it can be considered in system-level decision-making processes.
- *Value*. In its financial use, debt when managed correctly is a device to create value (e.g., having a mortgage enables owning a house.) The value is the economic difference between the system as it is and the system in an ideal state for the assumed environment. The attributes that enable such a valuation in software are difficult to elicit.
- *Present value*. In addition to the overall potential system value enabled by technical debt, the present value of the costs incurred as a result of the debt, including the time-to-impact and uncertainty of impact, must be mapped to the overall cost-benefit analysis.
- *Debt accretion*. Debt does not necessarily combine additively, but super-additively in the sense that taking on too much debt leads a system into a bad, perhaps irreparable state (e.g., of code complexity).

- *Environment*. In software engineering projects, debt is relative to a given or assumed environment.
- *Origin of debt*. It is important to distinguish sharply between strategic debt, taken on for some advantage, and unintentional debt, that is taken on either through poor practices or simply because the environment changed in a way that created a mismatch that reduces system value.
- *Impact of debt*. The locality (or lack thereof) of debt is important: are the elements that need to be changed to repay a debt localized or widely scattered?

As the properties imply, while the metaphor is compelling, it does not cover all aspects of the software development lifecycle. There are several sources of uncertainty in managing technical debt. Not all technical debt results in the obligation to be paid off. Rather, some technical debt appears to create opportunities to invest without obligations, that is, options. This phenomenon manifests itself in legacy systems where the system has incurred technical debt (it is not meeting new needs), but the organization chooses to leave it as is because the value of changing the system is less than the cost of those changes. Moreover, if an organization decides to pay back the debt, it is often impossible to determine upfront either the principal or the interest rate, in other words a payment strategy.

In addition, financial debt occurs as a result of deliberate action of borrowing; one *deliberately* incurs debt. But in software, technical debt can arise due to changes in environmental factors that are out of the development team's control even if good decisions may have been made. If the system does not evolve, then new environmental conditions may start creating high interest payments.

## 2.2 Related work

Technical debt metaphor to date mostly has been used as a communication device. Most writing about the concept has not been in the research literature, but in blogs and essays. The concept is, however, tied to several subjects that have been topics of research in iterative and incremental software development, software maintenance and evolution contexts for some time.

Steve McConnell [18] and Martin Fowler [7] categorize technical debt into distinct types, separating issues arising from recklessness from those decisions that are made strategically (Figure 1).

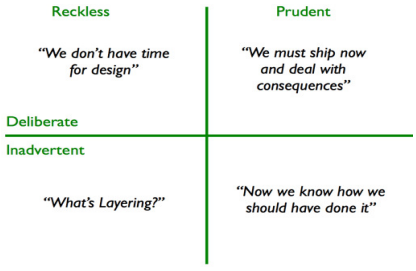


**Figure 1: Technical Debt [7]**

Evolution of a software system is no longer restricted to the maintenance phase of development, but now manifests itself during system elaboration and construction. Technical debt can be observed based on the structure of the system. Structural complexity is often inadvertently introduced as the number of dependencies between parts of the system grows and design goals are violated. Such increasing complexity makes systems rigid (hard to change), fragile (each change breaks something else), viscous (doing things right is harder) and opaque (hard to understand).

There is a significant body of research in software maintenance and evolution (see Mens [17] for a recent review of research and historical foundations.) Technical debt especially resonates with maintenance activities when it needs to be repaid, especially with refactoring and re-architecting. Lehman and Belady [15] postulate that for systems to remain useful they must change and that change will increase their complexity leading to *software decay* if refactoring is not done as needed. Parnas [19] calls this phenomenon *software aging* reflecting the failure of a product owner to modify it to meet changing needs. Engineers must, therefore, learn to track, manage, and mitigate software complexity since, left unchecked, it can lead to systems that are difficult to maintain and evolve.

Clone detection [12] and inferring systematic change patterns [13] are concerns relevant to code-level technical debt. Yet, again, in the context of large-scale, long-term projects, there is distinction between code-level and architecture-level abstractions, especially when it comes to relating these to a global concept such as debt.

# 3. OPEN RESEARCH QUESTIONS

At our workshop the participants shared many ideas for future research. As we engaged in defining technical debt, the scope of relevant issues expanded to include a variety of research problems. There is room for contribution from researchers with expertise in many areas of software engineering.

## 3.1 Refactoring opportunities

When the software functions correctly, we commonly depend on developers' experience and intuition to detect debt as maintenance becomes burdensome. Several research projects have investigated a number of ways to identify such problematic design. For example, researchers have implemented bad code smell detection analyses that find symptoms of poor design [8]. The challenge of such refactoring opportunity identification research is that it is difficult to evaluate the outcome of refactoring suggestions, even in a retrospective analysis setting. Even if the suggestion was accepted and implemented, it can be difficult to quantitatively assess the impact.

## 3.2 Architectural issues

Refactoring is the restructuring of an existing body of code, altering its internal structure without changing its external behavior. When significant architectural change is needed, small, local refactoring efforts cannot compensate for the lack of a coherent system-wide architecture. Currently, research looks at how to do tradeoff analysis based on architecturally significant requirements; however, typically, once decisions have been made they are not monitored throughout the life of the project and not related to code artifacts, resulting in architecture-level technical debt. As a result, evaluating if and how to re-architect to pay the debt down typically becomes based on subjective-criteria. Monitoring and managing technical debt in the architecture would provide analyses earlier in the development cycle for keeping the project on track. Some of can be used today. For example, analyzing and monitoring architecture violations based on dependency analysis in an ongoing effort to evolve and improve the architecture have been employed at L.L. Bean [11].

## 3.3 Identifying dominant sources of debt

There are many potential sources of technical debt at any time in any given system. It would be useful to know, for a given context, what sources of technical debt are most numerous, costly, or useful. Field studies of development practice and elicitation from practitioner experts are fruitful directions for research to reveal the relative importance of different sources. A related research question is to understand the dynamics of how debt is incurred, viewed, and resolved. Specifically, there appears to be a key difference between bad engineering practices that result in debt, and intentional strategic decisions that require incurring debt. Our role as researchers is to develop this metaphor, based on rigorous empirical examination of current development practices.

## 3.4 Measurement issues

Measurement of technical debt is difficult, and there are many areas for research on this issue. Each type of technical debt has associated with it various techniques for quantitative characterization. For example, the extent of code clones might be a useful measure of design debt. The number of TBDs in the requirements document might be a useful measure of another type of debt. Designing and validating measures at this level is an important area of research. However, these individual measures of technical debt need to be combined into a form useful for decision-making. Following the metaphor, it is useful to aggregate these individual, technical measures into several metaphor-specific concepts:

- Principal – given a particular type of technical debt, the estimated cost of eliminating that debt (e.g., testing, refactoring.)
- Interest probability – the probability that a particular type of technical debt will in fact have visible consequences (e.g., how likely it is that a defect exists in the untested part, or how likely the code in need of refactoring will have to be modified.)
- Interest amount – the added cost of performing maintenance on the part of the system that contains technical debt (e.g., the cost of fixing a defect when it is discovered by a customer as compared to earlier when it would have been detected if testing had been completed, or the extra cost of modifying a component in need of refactoring as compared to the cost of modifying it after refactoring.)

## 3.5 Non-code artifacts

There are also research questions related to non-code artifacts, particularly design artifacts, testing, and requirements documents. There can be sources of technical debt in any of these artifacts. For example, testing debt occurs when a test plan is not completely carried out. If a part of a system does not undergo all the testing that was intended, there is a risk that defects remain that would otherwise have been detected. Examination of a test plan against test results would reveal such debt. Related issues exist in requirements documents, for example the relationship between numbers of TBDs and maintenance costs or stability. Design artifact-related technical debt includes architectural issues, discussed earlier, as well as issues of updating and completeness.

Research into the feasibility of characterizing technical debt, or finding other ways to characterize it (e.g., using other parts of the financial metaphor, such as investment strategy), in the context of supporting decision making, is a research problem.

## 3.6 Monitoring

A little technical debt may not be a problem, but it becomes a problem when there is "too much" debt. This implies that there must be some rules about what "too much" debt looks like such as acceptability thresholds. An existing technical debt visualization plug-in demonstrates how to monitor coding rules violations and providing measures using debt heuristics [9]. It is envisioned that such thresholds are not simple numbers, but instead are complex decision processes. This implies methods for determining the level of technical debt over time, recognizing trends, and disseminating warnings at appropriate times. Such monitoring must be integrated into the development and management environment through appropriate tooling.

## 3.7 Process issues

Technical debt has real, and often significant, economic costs. A reason that technical risks are inadequately understood and managed today is that we lack rigorous ways of quantifying the present values of these costs. These costs in turn create opportunities for investments to remediate the technical debt. Even more challenging is how to value these investment opportunities, so that decisions can be made about which opportunities to exploit: for example, should we refactor, or should we implement another feature? Effective management of technical debt demands a rational basis for making investment decisions. The questions that arise are: Should we limit feature delivery and do a major refactoring? Should we invest in architecture? How much refactoring is enough? To ensure that the technical debt concept leads to well-founded practices, research is needed to enable valuation of the liability created by technical debt and of projects launched to remediate it. Such techniques in turn require ways of making technical debt explicit and are subject to tracking and management within defined development processes.

# 4. CONCLUSION

Technical debt recasts a technical concept as an economic one. Cost, benefit, and value aspects of software development have begun to be addressed as a part of the value-driven software engineering agenda in the broad [1], [5], but have not yet culminated in rigorous analysis models and research directions for large-scale projects. Two important questions about any proposed research initiative are the following: (1) what will be different if the research is successful, and (2) who will care, and why?

The vision emerging in the research community is that developers, architects, managers, and other stakeholders will have ready access to explicit representations of the technical debt in a given project as aids to project decision-making. Such representations could, for example, be presented through software project dashboards.

The impact of this research, if it succeeds, will be improved software development productivity and quality. Software developers and managers will better reason about the liabilities and opportunities created by technical debt and make better decisions about managing them. Software engineers would understand the rationale that managers use in making such decisions. This will lead to improved software maintenance and, in the end, better software. Finally, software tool developers will have a new set of functions to support and new markets for their tools based on a coherent framing of the issues.

## 5. ACKNOWLEDGMENTS

The participation of Dr. Seaman, Mr. Guo, and Mr. Zazworka in this work is supported by the NSF under grant # 0916699.

Kevin Sullivan's work in this area is supported in part by NSF CCF under grant # 0613840.

The Software Engineering Institute is an FFRDC sponsored by the US Department of Defense.

## 6. REFERENCES

[1] Biffl, S., Aurum, A., Boehm, B., Erdogmus, H. and Grunbacher, P. 2005. *Value-Based Software Engineering*. Springer, Berlin.

[2] Brown, N., Nord, R., Ozkaya, I. 2010 Enabling Agility through Architecture, *Crosstalk* Nov/Dec 2010.

[3] Cohn, M. 2006 *Agile Estimation and Planning*, Prentice Hall.

[4] Cunningham, W. 1992. The WyCash Portfolio Management System. *OOPSLA'92 Experience Report*.

[5] Denne, M., & Cleland-Huang, J. 2004. *Software by Numbers: Low-Risk, High-Return Development*. Upper Saddle River, N.J.: Prentice Hall.

[6] Erdogmus, H., Morisio, M., and Torchiano, M. 2005. On the Effectiveness of the Test-First Approach to Programming. *IEEE Trans. Softw. Eng*. 31, 3 (Mar. 2005), 226-237.

[7] Fowler, M. *Technical Debt Quadrant*. Bliki [Blog] 2009. [cited 2010 June 14]; Available from: http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html.

[8] Fowler, M. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.

[9] Gaudin, O. 2009. *Evaluate your technical debt with Sonar*. [cited 2010 June 14]; Available from: http://www.sonarsource.org/evaluate-your-technical-debt-with-sonar/

[10] Highsmith, J. 2009. *Agile Project Management 2nd ed*. Addison Wesley.

[11] Hinsman C., Sangal, N., Stafford, J. 2009. Achieving Agility Through Architecture Visibility, in *LNCS 5581/2009, Architectures for Adaptive Software Systems*, pp.116-129.

[12] Kamiya, T.; Kusumoto, S. & Inoue, K. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE TSE,* 2002*, 28.*

[13] Kim, M. and Notkin, D. 2009. Discovering and representing systematic code changes. In *Proc. ICSE 31*.

[14] Leffingwell, D. 2007. *Scaling Software Agility*, Boston: Addison-Wesley.

[15] Lehman, M.M. and Belady, L.A. (eds) 1985. *Program evolution: processes of software change*. Academic Press Professional, Inc.

[16] Martin, Robert C. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship*. Addison Wesley.

[17] Mens, T. 2008. Introduction and Roadmap: History and Challenges of Software Evolution in *T. Mens, S. Demeyer eds. Software Evolution*, Springer, 2008, pp. 1-11.

[18] McConnell, S. 2007. *Technical Debt*. 10x Software Development [cited 2010 June 14]; http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx.

[19] Parnas, D.L. 1994. Software Aging, *Proc. ICSE 16*.

[20] Rothman, J. *An Incremental Technique to Pay Off Testing Technical Debt*. [Weekly Column] 2006 [cited 2010 February 9]; Available from: http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=11011&tth=DYN&tt=siteemail&iDyn=2.

[21] Sutherland, J. 2005. Future of Scrum: Parallel Pipelining of Sprints in Complex Projects. *Proceedings of the Agile 2005*, pp. 90-102.