Technical Debt: From Metaphor to Theory and Practice

Philippe Kruchten, University of British Columbia, Vancouver

Robert L. Nord and Ipek Ozkaya, Software Engineering Institute

THE METAPHOR OF technical debt in software development was introduced two decades ago by Ward Cunningham[1] to explain to nontechnical product stakeholders the need for what we call now "refactoring." It has been refined and expanded since, notably by Steve McConnell in his taxonomy,[2] Martin Fowler with his four quadrants,[3] and Jim Highsmith and his colleagues from the Cutter Consortium with their model of the impact of technical debt on the total cost of ownership.[4]

From the original description—"not quite right code which we postpone making it right"[1]—various people have used the metaphor of technical "debt" to describe many other kinds of debts or ills of software development, encompassing broadly anything that stands in the way of deploying, selling, or evolving a software system or anything that adds to the friction from which software development endeavors suffer: test debt, people debt, architectural debt, requirement debt, documentation debt, or just an amorphous, all-encompassing software debt.[5] Consequently, the concept of technical debt in software development has become somewhat diluted lately. Is a new requirement, function, or feature not yet implemented "requirement debt"? Do we call postponing the development of a new function "planning debt"? The metaphor is losing some of its strength.

Furthermore, once we identify tools such as static code analyzers to assist us in identifying technical debt, there's a danger of equating it with whatever our tools can detect. This approach leads to leaving aside large amounts of potential technical debt that's undetectable by tools, such as structural or architectural debt or technological gaps. Gaps in technology are of particular interest because the debt incurred

See www.computer.org/software-multimedia for multimedia content related to this article.

0740-7459/12/$31.00 © 2012 IEEE

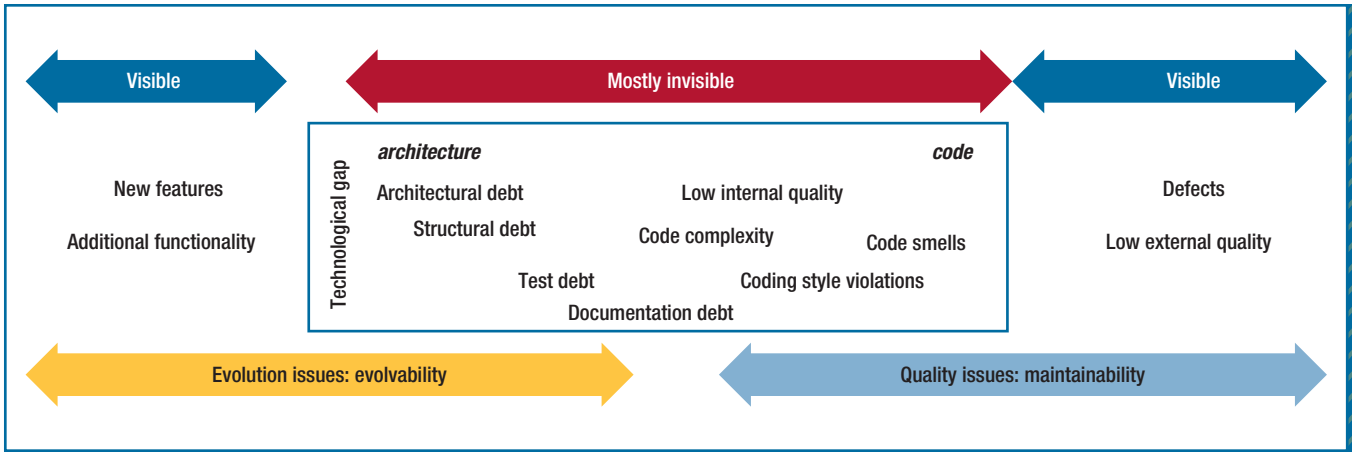

FIGURE 1. The technical debt landscape. On the left, evolution or its challenges; on the right, quality issues, both internal and external.

isn't the result of having made a wrong choice originally, but rather the result of the context's evolution—the passing of time—so that the choice isn't quite right in retrospect. Technical debt in this case is due to external events: technological obsolescence, change of environment, rapid commercial success, advent of new and better technologies, and so on—in other words, the invisible aspects of natural software aging and evolution. You could even argue that "gold plating" an architectural design, making the system more flexible and adaptable than it actually needs to be, can be a form of technical debt, if this added flexibility hinders future development without actually being exploited.

Organizing the Technical Debt Landscape

To make some progress, we need to go beyond debt as a "rhetorical concept."[6] We need a better definition of what constitutes technical debt and some perspective or viewpoints that let us reason across a wide range of technical debt. In short, we need a theoretical foundation.

Figure 1 shows a possible organization of a technical debt landscape—or rather, of software improvement from a given state. We can distinguish visible elements such as new functionality to add and defects to fix, and the invisible elements (or rather, those visible only to software developers). We can see that on the left, we're dealing primarily with evolution or its challenges, whereas on the right, we're dealing with quality issues, both internal and external. We propose to limit debt to the invisible elements—that is, to the elements in the rectangular box, including the invisible aspects of evolution and quality.

Tackling Technical Debt

Most authors agree that the major cause of technical debt is schedule pressure. However, on the right side of the picture, when debt is associated with quality and maintainability issues, other causes become probable, such as carelessness, lack of education, poor processes, nonsystematic verification of quality, or basic incompetence.

Because they use an iterative development process, many agile teams seem to believe that they're completely immune to technical debt. Although iterations offer the opportunity to reimburse debt in a timely fashion, the opposite often occurs. Developing and delivering very rapidly, with no time for proper design or to reflect on the longer term, and a lack of rigor or systematic testing (including automated testing) leads some agile projects into massive amounts of debt very rapidly. In fact, such debt can mount much more quickly than in any old-fashioned waterfall-like project. But in the end, it's all a matter of choice: where time to market is essential, the debt might actually be a good investment, but it's imperative to remain aware of this debt and the increased friction it will impose on the development team, as Cunningham suggested.[1]

So how can we tackle technical debt, or at least avoid accumulating too much of it? The first step is awareness: identifying debt and its causes. The next step is to manage this debt explicitly, which involves listing debt-related tasks in a common backlog during release and iteration planning, along with other "things to do."[7] Figure 2 illustrates how these elements might be organized in a backlog.[8] The element areas' colors reconcile four types of possible improvements—the tasks to attend to in the future to increase value, such as adding new features (green) or investing in the architecture (yellow), and to reduce the negative effects on value of defects (red) or technical debt (black).

Project backlogs often contain only

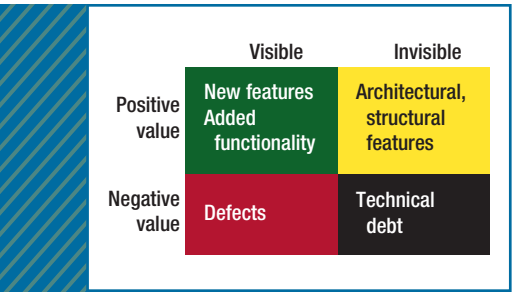

FIGURE 2. Four colors in a backlog. The element areas reconcile four types of possible improvements—the tasks to attend to in the future to increase value, such as adding new features (green) or investing in the architecture (yellow), and to reduce the negative effects on value of defects (red) or technical debt (black).

the green elements; a few technical practitioners keep in mind the yellow elements. The red elements appear elsewhere, perhaps in a defect database, and the black elements are nowhere to be found but they increasingly cripple the development, reducing velocity.

It's important to keep in mind, however, that technical debt is not only about code and code quality. Code analysis tools will identify a small number of the black elements. Therefore, code analysis tools aren't sufficient for identifying technical debt: more often than not, technical debt isn't related to code and its intrinsic qualities but to structural or architectural choices or to technological gaps. No tool will reveal that, two years ago, the team should have used some tool to internationalize and localize the code.

Architecture plays a significant role in the development of large systems, together with other development activities, such as documentation and testing (which are often lacking). These activities can add significantly to the debt and thus are part of the technical debt landscape in Figure 1. Code analysis will only tackle the right side of the box. Professionalism, diligence, dedication, and craftsmanship will certainly help, but they aren't the key determinants in reducing technical debt.

A Unified Theory?

Kevin Sullivan suggested that a simple model for tackling technical debt represents a software development endeavor as a sequence of changes, most of them improvements.[9] At a given point in time, the past set of changes is what defines the current state of the software. Some of these past changes are the events that triggered the current debt: the change or the way it was implemented isn't quite right from the current perspective.

The main issue facing the software development organization is how to decide about future changes: What evolution should the software system undergo, and in which sequence? This evolution is, in most cases, constrained by cost: the resources available to apply to making these changes, most likely driven by value, as viewed by external stakeholders.

The decision-making process about which sequence of changes to apply could be the main reconciling point across the whole landscape shown in Figure 1, from adding new features and adapting to new technologies to fixing defects and improving the quality, intrinsic or extrinsic. Because this decision process is about balancing cost and value, perhaps economic or financial models could become the unifying concept behind the whole landscape. A few have already been explored to some degree:

- Net Present Value (NPV) for a product, from the finance world;
- opportunity cost;
- real option analysis (ROA), or valuation; and
- total cost of ownership (TCO) for an IT system.

These four models were discussed in a recent ICSE workshop on technical debt,[9] with one of them (NPV) offering the most promise: it's better formalized than opportunity cost and simpler and less proprietary than TCO, while ROA can be seen as a probabilistic extension to NPV. TCO presents the danger, mentioned earlier, of diluting technical debt by introducing elements not related to software development (deployment, operations, and support).

Technical debt shouldn't be treated in isolation from adding new functionality or fixing defects, even though these aren't included in the definition of debt presented here. The challenge is in expressing all software development activities in terms of sequences of changes associated with a cost and a value (over time). These changes aren't independent, unfortunately. Their interdependencies play a big role—as Mark Denne and Jane Cleland-Huang have shown, in particular, visible features depend on less visible architectural aspects.[10]

In this new perspective, a system's technical debt at a given point in time could be defined as deferred investment opportunities or poorly managed risks.

In This Issue

This installment of *IEEE Software* gives readers different illustrations of the multifaceted concept of technical debt. Erin Lim, Nitin Taksante, and Carolyn Seaman went out into industry to check how software developers actually conceptualize, perceive, experience, and manage technical debt. They report their results in "A Balancing Act: What Software Practitioners Have to Say about Technical Debt." Their analysis describes the large and complex trade space of stakeholders' short- and long-term concerns and their strategies to keep them in balance.

Raja Bavani complements this view from the trenches with interviews of two agile experts, Johanna Rothman and Lisa Crispin, in "Distributed

Teams, Agile Testing, and Technical Debt." He then goes on to offer his own taxonomy of technical debt and how it relates to testing.

Bill Curtis, Jay Sappidi, and Alexandra Szynkarski explore the viability of an estimation framework for detecting technical debt using real-world data. They use the code analysis toolkit developed by CAST Software to identify technical debt in large systems, based on structural quality data, and literally put a price on it in "Estimating the Principal of an Application's Technical Debt."

As an alternative, Jean-Louis Letouzey and Michel Ilkiewicz describe "Managing Technical Debt with the SQALE Method." The SQALE approach is based on an analysis of an application's source code, using the indicators of quality attributes defined by the ISO systems and software quality standard (testability, maintainability, portability, and so on) to narrow down the point of focus.

Have we outgrown the financial debt metaphor? Does it still work? Do we misuse it? Israel Gat and Christof Ebert disagree on this topic in a point/counterpoint article.

We hope to keep this debt metaphor useful by confining it to what is really a debt—namely, the invisible result of past decisions about software that negatively affect its future—and by not extending the concept to anything that has a cost. From a practical perspective, we hope to see more tools and methods to identify and manage debt, covering more elements of the landscape. From a theoretical standpoint, we'll see models emerging, very likely rooted in financial theories, such as NPV, out of which better measurements and reasoning about this form of debt can take place in the wider context of software evolution or software improvement.

ABOUT THE AUTHORS

PHILIPPE KRUCHTEN is professor of software engineering at the University of British Columbia in Vancouver, Canada. A founding member of IFIP WG2.10, he conducts research in the software development process and software architecture. Kruchten received his PhD from the École Nationale Supérieure des Télécommunications in Paris. He's a professional engineer in Canada, an IEEE CSDP, and a senior (but not senile) member of the IEEE Computer Society. Contact him at kruchten@ieee.org.

ROBERT L. NORD is a senior member of the technical staff in the Research, Technology, and System Solutions Program at the Software Engineering Institute. He's engaged in activities focusing on agile and architecture at scale and works to develop and communicate effective methods and practices for software architecture. Nord received a PhD in computer science from Carnegie Mellon University and is a distinguished member of ACM. Contact him at rn@sei.cmu.edu.

IPEK OZKAYA is a senior member of the technical staff in the Research, Technology, and System Solutions Program at the Software Engineering Institute. She conducts research in empirical methods for improving software development efficiency and system evolution with a focus on software architecture practices, software economics, and requirements management. Ozkaya received a PhD in computational design from Carnegie Mellon University. She serves on the advisory board of *IEEE Software*. Contact her at ozkaya@sei.cmu.edu.

Acknowledgments

Many thanks to all the participants of the 3rd International Workshop on Technical Debt at ICSE 2012 in Zürich and to our reviewers, Len Bass and Raghvinder Sangwan. This material is based upon work funded and supported by the US Department of Defense under contract number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution.

References

1. W. Cunningham, "The WyCash Portfolio Management System," *Proc. OOPSLA*, ACM, 1992; http://c2.com/doc/oopsla92.html.
2. S. McConnell, "Technical Debt," blog, 2007; http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx.
3. M. Fowler, "Technical Debt," blog, 2009; http://martinfowler.com/bliki/TechnicalDebt.html.
4. I. Gat, ed., "Special Issue on Technical Debt," *Cutter IT J.*, vol. 23, no. 10, 2010.
5. C. Sterling, *Managing Software Debt: Building for Inevitable Change*, Addison-Wesley Professional, 2010.
6. N. Brown et al., "Managing Technical Debt in Software-Intensive Systems," *Proc. Future of Software Eng. Research*, ACM, 2010, pp. 47–52; doi: 10.1145/1882362.1882373.
7. N. Brown, R. Nord, and I. Ozkaya, "Enabling Agility through Architecture," *CrossTalk*, Nov./Dec. 2010, pp. 12–18.
8. P. Kruchten, "What Colour Is Your Backlog?," blog, 2008; http://philippe.kruchten.com/talks.
9. P. Kruchten et al., "Report on the 3rd Workshop on Managing Technical Debt," to be published in *ACM SIGSOFT Software Eng. Notes*, vol. 37, no. 5, 2012; http:www.sigsoft.org/SEN.
10. M. Denne and J. Cleland-Huang, "The Incremental Funding Method: Data-Driven Software Development," *IEEE Software*, vol. 21, no. 3, 2004, pp. 39–47.