Cutter IT Journal

The Journal of Information Technology Management

Vol. 29, No. 3
March 2016

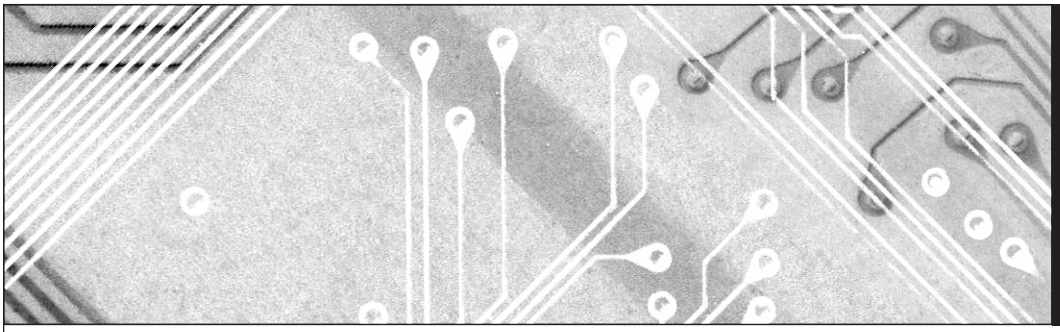
"A major portion of technical debt — if not the vast majority — seems to spring from a very well-understood, well-identified source: time pressures. Not only do time pressures compel developers to cut corners, but they become the chief obstacle to fixing the code."

— Tom Grant, Guest Editor

Technical Debt:

The Continued Burden on Software Innovation

CUTTER CONSORTIUM



Cutter IT Journal

About Cutter IT Journal

Part of Cutter Consortium's mission is to foster debate and dialogue on the business technology issues challenging enterprises today, helping organizations leverage IT for competitive advantage and business success. Cutter's philosophy is that most of the issues that managers face are complex enough to merit examination that goes beyond simple pronouncements. Founded in 1987 as *American Programmer* by Ed Yourdon, *Cutter IT Journal* is one of Cutter's key venues for debate.

The monthly *Cutter IT Journal* and its companion *Cutter IT Advisor* offer a variety of perspectives on the issues you're dealing with today. Armed with opinion, data, and advice, you'll be able to make the best decisions, employ the best practices, and choose the right strategies for your organization.

Unlike academic journals, *Cutter IT Journal* doesn't water down or delay its coverage of timely issues with lengthy peer reviews. Each month, our expert Guest Editor delivers articles by internationally known IT practitioners that include case studies, research findings, and experience-based opinion on the IT topics enterprises face today — not issues you were dealing with six months ago, or those that are so esoteric you might not ever need to learn from others' experiences. No other journal brings together so many cutting-edge thinkers or lets them speak so bluntly.

Cutter IT Journal subscribers consider the *Journal* a "consultancy in print" and liken each month's issue to the impassioned debates they participate in at the end of a day at a conference.

Every facet of IT — application integration, security, portfolio management, and testing, to name a few — plays a role in the success or failure of your organization's IT efforts. Only *Cutter IT Journal* and *Cutter IT Advisor* deliver a comprehensive treatment of these critical issues and help you make informed decisions about the strategies that can improve IT's performance.

Cutter IT Journal is unique in that it is written by IT professionals — people like you who face the same challenges and are under the same pressures to get the job done. *Cutter IT Journal* brings you frank, honest accounts of what works, what doesn't, and why.

Put your IT concerns in a business context. Discover the best ways to pitch new ideas to executive management. Ensure the success of your IT organization in an economy that encourages outsourcing and intense international competition. Avoid the common pitfalls and work smarter while under tighter constraints. You'll learn how to do all this and more when you subscribe to *Cutter IT Journal*.

Cutter IT Journal®

Cutter Business Technology Council: Rob Austin, Ron Blitstein, Tom DeMarco, Lynne Ellyn, Vince Kellen, Tim Lister, Lou Mazzucchelli, Ken Orr, and Robert D. Scott

Editor Emeritus: Ed Yourdon
Publisher: Karen Fine Coburn
Group Publisher: Chris Generali
Managing Editor: Karen Pasley
Production Editor: Linda M. Dias
Client Services: service@cutter.com

Cutter IT Journal® is published 12 times a year by Cutter Information LLC, 37 Broadway, Suite 1, Arlington, MA 02474-5552, USA (Tel: +1 781 648 8700; Fax: +1 781 648 8707; Email: citjeditorial@cutter.com; Website: www.cutter.com; Twitter: @cuttertweets; Facebook: Cutter Consortium). Print ISSN: 1522-7383; online/electronic ISSN: 1554-5946.

Subscription rates are US $485 a year in North America, US $585 elsewhere, payable to Cutter Information LLC. Reprints, bulk purchases, past issues, and multiple subscription and site license rates are available on request.

Opening Statement

by Tom Grant, Guest Editor

THE SOMEWHAT-HIDDEN COST OF RAPID SOFTWARE INNOVATION

One of the defining characteristics of software innovation, as opposed to other types of innovation, is the speed and ease of production. Other than time and labor, the only resources needed are relatively inexpensive hardware and software. Inventing a new type of jet propulsion, industrial ceramic, Alzheimer's treatment, or oil extraction method requires far more expensive equipment and other materials — not to mention more time, skilled personnel, and, in many cases, extra effort to meet regulatory requirements — than building a new social media service.

Usually, this ease of innovation is cause for celebration. The entry requirements into the software market are lower than, say, the automobile market. The speed with which innovation can radically transform the way people work and play is so great that we have come to take it for granted and even judge companies like Apple on the basis of whether they can make every year an *annus mirabilis* of discontinuous product innovation. Even in the more prosaic world of software innovation in corporate IT, we urge companies to go much faster than they are in building and deploying new software systems and criticize them for not seizing the opportunity for profound "digital transformation."

There is a price tag for innovating this quickly and easily. One of the costly line items is technical debt, the increased drag on the ability to do software innovation that arises from a very specific source: failing to code with a proper level of care and diligence. Writing code is much like another procedural exercise — writing laws and regulations. The less care one takes in the process of creating these instructions, the harder it will be to diagnose unintended problems and fix them, or even to build on the existing procedural foundations.[1] Ambiguous intent, excessive complexity, untraceable logical paths, untested scenarios, and other problems are the inevitable result of rushing into "production" both statutes and compiled code. In the case of software, these problems take the form of cyclomatic complexity, indecipherable naming conventions, inadequate unit testing, and other problems.

As Cutter Fellow Ward Cunningham, who devised the concept of technical debt, was quick to point out, all forms of technical debt are not necessarily bad.[2] For example, a software team might want to experiment with a new approach to cloud-based storage to help an application scale. In this case, dotting all the programming i's and crossing all the unit testing t's is not merited. If the experiment fails, the extra work spent on code review, proper naming conventions, and other steps needed to prevent future technical debt would be wasted effort. On the other hand, if the experiment succeeds, it is imperative that the team go back and clean up the code. Failing to do so will make it harder to either build on or fix that code.

There is a price tag for innovating this quickly and easily. One of the costly line items is technical debt, the increased drag on the ability to do software innovation that arises from a very specific source: failing to code with a proper level of care and diligence.

Of course, we would not be talking about technical debt if teams took the time to refactor, as well as other steps needed to "do it right." Temporary measures become permanent, which imposes a burden on the team. The greater the technical debt, the harder it is to work with the code base, slowing down further software innovation.

What stops highly skilled, dedicated professionals from doing it right in the first place? Some technical debt is inadvertent, even unconscious: a developer fresh out of college, for example, might not understand how important clear naming conventions are and may need mentoring on how to avoid some common complexity traps when designing classes. However, a major portion of technical debt — if not the vast majority — seems to spring from a very well-understood, well-identified source: time pressures.

Here is where the ease of software innovation becomes a burden. Not only do time pressures compel developers to cut corners, but they become the chief obstacle to fixing the code. In the face of increasing demand from the business side of the organization (sales, marketing, and executives in software companies; internal customers and executives in IT organizations), which neither understands how software innovation works nor cares about the particulars of writing good code, developers are hard-pressed to justify investing the time in short-term refactoring, even if it has long-term benefits.

In the face of increasing demand from the business side of the organization, which neither understands how software innovation works nor cares about the particulars of writing good code, developers are hard-pressed to justify investing the time in short-term refactoring, even if it has long-term benefits.

GRAPPLING WITH TECHNICAL DEBT

That very cursory description of the problem oversimplifies and overlooks many important details. For instance, we have not gone near the differences between technical debt and other kinds of debt, the different forms of intentional and unintentional technical debt, the specific coding sins that lead to technical debt, or the steps needed to both eliminate current debt and prevent future debt. On that last front, opinions vary widely, from "Run a static code analysis tool once in a while" to "Just don't write poor code." Fortunately, in this issue, we have many good articles that help flesh out these critical details.

Our first article, by Declan Whelan, provides a more complete overview of technical debt. Whelan is one of the cochairs of the technical debt working group for the Agile Alliance, to which I was also a contributor.[3] While he argues that technical debt is more a symptom of larger organizational and team problems than a problem unto itself, he lays out a clear method for diagnosing and addressing technical debt, which he believes is a much larger burden on software innovation than most realize.

Our next contributor, Jean-Louis Letouzey, is the other cochair of this working group and a noted expert on technical debt. Letouzey provides an introduction to a very specific strategy for dealing with technical debt, the SQALE method. In his article, he explains how SQALE measures technical debt at both the system and portfolio levels. While elements of SQALE will be mysterious to some business users, both they and the software teams can look at the output of assessment tools that perform SQALE analysis to see the current amount of technical debt and understand (if only in the rough outlines, for business users) the burden it imposes.

John Heintz, a Cutter Senior Consultant who has helped many clients deal with technical debt problems, uses the cost of change as the starting point for his article. Heintz then lays out a strategy for using technical debt assessment as the basis for better technical and business

decisions and offers examples of companies that have pursued this kind of strategy successfully. Echoing Cunningham's dictum that not all technical debt is necessarily bad, and certainly not all technical debt is worth removing, Heintz cites one example of a vendor that intentionally increased technical debt in the short term because that was the right business decision at the time.

Next, Richard Brenner warns against the risks of using the technical debt metaphor too loosely. In many cases, he argues, people have confused the metaphor with the reality, including situations where people look for objective measures of debt. Like other authors in this issue, Brenner believes that technical debt is a symptom of other maladies in the organization, and he provides a simple list of measures for preventing further technical debt accrual.

Ram Reddy then provides a picture of technical debt that goes beyond the boundaries of a single software system, which is often the implicit basis for describing and assessing technical debt. Not only does technical debt have an impact on other, related systems (e.g., on the applications that depend on a debt-ridden identity management system), but it also appears in the integration code that binds systems together. Technical debt exists throughout the services catalog, including the "shadow IT" systems that the IT organization struggles to bring into the official fold. One of the most damaging effects of technical debt from this IT services catalog perspective, Reddy argues, is the way in which debt locks into place systems that should be candidates for retirement.

In our final article, Mohan Babu K explains that not all technical debt results from developers typing on keyboards. Babu K expands the concept to include the drag on the portfolio and projects created by the failure to upgrade vendor-provided software. Over time, the cost of maintaining, integrating, and extending these systems increases if IT departments do not perform these upgrades. He cites the enterprise architect as someone well positioned to help organizations decide on the priority and timing of upgrades.

THE COST OF TECHNICAL DEBT CONTINUES TO INCREASE

As demonstrated by this very rich menu of perspectives on technical debt and the lively debate that discussions like these inspire, Cunningham struck a nerve with the technical debt concept. Whether you believe that the technical debt problem is larger or smaller than others might think, a symptom or a cause of organizational dysfunction, and/or an absolute or subjective measure of that dysfunction, it is an unintended cost of the ease of software innovation that organizations must take seriously. The sources of technical debt, such as the ever-increasing demand for new software, are not going away. The software landscape continues to grow in complexity, with new devices, frameworks, integrations, and other features. Therefore, we cannot afford to make the job of software innovation any harder than it already is.

ENDNOTES

[1]In a similar vein, musician Frank Zappa once said, "The United States is a nation of laws, badly written and randomly enforced."

[2]To hear Cunningham explain the nature of technical debt and his inspiration for creating the concept, see: http://c2.com/cgi/wiki?WardExplainsDebtMetaphor.

[3]For the complete package of support materials that the working group has provided, see: www.agilealliance.org/resources/initiatives/technical-debt.

Tom Grant is Practice Director of Cutter Consortium's Agile Product Management & Software Engineering Excellence practice and also a member of the Business Technology & Digital Transformation Strategies practice. Dr. Grant is a creative problem solver, bringing his expertise in serious games and collaboration to business and software development organizations to increase team productivity through innovation. His well-rounded experience brings a unique richness to his insights, equipping him to offer practical solutions.

Dr. Grant's expertise in software development and delivery has a particular focus on Agile, Lean, application lifecycle management (ALM), product management, serious games, collaboration, innovation, and requirements. Dr. Grant has contributed to client success in various market segments, including government, manufacturing, and finance, through ALM and Agile assessments; selecting tools and creating metrics for software development and delivery; and creating software vendor, corporate, and product strategies.

Most recently, Dr. Grant was a Senior Analyst at Forrester Research. Previously, he served as VP of Product Management at Xythos, a small collaboration software company, and led product management teams at Oracle and various Silicon Valley firms. He is a widely published author and sought-after speaker at many industry events. Dr. Grant earned his PhD in political science from the University of California at Irvine. He can be reached at tgrant@cutter.com.

Technical Debt: It's Not the *Real* Problem

by Declan Whelan

As an Agile coach and consultant, I hear a lot about technical debt from my clients. In fact, it is a *major* issue for almost all of them. In the 2011-2012 CRASH Report from CAST Software, the remediation cost for technical debt was estimated at $3.61 per line of code.[1] I believe the actual financial impact to be much higher, as the report did not include secondary impacts such as increased time to market or the cost of replacing disgruntled software developers who grew tired of wrangling legacy code and moved on.

If there is rampant technical debt with most software products and services, why don't organizations do something about it? Well, it's complicated. Let's take a step back and look at what we mean by technical debt and how we can measure it. Then let's take a longer look at why technical debt exists and what you can do to address it.

WHAT IS TECHNICAL DEBT?

As most — if not all — of us know, it was Cutter Fellow Ward Cunningham who coined the term "technical debt." Currently, there is an initiative within the Agile Alliance to further define and recommend ways to address technical debt. In a soon-to-be-released white paper, the group states the following:

> When taking shortcuts and delivering code that is not quite right for the programming task of the moment, a development team incurs Technical Debt. This debt decreases productivity. This loss of productivity is the interest of the Technical Debt.[2]

It can be hard to communicate the size and impact of technical debt. If you were to wander into the kitchen of a restaurant and see grease on the floors, flickering fluorescent bulbs, dirty dishes piled up, and mouse droppings everywhere, you would be appalled. You don't need to be a foodie or restaurant owner to recognize that there is something seriously wrong and that it had better improve fast or the restaurant will be in serious trouble. It is difficult for most people to see technical debt in a similar way. I have resorted to printing out large methods or classes and taping them on the wall to make the complexity visible. However, this does not evoke the same visceral reaction or sense of urgency that a filthy kitchen does.

Technical Measures

There are static analysis tools you can use to measure technical debt,[3] such as SQALE,[4] FindBugs, PMD, and Code Climate. These tools can help provide insights, especially when you observe trends in the generated metrics over time (e.g., reduced cyclomatic complexity over time could be a sign that technical debt is decreasing).

Team Measures

Often, technical measures don't provide a complete picture. For example, if there is legacy code rife with technical debt that rarely needs to be changed, then the technical metrics alone may provide an overly pessimistic view. For this reason, I recommend that teams also measure and track responses to the following questions:

- On a scale of 1 to 5, how happy are you working in the code base?
- What percentage of your time do you spend working around technical debt versus working on new features?
- How long do you feel it would take to address the main technical debt in the code?

You can measure these things right away with no infrastructure or tooling costs.

Make Your Technical Debt Visible

However you choose to measure technical debt, it is crucial to find ways to make the measures visible. Perhaps your teams could share technical measures and trend graphs at iteration reviews and planning sessions. Technical debt measures and trends could be "information radiators" in the team rooms.

WHY IS THERE RAMPANT TECHNICAL DEBT?

Technical Debt Is Normal, Isn't It?

While technical debt as a metaphor is clear and strong, like any metaphor it has limitations. Most of us incur financial debt during our lives, and it is often the right thing to do. Most of us could not own a house or a car without incurring debt. So having technical debt, by analogy, seems like a normal state of affairs. No big deal. Shrug. Suck it up!

This attitude leads managers and decisions makers to not take technical debt seriously. Do not fall into this trap. Instead, include technical debt in project and portfolio planning. Include specific mechanisms to measure, track, avoid, and mitigate technical debt.

Technical Abilities

I was a professional programmer and independent consultant for 20 years before I was introduced to Agile methods. The level of craftsmanship I mastered in the following five years of applying Agile technical practices eclipsed everything I had previously learned at university and on the job. The level of craftsmanship in many companies is simply not up to the challenge of building products and services at Internet pace.

When I work with recent graduates, I usually ask them if they have been exposed to any of the Agile practices during their undergraduate work. The answer is almost always disappointing. Something like "We learned about Scrum — you know, the stand-ups, user stories, scrum master, and that stuff. But we never learned about test-driven development or simple design." As a result, the code they write is usually complex and overdocumented. They have learned that they get the best marks when they stop coding the moment their program works and then document the crap out of it. Thus, the seeds of technical debt are born.

Technical Practices

Another trap I have seen organizations fall into is using Scrum without any supporting Agile engineering practices. Scrum provides tools for managing new features. However, there are no explicit mechanisms for managing technical debt work and no guidance on technical practices to deal with technical debt. This is a potent one-two combination that can result in Scrum teams building feature after feature as the technical debt mounts. Furthermore, this accumulation is often invisible because the teams naturally increase their story points to factor in the technical debt so their velocity stays the same. Everything looks fine as the big ball of mud grows and grows. Scrum without strong technical practices is a great way to build massive technical debt.

Silver Bullets

For most developers, working on a greenfield project is nirvana. We don't have to deal with past mistakes and can often start with the coolest new technologies that will help us avoid some of the pitfalls we encountered in earlier projects. However, there are often high expectations of how quickly we can deliver, especially if the new technology choices were framed as more effective than earlier ones.

Most of us could not own a house or a car without incurring debt. So having technical debt, by analogy, seems like a normal state of affairs.

As we start the project, we are learning these new technologies and possibly about new markets and competitors as well. As a result, there will be pressure to deliver that will likely cause us to take on technical debt for the first release. But it's not such a big deal, we tell ourselves, because we rocked the first release.

By the time we get to the second release, there is again pressure to deliver at a rate matching our pace for the first release. This can be difficult, though, as we have already accrued some technical debt. The predictable result is that we likely take on even *more* technical debt.

You can see where this is going. Fast-forward a few years and a few releases, and the project will be mired in technical debt. It will take *forever* to get new features out.

At this point, the organization will take stock of its options. Probably someone will suggest some cool new technology that will address a lot of the shortcomings of the current product. A rewrite will be proposed, and the organization will start the cycle over again. And this new project is doomed to the same fate because the organization has failed to address the root cause of technical debt — which is not taking it seriously in the first place. Very sad. Very wasteful. Very avoidable.

When first starting new products, organizations must take technical debt into account. They need to measure and track technical debt and mercilessly refactor the code on a regular basis.

Projects

Projects are one of the prime contributors to technical debt. Projects are fine for … well … project work. If the nature of your work is relatively short term with a clearly defined end of life, then projects may be a suitable management approach. If instead you build products or services that will require future releases and ongoing support, then projects are likely the worst way to manage this work. This is because every decision will be tuned to meeting the next project deadline. There will be no mechanism to optimize for the long-term sustainability of the product or service. Despite the best intentions of people working on the project, the technical debt will mount, because accruing this debt will not negatively impact the project delivery date. In fact, taking on technical debt is often the best way to achieve project success: "Let's just hack this out, and we'll fix it later." Later never comes, however, and the cycle repeats with the next project.

Trying to fix technical debt by simply fixing the code is like bailing a boat that is taking on water.

To break this cycle, organizations need to shift planning away from projects and toward products and services. This allows planning horizons to extend beyond typical project timelines. The extended timelines enable organizations to staff appropriately for the product timeline. The teams can now make better long-term decisions about technical debt because their focus shifts to the long-term viability of the product or service.

Handoffs

Our current organizational structures reflect an emphasis on functional roles where we place people doing similar work together — as on an assembly line. This may have been a good way to build cars many decades ago, but it is wholly ineffective for responding at Internet pace to the ever-changing context in which information products and services live.

As information flows from customers through the organization's process, information is inevitably lost. As information is lost and the systems are built, the developers will make assumptions or incorrect decisions about how the business works or how the system should operate. Each incorrect assumption or abstraction is embodied as technical debt in the code. In this way, technical debt accumulates around the knowledge loss in organizational handoffs.

Dynamic Inconsistency

In economics, dynamic inconsistency[5] describes how our preferences for certain decisions may change at different points in time. For example:

Scenario A: Choose between:

A1	One apple today
A2	Two apples tomorrow

Scenario B: Choose between:

B1	One apple in a year
B2	Two apples in a year plus one day

While some people might pick A1, almost no one would select B1. However, the scenarios are formally identical. We exhibit dynamic inconsistency if we choose A1 today and in 364 days choose B2 instead.

With technical debt, we know that we should be taking care of it all through the project. However, our bias for dynamic inconsistency can lead us to build new features each iteration with little attention to technical debt. This is "double trouble," because new code must coexist with the existing technical debt, and over time we end up with tangled layers of technical debt. And the interest on the technical debt compounds over time.

ADDRESSING TECHNICAL DEBT

I no longer think of technical debt as a problem. It is a symptom — a symptom of deeper system problems in our organizations. Trying to fix technical debt by simply fixing the code is like bailing a boat that is taking on water. It is likely necessary, but it won't stop the water coming in. We need to find and fix the root causes of the technical debt. There are no silver bullets, but here are some things to consider:

Learn How to Write Cleaner Code

Support teams in improving their craft. Team-based learning is ideal, as the team members can support each other. Invest in technical training and encourage the use of pair and mob programming so that good practices spread and individuals can learn together.

Relentless Refactoring

Technical debt metrics may point to poorly written code, but if that code rarely changes, should we really invest time to make that code better? Our efforts might be better directed to improving code that changes more frequently. A simple tactic popularized by Robert Martin is to apply the Boy Scout Rule:

> Leave the campground cleaner than you found it.[6]

This is a simple and powerful approach that focuses improvements on the areas of code that change the most. Teams could incorporate this rule into their working agreements.

The improvements could be very small, such as renaming a variable or removing an unnecessary comment. Yet these improvements will compound over time.

Planning Technical Debt Work

Some organizations may feel sufficient pain from technical debt that they want to address it more quickly than application of the Boy Scout Rule alone can accomplish. Some teams create technical debt reduction stories and negotiate with their product owners to allocate a percentage of their capacity to technical debt reduction. I have seen teams negotiate for technical debt iterations. One interesting twist on this is to keep a separate backlog of technical debt tasks. Then, when a feature is being developed in that area of the code, pull the technical debt task into the work for that feature. Whatever you choose to do, make that work visible.

Proceed with Caution

Removing technical debt from legacy code that does not have tests can be unsafe. Your teams may need to learn and apply new techniques such as the approaches introduced by Michael Feathers in his classic book *Working Effectively with Legacy Code*.[7] One technique Feathers suggests to extract existing code into new methods and classes (*Extract Method* and *Extract Class* refactorings) to build oases of code that can then be unit tested.

Retrospectives

Each team should run a retrospective focused on exploring technical debt. Have team members provide measures of the technical debt, root causes, and what they can do as a team about the situation. Then ask them to provide a list of obstacles they see to improving the situation and any suggestions they may have for overcoming them. Do the same at the middle management and executive levels. Collate and compare the results and support the teams and management in putting technical debt remediation measures into place.

Form Stable Teams

When teams have an extended responsibility to maintain products beyond project timelines, they will start to make better longer-term decisions, since they will have to live with the consequences. This will have a positive impact on technical debt. As a bonus, people will have more pride in their work, and there will be better onboarding for new team members.

No matter how you decide to tackle technical debt, frame each change as an experiment rather than "rolling out" wholesale changes to the organization.

Avoid Handoffs

Organize teams around the value flow and customer communication channels rather than around team functions. In other words, build teams that reach as close to the customers as possible both in understanding their needs and in delivering the product or service to them. Every handoff is a place where knowledge and value is lost.

Introduce Experiments

No matter how you decide to tackle technical debt, frame each change as an experiment rather than "rolling out" wholesale changes to the organization. Use retrospectives to guide experiments. Form a hypothesis, run the experiment, and measure the results. Most importantly, reflect on the results and let that new knowledge guide your next experiment. Create a backlog of experiments as a Kanban board to focus on sustainable change.

For example:

> **Action:** Provide Agile technical training to three teams.
>
> **Hypothesis:** Agile technical training will have a positive impact on technical debt, as teams will have better insights and skills for managing legacy code. We expect to see team happiness about working in the code improve by 1 point (on a scale of 1 to 5) after applying these techniques for two months.

WRAPPING UP

In 1967 Melvin Conway posited that "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations" (aka Conway's Law).

Technical debt is likewise a reflection of weaknesses in the organization's communication channels. Treat technical debt as a symptom of larger systemic issues within your organization. It is giving you valuable feedback to better understand how to improve your product and service delivery.

Use retrospectives to tease out that feedback in a coherent way across multiple teams and management levels. Measure your technical debt and make the metrics and the debt itself visible. Perform experiments to reduce the impact of technical debt. Rinse and repeat.

ENDNOTES

[1]Sappidi, Jay, Bill Curtis, and Alexandra Szynkarski. "The CRASH Report — 2011/2012 (CAST Report on Application Software Health)." CAST Software, 2012.

[2]Letouzey, Jean-Louis, et al. "Technical Debt Initiative." Agile Alliance, June 2015 (www.agilealliance.org/resources/initiatives/technical-debt).

[3]"List of tools for static code analysis" (Wikipedia).

[4]"SQALE" (Wikipedia).

[5]"Dynamic inconsistency" (Wikipedia).

[6]Martin, Robert C. *Clean Code*. Prentice Hall, 2009.

[7]Feathers, Michael C. *Working Effectively with Legacy Code*. Prentice Hall, 2005.

Declan Whelan is an Agile consultant, cofounder of Leanintuit, and a director at the Agile Alliance. Mr. Whelan works with organizations to improve their products and services through Agile and Lean practices. His personal mission in 2016 is to change the conversation around technical debt. Rather than viewing it as a problem to be fixed, we need to view it as feedback to drive improvements to our product and service delivery. He can be reached at declan@leanintuit.com.

Using Technical Debt to Make Good Decisions

by John Heintz

"Knowledge is power," wrote Francis Bacon. Making decisions without important information can result in poor choices. That is especially true with respect to technical debt. Using a technical debt framework to inform our decision making is a powerful technique to help us quickly and confidently make better judgments. I've worked with numerous organizations that have benefited from understanding their technical debt this way, and in this article, I will share stories about three different clients and how they were able to make good decisions for their businesses based on the data we found during technical debt assessments.

MAKING GOOD DECISIONS IN YOUR PROJECTS

Making decisions is a part of every day, in every job. You've probably asked at least one of these questions today:

- Should we start project X?
- Should we cancel project Y?
- Should we reinvest in another project?
- Should we put some features on hold, or speed them up?

Only sometimes are these questions easy to answer, and often they create conflict and indecision. Today's complex business landscape, combined with increasing expectations, creates growing pressure for decisions to be made quickly and accurately. We can't afford to make the wrong decision, and we can't afford inaction.

While risk analysis is frequently used to help make decisions, an often-overlooked risk involves time. We need to get to market; we want to quickly respond to change. Responding to change, however, means that we must remain responsive over time. Building for only the short term can easily create unmaintainable systems. This poses a significant risk to our futures because the competition and technology landscape are constantly changing. The following questions can help uncover the effect that time has on our project:

- What is the risk that we will slow down?
- What are the consequences of slowing down?
- How can we prevent our slowing down?

In addition to factoring in time directly, we need to consider the cost of delay. We can calculate how much it costs us to not release features.[1] Measuring and calculating this cost can affect many of our decisions.

THE COST OF CHANGE

Figure 1 shows that increasing the cost of change leads to slowing down our customer responsiveness.[2] One of the primary reasons that we slow down is because it gets harder and harder to change things. If the cost of change in our systems goes up, then at some point we become less responsive, and we slow down. Here, we are breaking the cost of change into two different components. One of them is the basic underlying cost of change, the natural or optimal cost of change for our systems. The second curve shows the additional cost of change due to technical debt. When our systems have high technical debt, it makes everything harder to do.

Figure 1 — Cost of change and customer responsiveness. (Source: Highsmith.)

DEFINING TECHNICAL DEBT

What is technical debt? Consider the metaphor of running through mud. There are two consequences of running through mud. One of them is low speed, because the mud has high friction; therefore it slows you down. The second consequence is that mud is much less stable, making it much easier to injure yourself, such as twisting your ankle or falling. These consequences are metaphors for developing with systems that have high technical debt. Everything else about the systems is harder to do, slower, and more dangerous — there is a higher risk of failure in production, and the systems will be harder to maintain.

Even if we have a solid architecture, if it is implemented on poor-quality code we will still have problems. Making any changes or extensions to that architecture is hard because the code must be changed to do so. Our code is the foundation that we stand on; whether it is solid and stable or slippery and muddy depends on the level of technical debt.

How do we measure technical debt? Where does technical debt show up in our systems?

In Figure 2, the aspects highlighted in blue contribute to measures of technical debt. Code rule violations are indications of technical debt. Duplications in the code are copy-and-pasted blocks that exist in more than one place in the code base. The lack of comments or public API documentation is a risk because undocumented code can be misused accidentally. Code complexity — meaning McCabe cyclomatic complexity — is an indication that modules are either easy or hard to understand depending on how deeply nested the logic is. Code coverage, or rather lack of code coverage, means that we do not have a safety net of unit tests to tell us when one change has caused non-local failures.

Each of these things is a measurable aspect of the code, and against each one of these measurable aspects, we can assess an estimate of the time needed to fix it. For example, a code rule violation might take six minutes to fix, while duplicated blocks of code take two hours to fix. When estimates for all the aspects are applied, we can sum up the total effort require to remediate the total measured debt.

In this example, we have 1,488 days of total effort to remediate all of the measured technical debt in the system. Keep in mind that zero measurable technical debt is not the goal. That would be trying to achieve "perfect" code, which isn't exactly the goal in a good, functioning business system. What we want is appropriately low levels of technical debt so that we do not tend toward bad customer responsiveness or risks to future maintainability.

High complexity is an indication of error-prone modules. One analysis found that at cyclomatic complexity of 38 units per module (Java class), the risk of errors occurring was 50% (see Figure 3).[3] That is a significantly

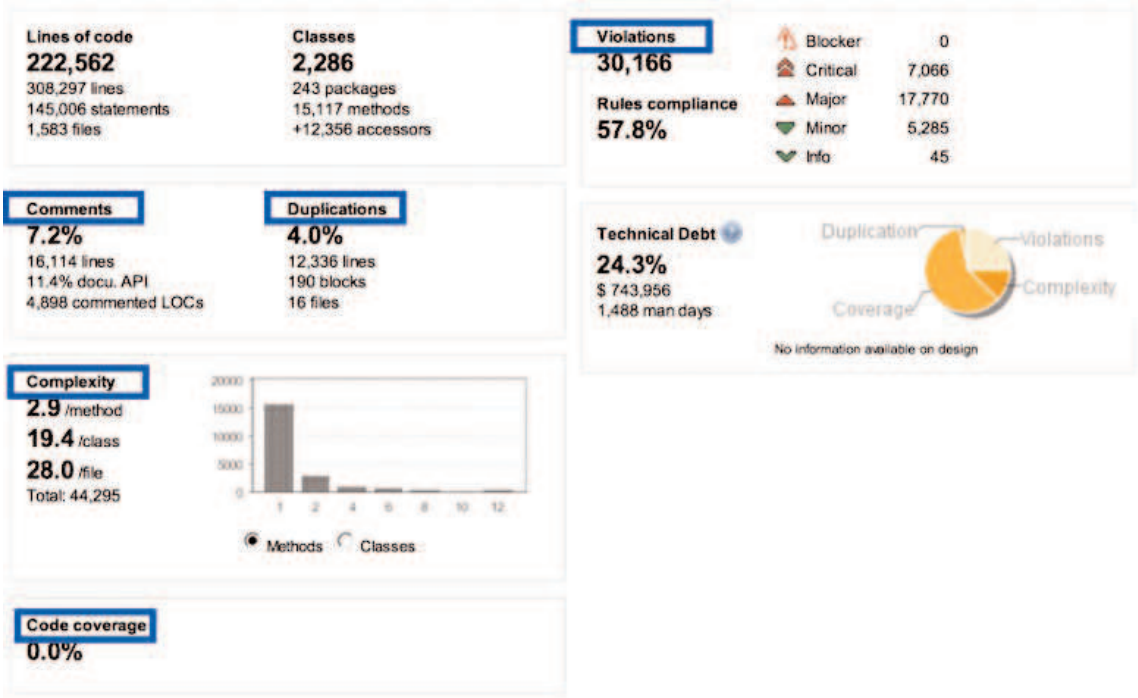


Figure 2 — Aspects of measurable technical debt.

high number. The lowest risk was at a complexity of 11, which is a very low cyclomatic complexity for a Java class, with a risk of error of 28%. The risk rises very rapidly with increasing complexity.

So what is technical debt again? Technical debt is two things. First, technical debt is an indication of the total remediation cost. Again, we do not want to remediate all the way down to zero but to a responsibly low level, and then to stay at that responsible level — not to trend up. So it is an effort cost to fix measurable problems in the code base. Second, technical debt is an indication of future risks of failure. In this case, what we can see is that technical debt gives us a hint at how risky projects are to our future cost of change or reliability in production.

When customer responsiveness has started to fall off precipitously, as in Figure 1, we have only three strategies that we can apply:

1. We can do nothing, and likely everything will continue to get worse. Our customer responsiveness will continue to go down.
2. We can replace the system in a big rewrite. This is, of course, a high-cost and high-risk endeavor.
3. We can invest in incrementally refactoring and cleaning up the system, paying down that technical debt.

None of these strategies is easy, and all have consequences. Once we have gotten to a significantly high cost of change and low customer responsiveness, however, they are the only options left to us.

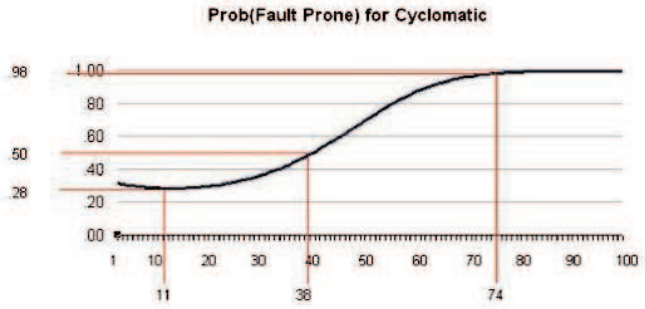


Figure 3 — Likelihood of error proneness given cyclomatic complexity. (Source: Terrill.)

USING TECHNICAL DEBT TO MAKE GOOD DECISIONS: THREE STORIES

Story 1: The Online Broker That Refactored

My first story is of an online broker in Texas that refactored and cleaned up their code base.[4] They had a 10-year-old Java-based system in production and were serving a large number of customers. When my colleagues and I engaged with them, they were experiencing reliability problems with the system and had themselves concluded that everything was hard to do. This was a high-friction environment that had become difficult to maintain.

Figures 4 and 5 show the technical debt at the beginning of the project and at the end of the project. The key here is that at the beginning of the project, we measured 740 person-days of technical debt. After the refactoring effort, only 415 person-days of technical debt remained measurable in the system. We had made significant cleanup progress, mostly at the lower levels of the system.

Figure 4 — Technical debt at the beginning of the project.

Figure 5 — Technical debt at the end of the project.

Let's look at this case in a little more detail. This was an online broker of precious metals such as gold and silver. Managing and tracking the financial exchanges was paramount; the cost of failure was exceptionally high.

We refactored the heart of the application, cleaning up and exposing the central business logic. We built a Hibernate object-relational mapping domain model as the core of the application, refactoring all of the business logic that had been strewn about the application into that core logic. We succeeded in creating a layered design and a clear dependency-managed system. This gave us a solid foundation in the application to then make better, safer decisions going forward.

Providing technical debt measures to the online broker allowed them to see why they were experiencing reliability problems with the system and why everything was so hard to do. Here are some of the key things that changed during this project. At the start of the project, there were over 90,000 lines of code in the production system. At the finish of the project, there were just over 70,000. We reduced the size of the system. We removed lines of code. We also removed significant complexity. The average complexity at the start of the project was 23 units of cyclomatic complexity per Java class. At the end of the project, that average went down to 15.

While there were still some classes that had significantly high complexity, we made substantial changes across many of the classes, including key central business classes that were simplified greatly. The duplicated blocks of code went down from 22% to 7.6%. Admittedly, 7.6% is still a high number for code duplications, but if we had measured only those code duplications in the central business domain layer, it would have been a much, much lower figure than that. The user interface and top layer of the application were not refactored significantly and still had quite a bit of duplication in them.

We also increased the unit test coverage from 0% to nearly 20%, which was a major increase in testing coverage. This gave the development team a modest but solid safety net with which to work.

Story 2: The Online Retailer That Rebuilt

We also worked with a San Francisco–based, international online retailer. They have a dozen teams spread throughout the US, Europe, and India. The user interface for their system was becoming very difficult to maintain, very buggy, and very slow to change. There was already an internal ground-up effort to replace it. Management, however, was skeptical of this effort and unsure whether it was actually needed.

We found two important factors for management to consider. First, our analysis of the code bases showed that the user interface layer was, by far, the most technical debt–saturated part of the system, measuring at least $11.28 per line of code of technical debt. This is outside of the common range that we usually see, which is usually $2-$10 per line of code. So this was significantly tech debt–ridden code.

Second, the Java server pages totaled 762,000 lines of code. The amount of duplication in this code base, in this user interface, was over 40%. This was a very high number, indicating that the ability to maintain this code had degraded considerably. After we showed management this information, they were able to conclude with confidence that the rewrite effort was justified. The organization was then able to move forward with a unified purpose and replace the user interface.

Story 3: The Tool Vendor That Increased Technical Debt

Making good decisions about technical debt would seem to preclude the notion that you might choose to *increase* technical debt. However, we worked with a tool vendor that chose to do just that. Why was increasing technical debt a good idea for this company?

The enterprise in question, a Germany-based international manufacturer, had adopted both Agile and technical debt measures. They had stringent targets for their technical debt, achieving some of the lowest levels of technical debt we have measured. After making significant progress on reducing technical debt in their systems, a high-value opportunity to win a customer presented itself, but the company was given an exceptionally short time frame in which to land the business. If they had chosen to continue with the same strategy of pushing technical debt down, they would have not been able to deliver in time to win this customer and their business.

The cleanup period shown on the left of Figure 6 is what they *had* been doing; namely, significantly reducing the technical debt in the system. When this new opportunity arose, they were presented with the option of passing on it or duplicating and copying an entire subsystem, thereby creating a massive amount of duplication in their systems and increasing their technical debt significantly. This increase in technical debt was in violation of their stated policies.

In the end, they made a very reasoned and mature decision to duplicate the subsystem, while remaining diligent about technical debt and following up afterward to pay back the debt they incurred. In the paydown

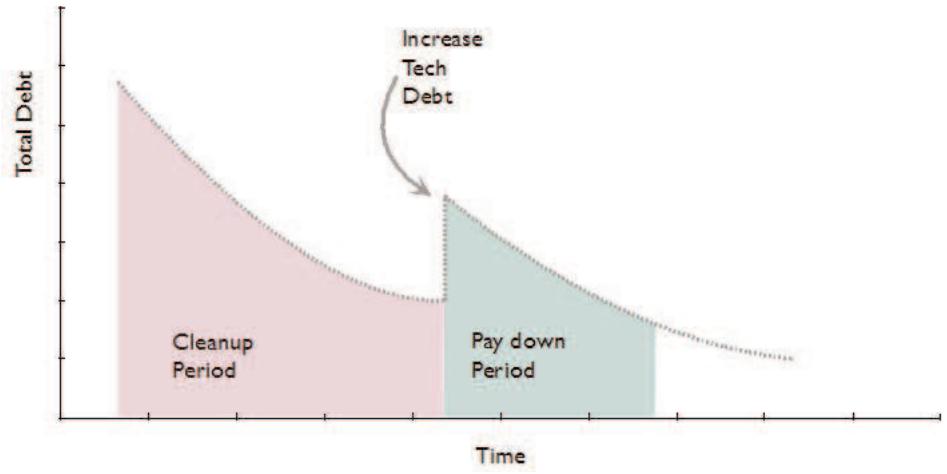


Figure 6 — Managing technical debt, before and after a significant opportunity.

period, they continued to invest in reducing that debt and removing the duplication over time by refactoring the systems. In this way, though they made a business decision to incur debt that had high payback, they had the maturity to continue remediating their debt.

CONCLUSION

If time is money, we need to consider the risk of going slow in the future. We can use cost of delay and customer responsiveness as two measures to quantify this risk. The takeaway, though, is that while going fast today is critically important, we also need to build the capability to go even faster tomorrow. If our systems are going to slow down, and we do not work to prevent that from happening, then we're setting ourselves up for low customer responsiveness problems in the future.

Technical debt measures two things: (1) the total remediation cost to clean up the measurable code-level technical debt aspects of the system, and (2) the risk of future failures. It is an indication of that risk.

What's next? Here are a number of things to do with your team:

- Identify the products at risk of going slow.
- Consider using the cost of delay to assess that risk.
- Assess technical debt measures for each at-risk project.
- Use technical debt measures as a leading indicator to develop a remediation plan to prevent future problems with customer responsiveness.

ENDNOTES

[1]Gat, Israel. "Cost of Delay Strategies in the Presence of Technical Debt." Cutter Consortium Agile Product Management & Software Engineering Excellence *Advisor*, 3 March 2011.

[2]Highsmith, Jim. "The Financial Implications of Technical Debt." Jim Highsmith (blog), 19 October 2010.

[3]Terrill, Gavin. "Cyclomatic Complexity Revisited." InfoQ, 31 March 2008.

[4]To read more about this project, see: Heintz, John. "Modernizing the Delorean System: Comparing Actual and Predicted Results of a Technical Debt Reduction Project." *Cutter IT Journal*, Vol. 23, No. 10, 2010.

John Heintz is a Senior Consultant with Cutter Consortium's Agile Product Management & Software Engineering Excellence practice. He is an experienced Agile manager, particularly in Lean and Kanban. In 2008, Mr. Heintz founded Gist Labs to further focus on the essential criteria for innovative success. On a recent project, he coached a 100-person Agile/Lean game studio, helping the organization increase its throughput of game features per month while coordinating cross-team communication paths, resulting in a doubling of features in one year.

Mr. Heintz began his career as a technologist and coach, always seeking solutions with greater leverage and deeper simplicity. His approach to systems and team building emerged in 1999 while leading his first Scrum team, coaching XP and test-driven development. Mr. Heintz has consulted with clients on various engagements, including for enterprise architecture, development practices, XP and Scrum leadership, Lean value stream mapping, and RESTful/messaging architecture, and has developed an inhouse training course in AspectJ. He has built single-source hyperdocument SGML publishing systems, a version-control CORBA/Python CMS, and an AspectJ dependency acquisition framework, and added test automation to many Java and .NET systems. He is a regular speaker at industry events, including No Fluff Just Stuff (NFJS), Architecture and Design World, Dallas JavaMUG, *and* Agile Austin. *Mr. Heintz holds a BS in electrical engineering from the University of Michigan. He can be reached at jheintz@cutter.com.*

Managing Technical Debt with the SQALE Method

by Jean-Louis Letouzey

Since its publication in 2010, SQALE[1] has become the industry standard method for managing technical debt. This open source, royalty-free method is implemented by multiple static analysis tools, including the SonarQube platform,[2] which is used in more than 50,000 companies, with an estimated 2 million users.[3]

This article will present the key concepts of the SQALE method and explain how to use it, either in a day-to-day context (as, for example, within an Agile project) or at corporate level to govern a portfolio and optimize its technical debt. The main goals of the method are to:

- Provide a rough estimation of the principal and interest of the technical debt of a piece of source code. It could be a small piece like a file or a complete IT domain build of numerous applications.
- Provide indicators that allow detailed analysis of the nature of the technical debt.
- Support remediation strategies with relevant indicators. As we will see later, there is no one magic strategy for paying back technical debt. There are many potential strategies, and the right choice is highly dependent on the context.
- Be implementable within an automated solution in order to provide real-time visibility and decision support.

To achieve these goals, the SQALE method uses four concepts:

1. A quality model
2. Estimation models
3. Indices
4. Indicators

I will describe each of these concepts below.

THE QUALITY MODEL

The SQALE quality model is the list of good practices that a project team or organization considers its definition of "right code." This list will serve as a reference for estimating the technical debt of the code. Any noncompliance with the quality model creates debt, and, conversely, there is no debt without the breach of at least one of the requirements.

If you don't have such a list and don't have time to establish one, you can use the Agile Alliance Debt Analysis Model (A2DAM) just released by the Agile Alliance. The A2DAM is a list of very basic good practices that you can use as a quick start. You can also use the default list provided by your static analysis tool. Project retrospectives are good opportunities to adapt and enrich the initial list to the specific context of your project.

THE ESTIMATION MODELS

The SQALE method contains two estimation models. One is used to estimate the time to remediate each debt item contained within the code and identified by the static analysis tool. This time is the principal associated with the debt item and is called the *remediation cost*. As an example, during the last week, a project team made some quick-and-dirty implementations in order to satisfy an important deadline. In doing so, they made 15 violations of their definition of right code. Using the SQALE estimation model, the tool will estimate the associated remediation cost as 3h 20min. In other words, by taking some shortcuts, the team has "borrowed" 3h and 20min of work, time that they will have to spend later to implement the code correctly.

The second model estimates the impact of the debt items on the business and is called the *non-remediation cost*. It estimates the future additional costs, such as extra work imposed on anyone working with the code, that arise from technical debt. This cost could also be considered as the cost of delaying the remediation.

Therefore, with SQALE, each debt item has two costs: the remediation cost and the non-remediation cost. All these calculations are performed by the analysis tools supporting the method. Most of these tools have

SQALE estimation models already preconfigured, so you can start analyzing your code with their default settings. As with all estimation models, it will be beneficial to adjust them after you have seen the results.

When you add all the remediation costs of all the debt items discovered by the code analysis, you get the *technical debt* of the component, application, or software domain. When you add all the non-remediation costs of a software component, you get the *business impact* (the interest part) of the component.

THE INDICES

I have already introduced the technical debt index and the business impact index resulting from the two estimation models of the method. These two indicators should be monitored and made transparent to all the participants in a project. Everybody will know at all times how much technical debt the project is facing in terms of either principal or interest.

Another important index is the SQALE debt ratio, which is the technical debt divided by the budget of the project. To get back to the financial metaphor, one way to evaluate the health of a company is to calculate its debt ratio, which is the ratio between the company's debts and assets. By analogy, the SQALE debt ratio allows you to monitor the health of your projects and applications.

THE INDICATORS

The most used indicator is the SQALE rating (see Figure 1).

It is obtained by plotting the SQALE debt ratios of projects and applications on a grid in order to yield a simple letter grade: A, B, C, D, or E (see Figure 2). A low debt ratio produces an A grade, whereas a high debt ratio results in a E grade. Associating colors to each grade allows the tool to present a global map of a very large portfolio, thus enabling users to immediately identify a potential threat (see Figure 3[4]).

The second most used indicator is the SQALE pyramid, which represents the distribution of technical debt in terms of quality characteristics (see Figure 4). This indicator can be read in two ways:

1. The analytic view (represented by the numbers in the left column and the light blue bars)
2. The consolidated view (represented by the numbers in the right column and the dark blue bars)

Let's start with the analytic view. In the example shown in Figure 4, the debt related to reliability is 8d 1h. If this amount is perceived as more debt than is desirable, the development team can initiate training or coaching on one or more factors that are the cause of this debt (e.g., inadequate exception handling, or a potential null pointer exception). By taking these steps, the team can limit the rate of future technical debt accumulation and improve the reliability of the delivered code.

Figure 1 — An application summary in SonarQube.

Rating	Up to	Color
A	1%	
B	2%	
C	4%	
D	8%	
E	∞	

Figure 2 — An example of a SQALE rating grid.

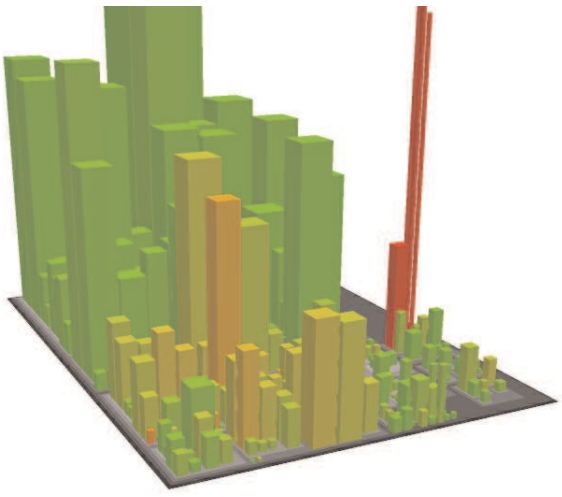


Figure 3 — A city-like representation of a large portfolio.

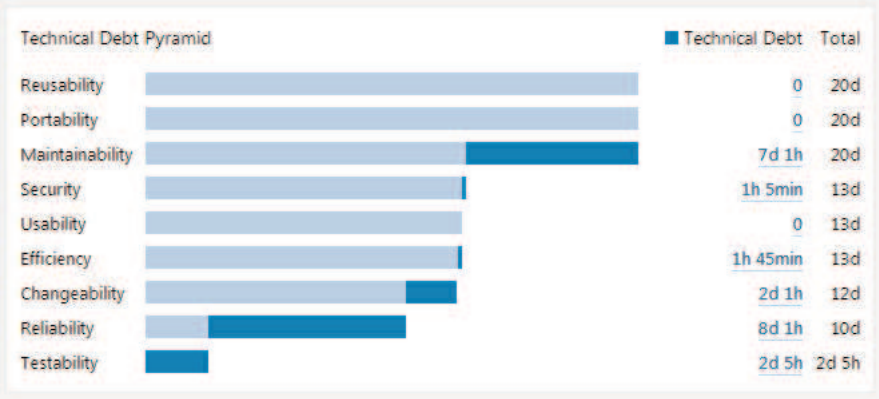


Figure 4 — A SQALE pyramid.

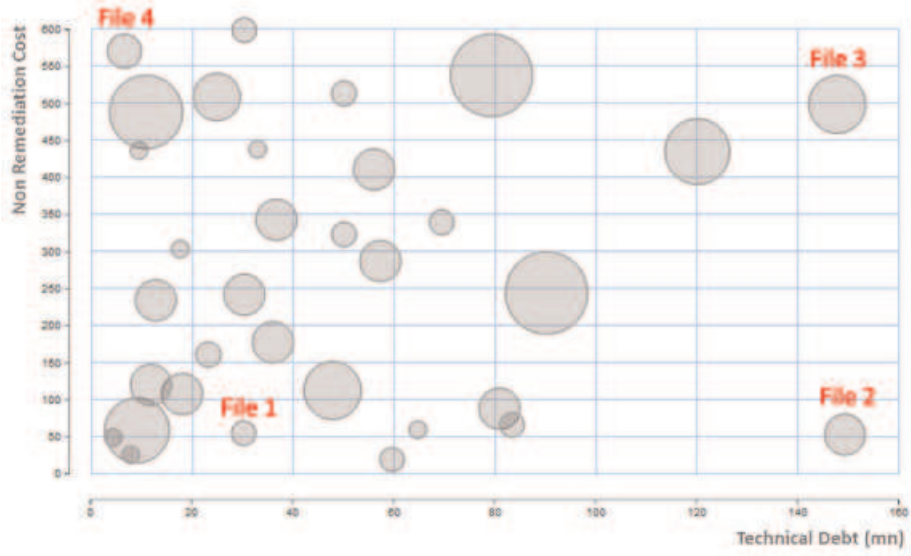


Figure 5 — A SQALE debt map at the file level.

Now let's see how to interpret the consolidated view of the pyramid, which is obtained when we add the debt of all lower characteristic levels for a given characteristic. These calculations are shown by the numbers in the right columns. Take the example of changeability, which has a consolidated value of 12 days. Agile projects generate a large number of change cycles to the code. The necessary quality characteristics to support these developments are testability, reliability, and changeability. Because changeability builds on reliability and testability, the true distance between the current state of the code and the target state of having easily changeable code is the summation of the debt associated with each of the three characteristics (2d 5h + 8d 1h + 2d 1h = 12d 7 h [rounded to 12d by the tool]). This consolidated value answers the following question from a business representative: "How far are we from having changeable software?" This consolidation mechanism is applicable to all characteristics.

The last SQALE indicator I want to introduce is the SQALE debt map. This is a bubble graph on which an item (a file, a component, an application) is represented on two axes, the technical debt and the business impact (see Figure 5). I will discuss its usage later on.

MANAGING THE TECHNICAL DEBT OF A PROJECT OR AN APPLICATION

Once you know how much debt you are facing and you have the ability to analyze the nature of the debt, you are in quite a good position to start paying your debt back.

The first instinct would be to rush in and fix the debt items that have a very high impact (non-remediation

cost) and a low remediation cost. Such remediations will have a very high ROI and so they will be easy to justify.

While this logic sounds rational, in many cases it is not optimal. In reality, you will find that some debt items, including some with very high impact, are located in pieces of code that also have a structural debt issue. These pieces should be completely refactored because they are too complex, have bad coupling, or appear to be duplicated code. In such cases, if you start fixing the high-impact issues first, the time you spend on those items will be lost when you fix the structural issues later. There are frequent scope dependencies between debt items, and this should be taken into account in the prioritization.

So, because the prioritization of refactoring is more complex than it appears at first glance, the SQALE method supports three different strategies that correspond to different contexts. The key input is the available budget for the refactoring.

Case 1

You are far from the delivery date, and you are able to allocate time to address a large percentage of your total technical debt (at least 60%).

In this case, as I explained before, you need to improve the quality of code by first fixing the structural debt. In practice, this is equivalent to making it testable. This is the technical debt associated with the first level of the SQALE pyramid, and it relates to issues such as too complex methods, duplicated code, and so on. After that, you pay back the debt associated with the next layer of the SQALE pyramid, which is reliability. And you continue up to the highest level of your pyramid.

Case 2

You have limited time. You can't repay the debt related to testability because it's structural and too time-consuming. You will be constrained to deliver your application with remaining debt. Thus, it would be wise to reduce the business impact (or the non-remediation cost) of this debt. You will focus your efforts on fixing the issues with the highest potential business impact. These are, in general, the issues related to reliability and security. In the case of the example in Figure 4, you will need about 8d and 2h.

Case 3

You have very limited time. You can't pay back all the debt associated with reliability and security, so you focus on the remediations that have the highest return on investment. In this case, you use the debt map (in Figure 5), and you fix files in the upper left corner, because their fixes have a very high ROI.

As noted earlier, this last strategy to improve the code is not optimal, because you will probably fix potential bugs in pieces of code that should be refactored for structural reasons. If so, this time will be lost. We can say that this is the quick-and-dirty way to manage technical debt.

These three cases summarize how the SQALE method provides relevant indicators that help teams and other interested parties make the optimal debt remediation decision, whatever the context.

If an application provides very little business value and its annual maintenance workload is very low, the fact that it is not well positioned in the debt map is not worrisome.

MANAGING THE TECHNICAL DEBT OF A LEGACY PORTFOLIO

From a management point of view, it is very beneficial to get a transparent view of the technical debt of the complete portfolio. To do so, the SQALE method proposes using a portfolio-level debt map. In this case, the points on the map are applications. Each application is positioned according to its technical debt density and its non-remediation cost density. This allows teams, portfolio managers, executives, and others to analyze the situation of the complete portfolio and to compare all the different applications whatever their technology, size, or context.

Consider the example shown in Figure 6. App B contains about 5 times more technical debt than App A. All things being equal, the technical debt of App C is 40 times more dangerous than that of App A. Using the debt map in this way will help you to analyze the situation and identify which parts of your portfolio need attention.

If an application provides very little business value and its annual maintenance workload is very low, the fact that it is not well positioned in the debt map (meaning that it is in the top-right corner) is not worrisome. On the contrary, if an application is critical and its code is not of good quality (i.e., also positioned at the top-right portion of the debt map), this represents a risk, and paying back the technical debt of this application may be a high priority.

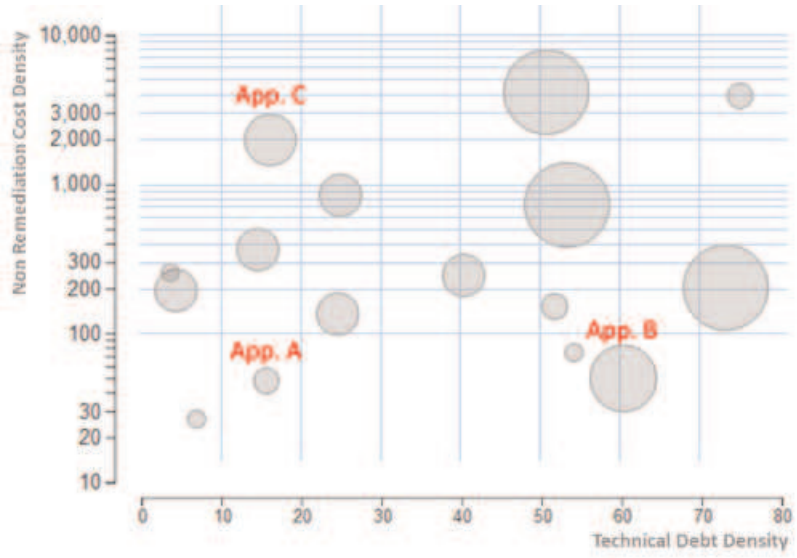


Figure 6 — A SQALE debt map at the application level.

In order to get such visibility and decision capability, it is important to use consistent estimation models across the whole organization. This is not so easy to achieve, because each business unit will find good reasons for using its own models. Establishing common estimation models is a corporate initiative and should be sponsored at the C level. It may require some external consulting support to achieve consensus among all the business units.

PARTING THOUGHTS

In addition to the debt ratio and the rating, the SQALE method includes many more useful indicators that provide deep insights into an organization's technical debt situation and support optimal decisions about it. This set of graphic indicators helps to establish a visual language for reporting the current state of a project or portfolio. This visual language is tool independent, which is an important contribution to the success of the method.

ENDNOTES

[1]The method definition document and other related articles and blog posts are available at www.sqale.org.

[2]The SonarQube platform is an open source project powered by SonarSource. Most of the graphics samples provided in this article are screenshot-produced with this tool.

[3]Campbell, G. Ann. "Mainstream: Noun. The Principal or Dominant Course, Tendency, or Trend." SonarQube (blog), 30 September 2015 (www.sonarqube.org/mainstream-noun-the-principal-or-dominant-course-tendency-or-trend).

[4]The map provided in Figure 3 is produced by a SonarQube plugin developed by Excentia (www.excentia.es).

Jean-Louis Letouzey is an expert consultant at Inspearit and the author of the SQALE method for managing technical debt. Mr. Letouzey consults to the world's leading organizations for the implementation of corporate solutions for managing technical debt. He is also frequently embedded in due diligence teams for assessing the technical debt of the acquired software. Mr. Letouzey is a frequent technical speaker at international conferences. He can be reached at jl.letouzey@gmail.com.

The Psychology and Politics of Technical Debt:
How We Incur Technical Debt and Why Retiring It Is So Difficult

by Richard Brenner

Many long-standing problems like technical debt owe their longevity to two factors — not dealing effectively with their causes and not dealing effectively with their resilience. Because what limits our ability to deal with technical debt might not be technical, it is useful to explore possible psychological and political sources of the longevity of the technical debt problem. Below are five possibilities:

1. Important generators of technical debt lie beyond the control of IT.
2. The technical debt metaphor suggests unfavorable and misleading associations.
3. The language we use to discuss technical debt affects our ability to deal with it.
4. We regard technical debt as a real thing, rather than the abstract construct it is.
5. Cultural debt leads some to regard IT as an expense to be minimized rather than a strategic partner.

After exploring these five factors, I will suggest five guidelines for designing effective technical debt management policy.

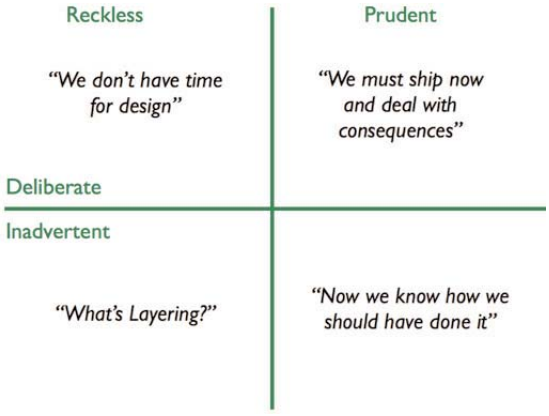


Figure 1 — Martin Fowler's Technical Debt Quadrant. (Source: Fowler.)

THE TECHNICAL DEBT GENERATION PROBLEM

Conventional definitions of technical debt attribute it to purely technical activity. For example, many hold technical debt to be a result of substandard IT practices,[1, 2] which contradicts Cutter Fellow Ward Cunningham's original conception of the problem (see sidebar "The Cunningham Definition of Technical Debt"),[3, 4] but such definitions are now widely accepted. In this article, I define technical debt as the collection of technology artifacts that we would like to revise or replace for sound engineering reasons. This definition includes Cunningham's notion and adds both substandard practices and obsolescence as sources of technical debt. From the technical perspective, this definition is well described by Martin Fowler's Technical Debt Quadrant,[5] which is a 2x2 matrix classification[6] in which the axes are intention and prudence (see Figure 1).

Although the general view of technical debt is that IT generates it, technical debt can result from nontechnical decisions taken entirely outside IT. Consider this example:

> A year ago, Company A acquired Company B. Consolidating their IT functions was supposed to improve efficiency, but technological incompatibilities have delayed consolidation. While maintenance continues for some of the former Company B's systems, the combined enterprise is carrying technical debt and paying interest on it.[7]

THE CUNNINGHAM DEFINITION OF TECHNICAL DEBT

In 1992, Ward Cunningham described technical debt this way:

> 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.

In this example, the debt is technical only in its manifestation. It is not technical debt in the usual sense, because Company A+B incurred it as a result of the acquisition, which did not anticipate the full cost of consolidating the two enterprises.

We can readily assess, unambiguously, the amount of financial debt we owe. Not so with technical debt.

Technical debt can arise for more mundane nontechnical reasons when organizational elements ask IT to "do whatever you have to do" (DWYHTD) to solve an urgent problem. Consider these scenarios:

- Sales debt: "We need this now, or we'll lose the account. DWYHTD!"
- HR debt: "The law requires that we comply by December 31. DWYHTD!"
- Customer support debt: "We're so swamped with calls on this problem that we can't deal with anything else. DWYHTD!"
- Budget debt: "Sorry, we cannot approve the resources for moving the final 30% of users from SharePoint 2007. Maybe next year. DWYHTD!"

In these examples, total spending consists of budgeted fiscal spending plus new technical debt, which IT incurs on behalf of other organizational elements. Budgetary processes control fiscal spending, but typically technical debt is uncontrolled. Managing technical debt accumulation in these situations would require enterprise policy changes.

Such policy changes must entail some form of accountability on the part of the organizational elements IT serves. We can reasonably expect people in those organizational elements, at times, to experience any mandated behavioral changes as curtailments of their freedom. If they do, we would likely observe a phenomenon called *psychological reactance*,[8] which tends to intensify opposition to perceived restrictions of freedom that circumstances might require.

Reactance can be a powerful behavioral determinant, as former US Air Force Major Martin Fracker argues. In 1994, as a cognitive psychologist and member of the faculty at the Air Command and Staff College, Fracker proposed psychological reactance as an explanation for Saddam Hussein's refusal to withdraw from Kuwait in 1990 despite the hopelessness of his position.[9] Researchers have investigated an analogous dynamic with respect to information systems adoption.[10]

THE TECHNICAL DEBT METAPHOR PROBLEM

Metaphors, which have the form "A is B," juxtapose concepts from disparate domains to underscore a particular idea.[11] Consider the metaphor "My son's bedroom is a war zone." His bedroom isn't literally a war zone, though it is messy. The A concept (my son's bedroom) is the *topic*, and the B concept (war zone) is the *vehicle*. To say "These technological artifacts are debt" is to use a metaphor to associate the topic, namely the technological artifacts, and the vehicle, namely financial debt.

Metaphors are powerful because they associate the topic and vehicle in compact, memorable ways. Consider some associations that the technical debt metaphor evokes. Perhaps the most fundamental is the interest on technical debt, which usually appears in the form of depressed IT productivity. The metaphor communicates the idea that outstanding technical debt has real, ongoing, undesirable consequences.

Yet a metaphor's associations can be a source of trouble, because we cannot control which attributes of the metaphor's vehicle the reader or listener will choose to associate with the metaphor's topic. I call this phenomenon *unintended association*.

Unintended association can be helpful. In finance, short-term debt usually carries a higher interest rate than long-term debt. In software engineering, we do not often consider the difference between short-term and long-term technical debt, but the distinction can provide useful guidance for decisions regarding incurring new technical debt or retiring existing debt, as discussed by software engineering expert Steve McConnell.[12] McConnell[13, 14] (as well as Raul Zablah and Christian Murphy of the University of Pennsylvania[15]) has shown how numerous other attributes of financial debt have technical analogs. Through unintended association, the debt metaphor suggests new insights that are potentially helpful in managing technical debt.

Unintended association can be problematic, however. If my two sons share a bedroom, saying that it's a war zone can evoke images of conflict between my sons, when I intend to communicate only messiness. So it is with the technical debt metaphor. Although we usually regard financial debts as having been incurred consciously, we aren't always aware of incurring technical debt. Moreover, we can readily assess, unambiguously, the amount of financial debt we owe. Not so with

technical debt. The impulse to "measure" technical debt arises, in part, from this unintended association. Moreover, people unfamiliar with managing technology might regard as incompetent any IT managers who are unaware of how they incurred technical debt, or who don't know precisely how large the debt is. Disabusing them of these notions can be difficult, in part because of unintended associations with regard to the technical debt metaphor.

Some more troublesome unintended associations stem from the social status of debtors in society. For many, excessive financial debt evokes images of profligate spending, laziness, and moral decay. These associations can hinder IT leaders as they urgently advocate for resources for technical debt management. Because of unintended association, some decision makers outside IT might regard technical debt as arising from mismanagement within IT, as evidenced by IT's ignorance of how they acquired the debt or how large it is. To the extent that they adopt this attitude, they're unlikely to support enterprise policy changes or additional resources for technical debt management within IT.

THE LANGUAGE PROBLEM

The language we use to describe problems influences how we think about them, and how — or how well, or whether — we solve them. For example, using the word *interest* affects how we regard technical debt. Interest rates on financial debts are relatively constant. Our experience with financial debt can thus suggest that the "interest rate" on technical debt is likewise relatively constant, even though it can fluctuate wildly, as I will discuss below. This misperception can lead to misplaced priorities for technical debt management if we don't account for interest rate fluctuations.

Yet more problematic language associated with the technical debt metaphor is the word *technical*. Many regard the term technical debt as appropriate, because it refers to a collection of technical artifacts. From that perspective, the term is apt.

But much technical debt has nontechnical sources. In situations like the Company A+B example above, the uncovered consolidation cost is usually called technical debt, but it is probably more properly regarded as *enterprise debt* that has a technological manifestation — a phrase that, unfortunately, does not quite roll off the tongue. The cost of retiring that debt, and the interest it accrues until its retirement, should be accounted for not as an IT operating expense, but as an IT expense associated with the acquisition. Labeling the debt in question technical debt obscures its true nature and typically prevents organizations from allocating resources sufficient to promptly retire it.

As we've seen, to meet the urgent needs of various organizational elements, IT sometimes incurs new technical debt, or defers retiring existing debt. To speak of these artifacts as technical debt, without reference to the reasons why the debts were incurred or remain, is an unfortunate use of language that tends to mistakenly identify IT as the debtor, thus insulating the true debtor from the debt. That error, in turn, can lead to errors in resource allocation, preventing debt reduction and retirement. Worse, it can obscure the true source of mismanagement, if any. Use of the word *technical* is at the root of this difficulty.

Much technical debt has nontechnical sources.... Labeling the debt in question technical debt obscures its true nature and typically prevents organizations from allocating resources sufficient to promptly retire it.

THE MEASUREMENT PROBLEM

When we seek solutions to problems, among the first questions we ask is, "How big is the problem?" Gauging the scale of the technical debt problem requires estimating both current debt and its interest charges. Both tasks are problematic. Let's consider debt estimation first.

Rarely can we estimate the absolute amount of technical debt in a given system. The urge to do so, and the urge to reject claims that absolute measurement is difficult if not impossible, are likely related to what psychologists call the *reification error*.[16] Philosophers call it the Fallacy of Misplaced Concreteness.[17]

The logical fallacy of reification occurs when we treat an abstract construct as if it were a concrete thing. Although reification can serve as helpful mental shorthand, it can produce costly cognitive errors. For example, advising someone who's depressed to get more self-esteem is unlikely to work, because self-esteem isn't something one can order from Amazon (or anywhere else). One can enhance self-esteem through counseling, reflection, and many other means, but it isn't a concrete object one can "get." Self-esteem is an abstract construct.

Technical debt is likewise an abstraction. We can *discuss* "measuring" it, but attempts to specify measurement procedures will eventually confront the inherently

abstract nature of technical debt, leading to debates about both definitions and the measurement process.

Consider, by contrast, national infrastructure debt associated with aging highway bridges.[18] Because bridges are physical things, successive independent assessments of this debt are likely to yield a fairly narrow distribution of results (low σ). On the other hand, independent "measurements" of total technical debt in a software system are much more likely to yield a broad distribution of results (high σ), because those results are affected by differences among the measuring processes and the definitions of technical debt.

Technical debt defaults are damaging. We might choose not to undertake some projects; we might not even propose others.

Moreover, technical debt as a management tool has meaning only relative to our intentions and ongoing activities. If we measure the technical debts of two systems that depend on a substrate technology that has just undergone a change, we might find that $Debt_{total} = Debt_1 + Debt_2$. But if we decide to retire System 2, and therefore suspend further maintenance and development on it, then for management purposes $Debt_2$ vanishes. It can vanish because it isn't real.

One additional point about debt measurement is worth clarifying. Unlike total technical debt, we can make reasonable estimates of *incremental technical debt* — the technical debt we would incur as a result of a particular contemplated implementation. The incremental technical debt is the cost of upgrading that implementation to the form we would have deployed if we were doing it "right." Estimating that effort is as straightforward (or not) as any other estimate. The distinction between measuring total technical debt and estimating incremental technical debt is that we need not detect the latter; we know what we're planning not to implement. By contrast, a measurement of total technical debt would require identifying all debt in the entire asset base.

Estimating interest on technical debt is no less problematic. Even if we could measure technical debt absolutely, the interest on that debt can fluctuate dramatically. In some time periods, the interest charge for a particular technical debt component might be zero if we aren't working on any system components that carry that debt. When we do work on those components, the interest charges can suddenly escalate. Contrast this with interest charges on financial debt, which typically depend only on the loan terms rather than what we're doing at any particular time.

Moreover, distinguishing between issuing new technical debt and paying interest on existing debt can be difficult. For example, Antonio Martini and Jan Bosch of Chalmers University of Technology have identified a phenomenon they call *debt contagion*,[19] whereby creating new system elements to be compatible with elements identified as debt effectively causes debt propagation. Although we can regard debt contagion as new technical debt, we could also argue that at least some part of it is metaphorically equivalent to borrowing funds to pay interest on existing debt. Because such borrowing signals an urgent need to retire the debt that led to it, tracking it as a distinct system attribute would be valuable to anyone intent on setting priorities for managing technical debt.

At times, interest charges on technical debt can be so high that we effectively default. Default occurs when the organization cannot meet — or elects not to meet — the interest payments[20] on some component of technical debt. This happens, for example, when an organization declines to undertake a project because doing so would involve modifying system components that are so burdened with technical debt that the probability of successful modification is unacceptably low, or the cost thereof would be unacceptably high, even if it could be completed on time. One might find comments in code such as "Don't ever touch this again" or "Do not modify. Contact Lisa to talk about when I can do it for you." Sadly, the authors of such comments have all too often moved on.

Technical debt defaults are damaging. We might choose not to undertake some projects; we might not even propose others. Worse, defaults can actually limit our imaginations. Although numerous static analyzers for technical debt and technical debt interest have been studied,[21-23] static analyzers aren't yet capable of estimating the cost of opportunities lost because of projects not undertaken, projects not proposed, or concepts not even imagined. We can only presume such costs to be very high indeed.

THE CULTURAL DEBT PROBLEM

Edgar Schein of MIT's Sloan School of Management defines organizational culture as "a pattern of shared basic assumptions learned by a group as it solved its problems of external adaptation and internal integration."[24] As information technologies have progressed,

some organizational cultures haven't adapted well enough or rapidly enough. When cultures view IT as a service organization, a remnant perhaps of the middle or late 20th century, it's not uncommon for some to regard IT as a source of expense to be minimized rather than as a strategic partner.[25-26] Trends toward strategic acceptance of IT are only slightly favorable, according to recent surveys of CIOs.[27] It is illuminating to view this artifact of organizational culture as a form of *cultural debt*.

Cultural debt, like technical debt, imposes interest charges. This form of cultural debt contributes to an alignment of organizational political power that constrains IT resources and tends to increase technical debt. Viewed from this perspective, some components of technical debt are actually the interest charges for cultural debt.

The condition persists, in part, because of a phenomenon known as the *identified patient*, as discussed by family therapist Virginia Satir.[28] In affected families, the identified patient is the troubled family member who serves as scapegoat, or whose symptoms serve an unhealthy family function. In some enterprises burdened with cultural debt, IT is the identified patient, enabling other organizational elements to operate in what they mistakenly regard as healthy balance. Although these organizational elements believe themselves to be coping as best they can with IT's failure to address its technical debt problem, the root cause of the trouble might be cultural debt, in which everyone plays a role.

Attempting to address technical debt while leaving cultural debt unaddressed — or growing — is unlikely to produce the desired results. Until everyone in the organization accepts their responsibility for technical debt formation, and supports policy changes that enable rational technical debt management, durable and superior organizational performance will remain beyond reach.

FIVE GUIDELINES FOR MANAGING TECHNICAL DEBT

The five problem areas described above suggest the outlines of a program for managing technical debt.

1. Control Issuance of New Technical Debt

Issuing technical debt is an essential enterprise function. It enables IT to provide services to organizational elements whose current budgetary authorization is otherwise insufficient for accomplishing their legitimate missions in unanticipated situations. Controlling the issuance of new technical debt restricts this deficit spending to activities both necessary for and closely aligned with enterprise objectives.

Nearly all projects generate new debt. Lessons learned documents that report on technical debt generation can support organizational learning about technical debt management.

When IT issues new technical debt on behalf of a business unit, sound technical debt management policy would hold that unit accountable for the debt and its interest charges. We can rate this debt using Fowler's Technical Debt Quadrant, adjusting interest charges accordingly. High-quality technical debt is debt that is issued with intention, and prudently; its interest rate could be adjusted downward. Low-quality technical debt is debt that results from inadvertent recklessness; its interest rate could be adjusted upward. This concept illustrates the importance of future research into projecting interest charges on new technical debt.

Attempting to address technical debt while leaving cultural debt unaddressed — or growing — is unlikely to produce the desired results.

2. Exploit the Psychology of Communications

Organizations can manage debts only if they recognize them. In the finance domain, the term *obligation* is often used in place of *debt*. It communicates powerfully the idea that debt must be repaid.

Any organizational asset — culture, HR processes, testing gear, the sales process, the product lineup, brands, and more — can accumulate debt. A term more general than technical debt, such as *asset obligation*, could communicate this concept. Organizations can reach their potential only if they are aware of *all* their asset obligations and only if they manage them successfully. Technical debt is just one form of asset obligation.

When we encounter reactance to new debt control policies, a typical management response is tighter controls. A more effective approach might emphasize concrete examples of the new freedoms that become available as the organization liberates itself from the burdens of accumulated technical debt.[29]

3. Be Guided by the Indirect Effects of Technical Debt

Technical debt (or asset obligation) retirement programs should focus foremost on reducing interest charges. For example, if we know that some debt component will reduce productivity for a particular project, applying resources to retire that debt component might be prudent. But elevated maintenance and development costs are just the most obvious forms of interest on technical debt. Indirect effects can be far more costly. An often-overlooked component of interest charges appears in the form of recruitment and retention costs pertaining to people who must labor on debt-ridden asset bases. Examine the organization carefully to identify all indirect effects of technical debt.

A transformation of enterprise culture that leads to a strategic role for IT is a necessary prerequisite to gaining control of technical debt.

4. Make Technical Debt Projections

IT can incur technical debt on its own behalf, but as we've seen, enterprises frequently incur technical debt at the behest of organizational elements other than IT. Even more important are the *external* forces that result in technical debt creation. For example, we know that CSS4 is coming.[30] When it is released, almost every website in the world will incur some degree of new technical debt. Indeed, every new release of substrate software, and much new hardware, can also create new technical debt. Rare is the enterprise that forecasts externally driven technical debt and forecasts resource allocations accordingly. Organizations that do so will soon gain competitive advantage.

5. Address Outstanding Cultural Debt

Enterprise leadership must assess whether the enterprise culture views IT as an expense to be minimized or as a strategic partner. If the former, expense minimization might have progressed so far that technical debt has reached levels that IT cannot effectively address alone. A transformation of enterprise culture that leads to a strategic role for IT is a necessary prerequisite to gaining control of technical debt.

PARTING THOUGHTS

While technical debt is certainly a problem, it can also be a symptom of an imbalance in the enterprise. If an imbalance exists, purely technical approaches to addressing the technical debt problem are unlikely to achieve much more than short-term relief. Resolving organizational imbalance, if it exists, could entail some degree of organizational transformation.

Cultural transformations are difficult, in part, because even after we successfully identify what must change, moving an organization as one toward that goal can be even more challenging. In this instance, because of a "virtuous cycle," the effort to manage technical debt can provide enterprise leaders the leverage they need. If the enterprise can hold accountable the organizational elements on whose behalf IT now issues technical debt, political power within the enterprise can be rebalanced, which would then facilitate the control of technical debt. IT can then assume the strategic role so necessary for enterprise success.

ENDNOTES

[1]"How to Calculate Technical Debt." Deloitte Insights (sponsored content in *The Wall Street Journal*), 21 January 2015 (http://deloitte.wsj.com/cio/2015/01/21/how-to-calculate-technical-debt).

[2]Kniberg, Henrik. "The Solution to Technical Debt." Crisp (blog), 12 July 2013 (http://blog.crisp.se/2013/07/12/henrikkniberg/the-solution-to-technical-debt).

[3]Cunningham, Ward. "The WyCash Portfolio Management System." *Addendum to the Proceedings of OOPSLA '92*. ACM, 1992.

[4]Cunningham, Ward. "Ward Explains Debt Metaphor" (video; http://c2.com/cgi/wiki?WardExplainsDebtMetaphor).

[5]Fowler, Martin. "TechnicalDebtQuadrant." Martin Fowler (blog), 14 October 2009 (http://martinfowler.com/bliki/TechnicalDebtQuadrant.html).

[6]Lowy, Alex, and Phil Hood. *The Power of the 2x2 Matrix: Using 2x2 Thinking to Solve Business Problems and Make Better Decisions*. Jossey-Bass, 2004.

[7]I choose not to use the term *technical interest* to refer to the interest rate on technical debt, because that term is already in use in the insurance industry.

[8]Brehm, Sharon S., and Jack W. Brehm. *Psychological Reactance: A Theory of Freedom and Control*. Academic Press, 1981.

[9]Fracker, Martin L. "Conquest and Cohesion: The Psychological Nature of War." In *Challenge and Response: Anticipating US Military Security Concern*s, edited by Karl P. Magyar. Air University Press, 1994.

[10]Matthias, Thomas, et al. "Psychological Reactance and Information Systems Adoption." In *Organizational Dynamics of Technology-Based Innovation: Diversifying the Research Agenda*, edited by Tom McMaster, et al. Springer, 2007.

[11]Compare the metaphor "A is B" to the simile "A is like B." In the literature of technical debt, similes and metaphors, which are both figures of speech, are often confused with each other, and with analogies. An analogy, by contrast, is not a figure of speech; it is a logical argument. For a most illuminating and complete discourse on the subject, see Clark, Brian. "Metaphor, Simile and Analogy: What's the Difference?" Copyblogger, 3 May 2007 (www.copyblogger.com/metaphor-simile-and-analogy-what%E2%80%99s-the-difference).

[12]McConnell, Steve. "Technical Debt." Construx (blog), 1 November 2007 (www.construx.com/10x_Software_Development/Technical_Debt).

[13]McConnell, Steve. "Managing Technical Debt — Construx Webinar." Construx, September 2011 (video; www.youtube.com/watch?v=lEKvzEyNtbk).

[14]McConnell, Steve. "Managing Technical Debt." Construx, June 2008 (www.construx.com/uploadedFiles/Construx/Construx_Content/Resources/Documents/Managing%20Technical%20Debt.pdf).

[15]Zablah, Raul, and Christian Murphy. "Restructuring and Refinancing Technical Debt." *Proceedings of the IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2015.

[16]Levy, David A. *Tools of Critical Thinking: Metathoughts for Psychology*. Allyn and Bacon, 1997.

[17]Whitehead, Alfred North. *Science and the Modern World*. Pelican Mentor (MacMillan), 1948.

[18]"2013 Report Card for America's Infrastructure." American Society of Civil Engineers, March 2013 (www.infrastructurereportcard.org/a/documents/2013-Report-Card.pdf).

[19]Martini, Antonio, and Jan Bosch. "The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles." *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, 2015.

[20]Downes, John, and Jordan E. Goodman. *Dictionary of Finance and Investment Terms*. 2nd edition. Barron's, 1987. In the financial domain, a default occurs when the debtor fails to meet scheduled payments, which could include principal in addition to interest. In the metaphor, however, we rarely make payments that combine principal and interest. Typically, principal is paid only during dedicated efforts to retire technical debt.

[21]Tomás, Pedro, María José Escalona, and Manuel Mejías. "Open Source Tools for Measuring the Internal Quality of Java Software Products: A Survey." *Computer Standards & Interfaces*, Vol. 36, No. 1, November 2013.

[22]Maldonado, Everton da S., and Emad Shihab. "Detecting and Quantifying Different Types of Self-Admitted Technical Debt." *Proceedings of the IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2015.

[23]Falessi, Davide, and Alexander Voegele. "Validating and Prioritizing Quality Rules for Managing Technical Debt: An Industrial Case Study." *Proceedings of the IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2015.

[24]Schein, Edgar A. *Organizational Culture and Leadership*. 4th edition. Jossey-Bass, 2010.

[25]Ross, Jeanne W., and David F. Feeny. "The Evolving Role of the CIO." In *Framing the Domains of IS Management Research: Glimpsing the Future through the Past*, edited by Robert W. Zmud. Pinnaflex, 2000.

[26]Ross, Jeanne W., and David F. Feeny. "The Evolving Role of the CIO." Center for Information Systems Research (CISR) Working Paper No. 308. Sloan School of Management, MIT, August 1999.

[27]"2016 State of the CIO Survey." *CIO,* January 2016.

[28]Satir, Virginia. *Conjoint Family Therapy*. Revised edition. Science and Behavior Books, 1967.

[29]I owe this insight to the work of University of Oklahoma communications professor Claude H. Miller and his coauthors on health communications. See: Miller, Claude H., et al. "Psychological Reactance and Promotional Health Messages: The Effects of Controlling Language, Lexical Concreteness, and the Restoration of Freedom." *Human Communication Research*, Vol. 33, No. 2, March 2007.

[30]Etemad, Elika J., and Tab Atkins, Jr., eds. "Selectors Level 4, Editor's Draft." W3C, 9 February 2016 (http://drafts.csswg.org/selectors-4).

Rick Brenner is Principal of Chaco Canyon Consulting. He works with people in dynamic problem-solving organizations who make complex products or deliver sophisticated services that need state-of-the-art teamwork and with organizations that achieve high performance by building stronger relationships among their people. Mr. Brenner focuses on improving personal and organizational effectiveness, especially in atypical situations, as in the case of continuous change, technical emergencies, and high-pressure project situations. From 1993 to 2014, he taught a course in business modeling at the Harvard University Extension School. He publishes a weekly e-mail newsletter and has written a number of essays that are available at his website, www.chacocanyon.com. He is the author of the e-book Leading Virtual Meetings for Real Results. *He can be reached at rbrenner@chacocanyon.com.*

Addressing the Hidden Obstacles to Innovation and Digital Disruption

by Ram Reddy

The IT function in the typical *Fortune* 500 company is increasingly expected to enable business innovation and support digital disruption while ensuring secure and reliable operations of existing enterprise applications. The expectations of IT are funded by IT budgets that are subject to regular cuts during the fiscal year when the company does not meet performance targets or experiences other competitive pressures. In the marketplace, shorter product and service lifecycles are creating enormous pressure for businesses to innovate to retain profitable market share.

One major obstacle to business agility and innovation is technology debt (TD). TD obstacles manifest themselves as non-IT executives complain that "we can't launch this new product/service as our IT systems will not allow us to." From an IT standpoint, the inability of existing IT systems to support the proposed new product/service launch is a result of past technology "workarounds" that were implemented to meet an accelerated timeline or reduced budget.

Given this history, how can the IT function engage with the business to remove technology debt and enable business agility and innovation? It is nearly impossible to prevent technology debt from being created — product/service lifecycles are getting shorter, making time to market an imperative for which taking shortcuts becomes a necessity. (The only exceptions that come to mind are of systems being developed and deployed for highly regulated environments or mission-critical applications in areas such as medicine and defense.) However, the size and scope of the technology debt being created can be contained, and on rare occasions some technology debt can be prevented, with proper analysis of the impact of a proposed workaround on business process agility/scalability.

Within IT, we have existing methodologies that can help non-IT executives engage with and take ownership of retiring technology debt. Despite enterprise architecture (EA) being touted for over a decade, rarely do we see companies map business architecture (BA) to underlying IT applications and supporting systems infrastructure. Transforming the EA function from a conceptual group to one that is responsible for creating the solution architecture will lay the groundwork for engaging business/functional owners in making the tradeoffs for workarounds.

In this article, I will explain how to get business engagement and "buy-in" to remove technology debt using existing IT processes such as the quarterly/annual budgeting process, design reviews, and functional walkthroughs. To obtain funding and business sponsorship for TD remediation projects, it is important to get non-IT executives to understand and support the TD remediation challenges. Understanding how technology debt is created is essential to preventing, containing, and retiring the debt.

TECHNOLOGY DEBT CREATION AND GROWTH

For discussion purposes, I will expand the definition of "technical debt" to "technology debt" to include different layers of the technology stack that composes a business application or function.[1] For example, the technology stack could consist of custom-developed code, configuration options for a COTS package, and the underlying database, computing, storage, network, and presentation layers that together deliver business functionality. The functionality could be a simple order entry (OE) system or a more complex ERP system with integrated business functions.

Figure 1 depicts an OE system and its components across the technology stack. At its inception, the system would have been designed to work with other layers of the stack through the application and Web service layers. The proper design for interaction is depicted with the thick bidirectional arrow. During the development phase, to meet performance goals or deal with resource constraints, workarounds are written to interact directly with other systems (such as the CRM and general ledger [GL] applications). These shortcuts — point-to-point integrations — are shown in Figure 1 as thin bidirectional arrows. For simplicity, the diagram shows

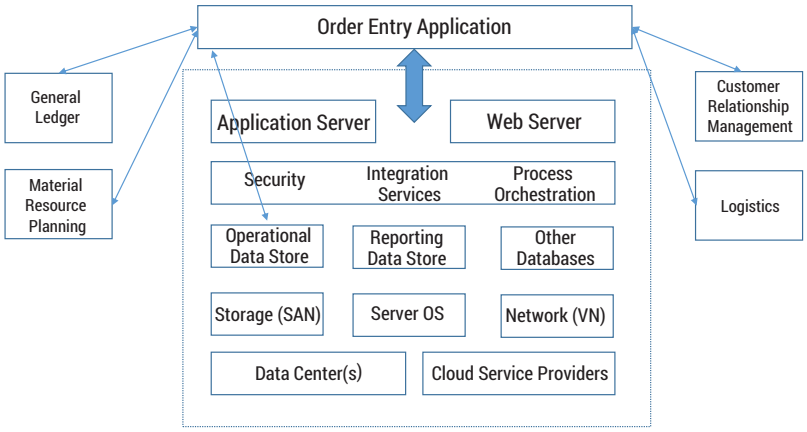


Figure 1 — Creating technology debt.

point-to-point interactions with just one application; in reality, there would be hundreds of point-to-point integrations with other applications and systems (such as the operational data store) by the time the OE system goes into production.

Once the OE system is in production, more integrations are added over time to deliver incremental functionality. The workarounds developed are typically not well documented, and the developers who wrote them would likely have moved on. The mostly undocumented workarounds (see Figure 1) create a "lock-in" between the OE, CRM, logistics, GL, MRP, and other layers of the technology stack. This lock-in grows over time as more and more upstream and downstream integrations are created. Any major changes to, or replacement of, systems requires a thorough analysis and remediation of the hundreds of integrations. This is an example of technology debt that needs to be remediated before enterprise applications in the lock-in category can support new capabilities or enable innovative business processes.

There are other sources of technology debt besides the example given above. To meet a reduced project budget, project teams make suboptimal choices on infrastructure areas such as servers, storage, and so on. While these choices allow teams to deliver systems that meet the current need, they may limit future growth — creating technology debt that needs to be addressed at a future date.

Technology debt can grow exponentially, especially if the company achieves business success using the IT system with workarounds. Successful workarounds create lock-in with the functional users of the system, making change difficult if not impossible. The IT workaround in production could consist of software shortcuts, quick-and-dirty definition of databases, and/or suboptimal systems infrastructure choices. The successful IT system will propagate through integration with upstream and downstream systems. Years later, when the business wants to change upstream or downstream systems to launch a new product/service, the production system may not support this without the workarounds being modified or replaced. In some instances, replacing the workarounds will have an adverse impact on current operations.

Systems inherited through mergers/acquisitions and "shadow IT" add to the technology debt. The size of the problem in most mature companies makes it a Herculean task, and no one wants to clean the proverbial Augean stables, especially if the debt is not visible. We need to document the technology debt across business processes and operational systems before we can seek non-IT executive support to remediate the problem. Using business architecture and IT service catalog processes, the IT function can identify TD areas, obtain business sponsorship, and secure funding for projects to remediate the debt.

The business architecture process[2] coupled with the IT services catalog[3] gives us the context to build and maintain an inventory of TD areas and their impact on business processes, operating costs, and service-level expectations.

GAINING BUSINESS BUY-IN USING BA AND THE IT SERVICES CATALOG

For the most part, the IT function is technically capable of designing and implementing solutions that remediate technology debt. The challenge lies in obtaining non-IT executive sponsorship and funding for TD projects. Figure 2 shows a high-level overview of the IT budgeting process that can be used to gain executive sponsorship and funding for TD projects. The suggested

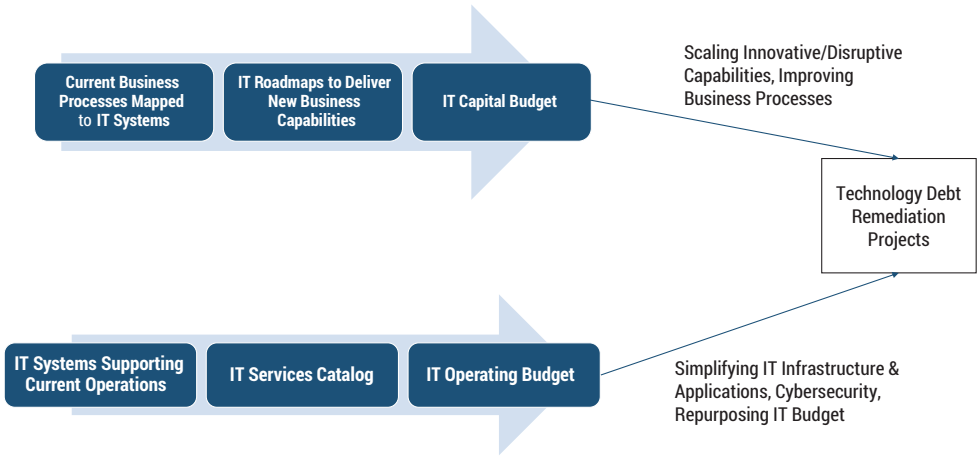


Figure 2 — Funding technology debt remediation projects.

approach is applicable even if a company does not have a formal IT services catalog or a BA repository.

Let us assume that the company depicted in Figure 1 decides to move to a cloud-based, software as a service (SaaS) provider for sales force automation (SFA), thereby replacing their existing CRM system. Migrating to the new platform will require an analysis of the upstream and downstream business processes that interact with the current CRM system.

During this analysis, the typical solution architecture is created with the sole purpose of implementing the SFA system. As part of the solution architecture, a TD review of the workaround integrations between the old CRM systems and other systems also needs to be done. The review and documentation of the technology debt in the BA repository should be led by the EA function, with input from subject matter experts of the various systems and technology stack areas. This review will pinpoint areas of technology debt that create process lock-in, impact scalability, and prevent improvement of existing business processes. This will result in the identification of TD remediation projects that should be funded as capital projects in support of the SFA system.

The SFA project team's primary focus will be on implementing the new system and not on removing technology debt surrounding the CRM system. The TD project needs to be executed in parallel with the SFA project. Artifacts from the BA process are critical for gaining support from non-IT executives for investment in TD remediation projects. The business process and IT roadmaps are visually compelling tools that can show the impact on business capabilities, agility, and prevention of innovation if technology debt is not remediated.

The other funding track for TD remediation projects is the annual/quarterly IT operating budget review. Most firms have elements of the data shown in the IT operating budget arrow in Figure 2. Even if a company does not have a comprehensive IT services catalog mapped to every operational system, it will have a high-level breakdown of the IT operating budget in support of different functional and technical areas. Areas of technology debt that would — if remediated — simplify IT infrastructure/applications, improve service levels, strengthen security, and repurpose the IT budget should be identified as part of this review process.

It is not necessary for systems providing new business capability to come out of the capital budgeting process or for IT infrastructure simplification projects to come out of the IT operating budget process. For ease of discussion, I have simplified the two tracks even though they are not mutually exclusive in terms of the types of projects. Similarly, the IT services catalog and BA processes can be quite extensive and detailed. However, to educate and engage non-IT executives in the sponsoring and funding of TD remediation projects, high-level business process maps and IT roadmaps that illustrate the impact of workarounds should suffice.

TACKLING TECHNOLOGY DEBT

Preventing TD

Technology debt can be created in the development, deployment, and production phases of the software development lifecycle. BA and functional ownership of the application/system are critical to preventing the creation of technology debt. Showing the non-IT system owner that implementing point-to-point integrations creates process lock-in and limits future business process improvements may lead to surprising results. Well-informed system owners may decide to reduce

project scope to meet budget reductions instead of approving workarounds that create technology debt.

Developed over time, an accurate BA repository and a comprehensive IT services catalog become invaluable tools in preventing technology debt from being created. By using artifacts from the BA repository and the IT services catalog, the adverse potential impact of workarounds can be shown on the business process and on the IT operating budget and service levels.

Containing TD

A major source of technology debt is the shadow IT functions within companies. In many instances, the IT function is unable to service the technology needs of a business unit or function, prompting the business unit to obtain and implement local IT solutions without the IT function's buy-in, budget, or resources. Conversely, the typical enterprise IT function is stretched thin and does not have the bandwidth to support ad hoc business/functional IT needs. Going forward, the enterprise IT function should engage, support, and guide shadow IT to make the optimal technology choices.

Optimal technology deployment may not be feasible if the business unit/function is evaluating quick-and-dirty prototype systems. Instead of fighting the business unit in question, IT should continue engagement and support, documenting the TD areas that will need to be addressed at a later date. Increasingly, business process innovation enabled by IT systems is coming from regional/local functional and business areas. The enterprise IT function should set aside resources to support shadow IT. Over time, most successful shadow IT systems become enterprise IT's responsibility anyhow, and engaging early with shadow IT can help contain the size and scope of technology debt by providing technical guidance that is typically missing in the business unit or function.

As part of the functional and technical design review gates for the OE system in Figure 1, the solution architect can educate the owners of the GL, MRP, and other systems about the process lock-in created by the point-to-point technology integrations. This can turn into a confrontational situation between the project manager and the functional owners of the OE, GL, MRP, and other systems, from whom the project manager will need resources to address the process lock-in issues. Tight budget and resource constraints can cause the functional owners to push back on the project manager.

The objective here is to identify the business processes outside the OE system impacted by the workarounds and to get functional owners to sign off on the workaround design that creates point-to-point integrations with the GL, MRP, CRM and logistics systems, as it has the potential to create process lock-ins (depicted in Figure 1). At a future date, any changes/upgrades to the GL or MRP processes would have to account for the impact of the OE workaround integrations. In the worst-case scenario, the process lock-in caused by the workaround is not discovered until an upstream or downstream system (such as the CRM or MRP) is modified and causes the OE system (now in production) to crash.

Typically the IT project team implements the workarounds without obtaining signoff from functional owners of upstream and downstream systems, as that would require expending scarce resources to obtain buy-in from these owners. As I've just noted, the result of such avoidance can be disastrous. Instead, discussing these issues as part of design review processes allows the business unit owners to provide input and take ownership of approved workarounds. This process of engagement allows the IT function to document the TD areas that may become remediation projects as part of the annual IT budget cycle.

Retiring TD

A small amount of technology debt is often retired as part of technical upgrades or maintenance releases. But in general, most mature companies grow the debt at a significant rate. Explaining the concept of technology debt to a non-IT person using Figure 1 can be challenging. Technology debt gets created because it is easy to hide in multilayered, complex systems.

The first step to retiring technology debt is to document the workarounds. The documentation should identify areas where the workarounds make innovation difficult, create process inefficiencies, allow security exploits, and increase IT operating costs. The IT function should use the existing BA framework to document and inventory the areas of concern. The second step is to effectively communicate and gain sponsorship from non-IT owners for the technology debt retirement projects. The third step is to ensure successful execution of funded TD remediation projects.

Consider the example of the OE system in Figure 1. The BA artifacts for the OE process would show the process flow of quotes, order placement, order confirmation, shipping notification, and so on. The supporting IT architecture artifacts would map the IT systems enabling the OE business processes. Similarly, the MRP system BA artifacts would show process flows for sales orders, forecast, planning, scheduling, order processing, and the like, with supporting IT systems artifacts.

The workaround of direct integration to the order processing module in the MRP system for order confirmation in the OE system creates process/system dependencies across OE and MRP business processes. This workaround would have been in response to the sales function demanding that a customer's order confirmation be immediate. Over time, the workaround results in not having enough product to meet the customer orders on the committed schedule, as the commitment was made while bypassing the MRP process flow. Correcting these types of issues caused by the workaround with more short-term fixes taxes the IT operating budget with reduced service levels.

The thin lines in Figure 1 represent hundreds of similar workarounds between functional business processes and supporting IT systems. The business and IT architecture artifacts can help educate non-IT executives on the existing process lock-ins between subprocesses across finance, operations, OE, business development, and so forth. One approach to determining the operating costs of supporting the hundreds of workarounds is to review the prior year's incident response and problem resolution logs to determine the resources expended to fix problems caused by the workarounds.

NOT A BURNING PLATFORM

Technology debt is insidious and grows like a giant underground fungus. On those occasions when a workaround is exploited to steal data, and the company gets unwanted publicity, tackling technology debt in the systems that were breached suddenly becomes a "burning platform," and enormous resources are expended to remediate the workarounds.

A company does not have to wait for a burning platform scenario to address technology debt. The situation described in Figure 1 occurs to varying degrees in most companies with enterprise systems. Retiring this debt will not happen without efforts being funded and run as separate projects with stakeholder sponsorship. In many instances, the sponsorship could run across multiple departments. It requires an organizational view of the business processes and supporting IT portfolio of applications.

The IT function supports the systems that enable business processes within a function (e.g., finance, sales, manufacturing, HR) and across the functions. Addressing technology debt within a function (e.g., finance) is relatively straightforward, as IT can inform and seek the CFO's sponsorship to remediate the debt. To remediate technology debt that is created across functions, as in Figure 1, IT needs to educate and gain sponsorship from multiple non-IT stakeholders. This is a challenging task, and having the BA artifacts in hand can help illuminate the problem areas.

If companies do not clearly identify initiatives that prevent, contain, and retire technology debt in their capital and operating budgets, they are only adding fuel to the fire. The process and technology lock-in that results will hinder innovation and increase operating costs.

ENDNOTES

[1]Ramasubbu, Narayan, Chris F. Kemerer, and C. Jason Woodard. "Managing Technical Debt: Insights from Recent Empirical Evidence." *IEEE Software*, Vol. 32, No. 2, March/April 2015.

[2]Business Architecture Guild (www.businessarchitectureguild.org).

[3]ITIL (www.itil.org.uk).

Ram Reddy is currently an independent management consultant. Mr. Reddy most recently served as Vice President for IT Strategy and Projects at Jacobs Engineering Corporation (JEC). He was awarded Computerworld*'s Premier 100 IT Leaders award in 2015 for helping implement a "follow the sun" IT support model for JEC's global workforce.*

Prior to joining JEC, Mr. Reddy held multiple IT leadership positions, including Chief Enterprise Architect and Director of Enterprise Applications at SAIC (a major aerospace/defense services firm), where he worked with senior executives to deliver and maintain IT applications in support of their business operations. Earlier, Mr. Reddy was the President of Tactica Consulting Group, a technology and business strategy consulting firm. He has advised Fortune *500 firms on strategic use of technology, restructuring the IT function for competitive agility, customer relationship management, and supply chain systems. Before Tactica, he was the CIO of a Tier 1 supplier in the automotive sector.*

Mr. Reddy's work experience spans industries, including manufacturing, contingent staffing, mass merchandising, procurement, and financial services. He is the author of Supply Chains to Virtual Integration *and has written extensively on realizing strategic business goals through information technology. He can be reached at ramxreddy@outlook.com.*

Vendor-Driven Technical Debt: Why It Matters and What to Do About It

by Mohan Babu K

The term "technical debt" is generally used to describe the burden created by decisions to cut corners when designing and coding software. The catchy metaphor is attributed to Cutter Fellow Ward Cunningham, who helped us think about how quick-and-dirty solutions set us up for debt that has to be paid back with interest.[1] Technical debt driven by software vendors is a less frequently discussed but significant variation on the theme.

I would like to broaden the conversation around technical debt to include the challenges of keeping up with software vendors' lifecycles. Such vendor-driven technical debt requires the continual attention of CIOs and technology executives, who need to balance limited budgets to cope with technical debt.

In this article, we will examine aspects of the problem and evaluate some of the techniques to address and repay technical debt.

TECHNICAL DEBT IN CONTEXT

Technical debt attributable to vendor upgrade cycles is a topic of intense debate in corporate IT groups, as well as enterprise architecture forums and online communities.[2] Enterprises of all sizes buy or license software products, solutions, and tools from vendors. These products range from small investments in worker productivity tools to large investments in ERP systems, CRM systems, databases, and specialized solutions designed to meet specific functional needs. The decision to implement a version of the software — for example, Oracle Database 12*c* Release 1 or SQL Server 2008 R2 or SAP ERP 6.0, EP 4 — is generally a strategic one, requiring considerable analysis, planning, and resources. Such decisions are taken at a particular point in time, while considering the organization's business needs and constraints in the technology landscape.

Software vendors and solution providers continually upgrade their product offerings, promising newer technical and functional capabilities. In order to provide support, vendors expect clients to keep up with their upgrade cycles. Software support from a vendor is critical to IT operations in order to:

- **Apply relevant security patches and technical fixes.** The source code for licensed software is generally owned by the software vendor, which is in a position to apply relevant updates, patches, and technical fixes. Such patches — implemented and tested centrally — are pushed to all clients.
- **Ensure compliance and controls over IT systems.** CIOs are required to certify to their stakeholders that systems meet relevant regulatory and compliance requirements. For example, the US Sarbanes-Oxley Act (SOX) requires the CEOs and CFOs of public companies to attest to the accuracy of financial reports and establish adequate internal controls over financial reporting. Passage of SOX resulted in an increased focus on IT controls, as these support financial processing and therefore fall into the scope of management's assessment of internal controls. Ensuring software systems are upgraded and managed as per vendor recommendations is a key aspect of compliance.

Upgrading to a newer version of software recommended by the vendor requires a deliberate impact assessment to understand the potential impact to systems upstream or downstream. Such an upgrade may have to be orchestrated with changes in the rest of the IT landscape; for example, during a large prescheduled program.

The vendor's upgrade cycles may not align with the customer's business or technology change cycle. Therefore, the organization's IS and business stakeholders may consciously opt out of a vendor-driven upgrade cycle, in effect taking on a technical debt that will need to be repaid by upgrading later. After a few cycles of not upgrading, the software may fall behind the vendor's support cycles, and the vendor may demand a penalty for supporting older versions. Some vendors call this "extended support," and it can be expensive.[3] In some cases, after adequate notice,

vendors may stop support of versions going back several generations, such as when Microsoft announced the end of the support lifecycle for Microsoft Windows NT 4.0 Server.[4] Such action by the vendor may force the enterprise to repay the debt by upgrading the solution or looking for an alternative. Repaying such technical debt by planning an upgrade can be expensive and disruptive to business.

WHY DO ORGANIZATIONS TAKE ON TECHNICAL DEBT?

IT executives and leaders are increasingly using the technical debt metaphor to articulate the hidden costs and tradeoffs associated with decisions that impact business capabilities and technologies in the organization. Martin Fowler, a leading software expert, classifies technical debt into four types, along two axes: reckless or prudent, and deliberate or inadvertent.[5]

Fowler's elegant model is focused on software development, but it can be extended to take a broader view of the enterprise architecture and application portfolio (see Figure 1):

- **Reckless/deliberate debt.** Project teams, including managers and architects, may sometimes give in to time-to-market pressures without adequate analysis or forethought. Such decisions may be reckless, but they are deliberate. For example, a business unit might opt to implement a different version of a supply chain solution, unable to wait for the global ERP to be rolled out across the enterprise. Such a debt will have to be repaid when the standard version of a global solution is finally planned for implementation at the business unit.
- **Prudent/deliberate debt.** A project team may take on a deliberate short-term debt with clear plans to address it. Such a prudent decision may be taken to enable the business to react to an external market change. For example, toward the end of 2015, when the global price of oil suddenly dipped below US $30 a barrel, oil drillers began reacting by shelving projects and shutting operations. Across the oil and gas industry, some $400 billion in expected investment was cancelled or delayed.[6] Amid widespread and abrupt cost-cutting pressures, executives may find it easy to postpone technology upgrade projects. In the oil and gas sector, IT executives are putting off technology upgrades, deliberately taking on technical debt that they hope to repay when the sector turns around.
- **Reckless/inadvertent debt.** This is a likely scenario in loosely governed organizations where teams are either ignorant of the consequences of taking on technical debt or recklessly flout the guiding principles. An example is that of a design team involved in a global SAP supply chain rollout that agrees to requests from the business to customize the solution, ignoring the fact that such custom development on a product will make future upgrades more difficult and expensive. In a review of enterprise systems, University of Pittsburgh researchers Narayan Ramasubbu and Chris F. Kemerer explain that "software update patches supplied by vendors as part of their maintenance plans could be incompatible with the customizations implemented by clients."[7]
- **Prudent/inadvertent debt.** Project teams may take a prudent decision that satisfies the functional needs at a point in time but may inadvertently miss other dependencies in the organization. An example is a global pricing program in North America that misses a key dependency with a corresponding global CRM solution implemented in Europe. Despite adequate due diligence, the inadvertent omission may result in technical debt when the global solutions are implemented across the regions.

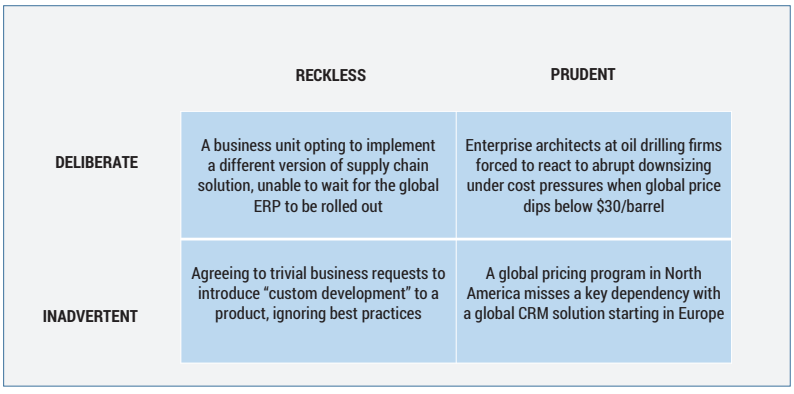


Figure 1 — Real-world examples of Fowler's four types of technical debt.

HOW TO MOVE FORWARD: ACKNOWLEDGE AND COMMUNICATE THE PROBLEM

Enterprise architects and IT executives recognize the problem of technical debt, but they may not have the resources and funding to deal with it. They need tools and techniques to communicate the problem to their stakeholders and engage with them.

In order to communicate with stakeholders, one needs to identify the magnitude and the span of the impact. In many cases, the impact of a vendor-recommended upgrade may be limited to a software platform used locally or regionally. Software platforms used across business units may have been designed to operate in a loosely coupled manner, in which case the effect of upgrades may also be limited. Such a debt may be addressed with line-of-business application owners or functional leaders.

In some cases, the problem may be widespread, attributable to an organizational culture of taking on debt during periods of high growth. For instance, the impact of not upgrading a critical ERP application may span business functions. Upgrading such a vital application platform may require engagement of senior business leaders who can influence their peers. Deutsche Bank offers one example of an engaged senior leader communicating the technical debt problem. During a press conference, co-CEO John Cryan publicly acknowledged the root causes of the problem of technical debt at the bank:

> About 80 percent of our 7,000 applications were outsourced to a multitude of different vendors. Design was basically done in silos or by joint standards where they were either hardly used or not used at all. The result is that our systems do not work together, and they are cumbersome when it comes to the application and often incompatible. A figure that worries me in particular ... is that about 35% of the hardware in the data centers has come close to the end of its lifecycles or is already beyond that.[8]

Most executives may not be inclined to use such a public forum to acknowledge their problem, but they can certainly take a page out of Deutsche Bank's playbook. Acknowledging the problem of technical debt, and communicating that problem to stakeholders, helps set a foundation for resolution. Such acknowledgement by IT and functional executives can act as a call to action that enables project teams to find a way forward.

ADDRESSING TECHNICAL DEBT

In this section, we will look at some of the proactive and reactive recommendations for dealing with vendor-created technical debt.

Proactive Engagement

IT executives should proactively engage with stakeholders to continually plan for upgrades to minimize widespread occurrence of software version backlogs. The most effective technique for addressing technical debt is to embed vendor-recommended version upgrades into existing governance and IT management processes, rather than trying to deal with such upgrades in isolation. Specific strategies include:

- **Align vendor roadmaps with business strategy.** Continual business engagement is an effective way to align vendor upgrade recommendations with functional requirements. Business strategies and roadmaps may indicate the need for newer functional capabilities and investment plans. During review discussions, IT executives should highlight existing technical debt and upgrade recommendations that may also be addressed in the roadmaps. Such interactions will ensure stakeholders are continually educated about technical debt and actively participate in tackling it.
- **Enterprise architecture governance.** An effective Architecture Review Board (ARB) will periodically review business change proposals against functional and technical roadmaps. In a previous *Cutter IT Journal* article,[9] I explain how an ARB can:
  - *Highlight architecture risk.* Do so by enforcing the architecture principles and best practices during reviews
  - *Ensure project alignment with predefined roadmaps* to enable long-term strategies. By taking a cross-functional view and ensuring visibility into projects, a well-functioning ARB should be able to identify situations in which project teams inadvertently or deliberately take on technical debt. When identifying a potential debt, the ARB should also help teams communicate the impact and guide them in tactics for addressing it.
- **Application architecture and design techniques.** IT executives and enterprise architects have the most influence when new software platforms are being introduced into an organization. Design techniques that can minimize the impact of future technical debt include:
  - *Loose coupling*. Thoughtful loose coupling of systems and interfaces and adopting standard, vendor-neutral integrations can pay dividends in the long run. Such loosely coupled application platforms and components could be upgraded without wider impact to the landscape.

- *Cloud hosting.* Cloud-based offerings from software vendors continue to mature. Moving to a platform as a service (PaaS) or software as a service (SaaS) offering from a vendor is a conscious way to cede control of application management in return for assured business capability and functionality. Doing so minimizes the need to manage vendor-driven upgrades. However, IS managers and teams may still need to keep track of integrations and data transfer that may be impacted by such upgrades.
- *Configuration before customization.* Most commercial software solutions provide the ability to configure business processes/workflows. Some functional requirements may be very unique and require customization of the platform via writing additional code. To the maximum extent possible, such customizations should be avoided or minimized.

Reactive Remediation

Proactively addressing issues arising from vendor-driven upgrades may minimize the technical debt challenges in the future. However, not all such efforts can prevent technical debt. An ARB, for instance, may agree that a project should take on a prudent/deliberate debt, with a clear means of addressing it later. There are several ways organizations can deal with technical debt after it occurs:

- **Pay as you go.** An effective way to cope with debt is to periodically pay it off. However, organizations will have more than one vendor solution and will have to orchestrate upgrades across vendor platforms and solutions in the landscape. Such upgrades should be planned to coincide with other changes and scheduled maintenance in order to minimize business disruption. Effective techniques for managing periodic upgrades include:

CASE IN POINT: A MULTINATIONAL REACTS TO PLATFORM END OF LIFE

Some time ago, I was engaged with a financial services company involved in an application portfolio-rationalization program. The stated objective of the program was to simplify the company's IT landscape and minimize the redundant and duplicate applications. By rationalizing the application portfolio, the organization expected to save over 30% in operating costs and to benefit from newer, more efficient infrastructure.

During an initial assessment, the team discovered that nearly a quarter of the 1,200 applications were running on Windows 2003–based servers. At that time, Microsoft announced an end of life for the operating system software.[1] In order to remain "supported," applications had to be migrated out of existing servers. The objective of the program thus expanded from application portfolio review to include legacy modernization and repayment of technical debt built up in the legacy application landscape.

DESIGN PRINCIPLES

In order to proactively address the technical debt and minimize future disruptions, the team defined a few key principles:

- **SaaS when possible.** Consider migration to an application vendor's SaaS offering.
  - *Outcome:* Many of the software solutions the organization had acquired over the years were now being offered as SaaS services. The team began reviewing all such vendor offerings and included them in the transition plan.
- **Cloud-first upgrade.** Where possible, the applications should be upgraded to run on the cloud.
  - *Outcome:* The team designed a private cloud on Microsoft's Azure. Applications running on Windows 2003 servers were evaluated for "cloud suitability" and included in the roadmap.
- **Configure before customizing.** Customization should be avoided or minimized.
  - *Outcome:* The team discovered that some of the customizations done in the past were no longer required. In a few instances, business users had to be convinced to do away with customized functionality. Most of the applications were upgraded without customizations.

THE RESULT

The program delivered the simplification objective, reducing operating cost. In addition, this initiative also started the organization on a journey to migrating applications to the cloud.

The vast majority of applications running on Windows 2003 servers were upgraded to operate on a virtual private cloud on Microsoft's Azure cloud platform. Only a small proportion of applications with specific design constraints were determined unsuitable for migration. After repayment of existing technical debt, the upgrade minimized the possibility of falling behind vendor upgrade roadmaps. This ensured that infrastructure will be closely aligned with recommendations from the vendor (Microsoft) that is responsible for managing the platform.

[1]See www.microsoft.com/en-us/server-cloud/products/windows-server-2003.

- *Defined policy on technology debt.* Organizations can institute a policy explicitly stating that critical software will not get behind vendor-supported versions. For example, the policy might state that "all software must be version n-2 or higher," where "n" is the most current vendor version. Such a policy should also be backed up with executive support and funding to ensure compliance.
- *Dependency matrix.* Organizations can use a dependency matrix to show compatibility of software and infrastructure versions recommended by vendors. The matrix should include most, if not all, of the dependent platforms in the landscape and should be used to guide upgrades.
- *Cleanup releases.* Where feasible, IT leaders should plan for "cleanup" releases to remediate the existing technical debt. If it is not feasible to plan such releases, one could explore opportunities to include upgrades with other scheduled changes.
- *Budgeting for technical debt.* The dependency matrix and other tools can be used to help estimate the effort involved in handling such upgrades.

- **Landscape simplification.** IT executives should seek opportunities in large transformation programs to tackle technical debt. An example of such a landscape review is illustrated in the case study (see sidebar "Case in Point: A Multinational Reacts to Platform End of Life").
- **Engagement with vendors.** Software vendors provide several venues for engaging with clients. These range from informal online forums, wikis, and discussion boards to more formal surveys, technical conferences, and product planning sessions. CIOs and executives of enterprises with a large investment (or installation) of a vendor solution might benefit from engaging in such formal forums to (1) influence backward compatibility of the product in the vendor roadmap, and (2) seek assistance from the vendor in addressing technical debt attributable to its product.
- **Other mitigation.** An organization may review the impact of vendor end-of-life support and take actions to mitigate the risk, which could include:
  - *Self-insuring.* The organization might manage without vendor support if the software is noncritical, there is no regulatory reason to seek support, and if analysis of recent history indicates very few instances where vendor support was actually sought.
  - *Outsourcing.* The vendor may sometimes sell or license the source code to enable the client organization or a third party to support an aging software product.

CONCLUSION

Technical debt attributable to software vendors is an ongoing challenge that IT executives need to manage. As with financial liability, it may not be practical or prudent to avoid all technical debt. With awareness of the existence and impact of such debt, however, IT executives can plan to effectively address it.

ENDNOTES

[1]Cunningham, Ward. "Ward Explains Debt Metaphor" (video; http://c2.com/cgi/wiki?WardExplainsDebtMetaphor).

[2]Ramasubbu, Narayan, and Chris F. Kemerer. "Technical Debt and the Reliability of Enterprise Software Systems: A Competing Risks Analysis." University of Pittsburgh, February 2015 (www.pitt.edu/~ckemerer/techdebt-MS-accepted%202015.pdf).

[3]Forum discussions on standard and extended support include "SAP Standard Support Fee" (http://scn.sap.com/thread/3187836) and "Oracle: Waiver of Extended Support" (www.oracle.com/us/support/library/extended-support-faq-515467.pdf).

[4]"Q&A: Support for Windows NT Server 4.0 Nears End; Exchange Server 5.5 to Follow in One Year." Microsoft (http://news.microsoft.com/2004/12/03/q-exchange-server-5-5-to-follow-in-one-year).

[5]Cairns, Chris, and Sarah Allen. "What Is Technical Debt?" 18F, US General Services Administration (GSA), 4 September 2015.

[6]Crooks, Ed, and Chris Adams. "Oil Majors' Business Model Under Increasing Pressure." *Financial Times*, 14 February 2016.

[7]Ramasubbu and Kemerer (see 2).

[8]Boulton, Clint. "Deutsche Bank Digging Out of Technical Debt, While Moving to the Cloud." *CIO*, 11 November 2015.

[9]Babu K, Mohan. "Enabling Successful EA Governance with an Architecture Review Board." *Cutter IT Journal*, Vol. 28, No. 2, 2015.

Mohan Babu K is an Enterprise IS Architect at Syngenta, based in Greensboro, North Carolina, USA. He has spent nearly two decades in technology management and has gained a strong insight into the lifecycle of portfolio management and the global delivery model. Having lived and worked in a dozen countries on three continents, Mr. Babu K has also gained an international perspective on business and society. His viewpoints and papers have been published in several international technical and nontechnical journals, including Cutter IT Journal, Business Integration Journal, Research-Technology Management, IEEE Computer, Computerworld, ACM Ubiquity, *and* Sourcingmag, *among others. Mr. Babu K is the author of a book on globalization titled* Offshoring IT Services: A Framework for Managing Outsourced Projects. *He can be reached at mohan@garamchai.com.*

John Heintz, Senior Consultant

CONSULTING

Technical Debt Assessment and Valuation

A Universal Tool for Evaluating, Governing, and Managing Software Projects

Do You Need a Technical Debt Assessment?

Are you a CIO looking to ensure delivery over development?

A CTO in search of early warning signs your development project is in trouble?

An M&A/due diligence investigator in need of assurance that the code you're acquiring isn't toxic?

A CEO responsible for governing the development process effectively and ensuring the execution of corresponding go-to-market plans in a reliable manner?

A venture capitalist determining how much (more) money to invest in your portfolio company?

Cutter's Technical Debt Assessment and Valuation is customized to meet your specific needs. For details, contact your Cutter Account Executive at sales@cutter.com or +1 781 648 8700.

Technical debt is a real cost. Whether you're looking at it from the perspective of a venture capitalist or CEO, or from the viewpoint of a CIO or CTO, or you're trying to determine if a merger or acquisition makes sense, knowing how much money is required to "pay back" your software's technical debt may be the very factor that proves your decision to be a good one or a very costly one.

In a Technical Debt Assessment and Valuation, Cutter's Senior Consultants — led by John Heintz — examine the quality of the software under examination through technical and business lenses. Whether that code is your own, has been developed by an acquisition candidate, or by a company you're investing (more) in, Cutter's Technical Debt Assessment and Valuation will enable you to:

- Get the vital answer to the question, "Is your software an asset or a liability?"
- Know how much (more) money you will need to invest in order to fix the code
- Get data and insights you need to guide the fix-it process for the software
- Identify projects that are likely to get in trouble at an early stage of the software lifecycle
- Determine if the technical debt is keeping your software development staff from responding quickly and effectively to customer requests

Plus, you'll get the tools you need to govern the software development process on an ongoing basis to avoid the expense of future technical debt.

Do you know the true value of your software? Are you sure? Does your value calculation include technical debt?

In a Technical Debt Assessment and Valuation, Cutter's Senior Consultants will identify the architecture, design, coding, testing, and documentation deficits that constitute technical debt. The assessment combines static code analytics with dynamic program analytics to give you "x-rays" of the software being examined at any desired granularity. You'll get a report and/or presentation that provide you with a dollar figure you can plug into your financial models so that you can objectively analyze your critical software assets. Easy-to-understand graphics depicting the quality of your code and the cost of your technical debt will enable your team to zero in on the most hazardous projects and fix them in a prioritized manner. And you'll get operational recommendations that take into

Agile Product Management & Software Engineering Excellence

account various qualitative and quantitative factors that characterize your software development process. These recommendations will help you make the best decisions about your ongoing strategy for this software development effort.

Cutter's Technical Debt Assessment and Valuation is most effective as an on-premises engagement. However, it can also be done as a largely off-premises engagement based on a snapshot of the code. For more details, or to arrange your Technical Debt Assessment and Valuation, contact your Cutter Account Executive at +1 781 648 8700 or sales@cutter.com.

"The ability to make the debt metaphor work for your advantage depends upon you writing code that is clean enough to be able to refactor as you come to understand your problem."

— Ward Cunningham, Cutter Fellow

Cutter Research & Opinion on Technical Debt

"Avoiding System Bankruptcy: How to Pay Off Your Technical Debt"
(https://www.cutter.com/article/avoiding-system-bankruptcy-how-pay-your-technical-debt-424966)

"Self-Insuring Your Software"
(https://www.cutter.com/article/self-insuring-your-software-425106)

"Technical Debt" from *Cutter IT Journal*
(www.cutter.com/itjournal/fulltext/2010/10/index.html)

"Delving into Technical Debt"
(www.cutter.com/project/fulltext/updates/2011/apmu1120.html)

"Quantifying the Start Afresh Option"
(http://blog.cutter.com)

"Enterprise Architecture, Technical Debt, and Technical Paralysis"
(www.cutter.com/content/architecture/fulltext/advisor/2011/ea110817.html)

"To Release No More or To 'Release' Always: Part II — Toward a New Business Design for Software"
(www.cutter.com/project/fulltext/updates/2008/apmu0823.html)

"The Agile Triangle — Quality Today and Tomorrow"
(www.cutter.com/project/fulltext/advisor/2010/apm100401.html)

Agile Assessment

Cutter's Technical Debt Assessment and Valuation is extremely synergetic with our Agile Assessment, a quantitative and qualitative analysis of an organization's use of Agile methods, its software engineering practices, and its project management skills and capabilities. When the two are conducted jointly, Cutter will present your team with a composite plan for fixing software quality deficits and software process deficits in tandem.

Cutter Consortium
37 Broadway, Suite 1, Arlington, MA 02474-5552, USA
Tel: +1 781 648 8700; Fax: +1 781 648 8707; www.cutter.com; sales@cutter.com

About Cutter Consortium

Cutter Consortium is a truly unique IT advisory firm, comprising a group of more than 100 internationally recognized experts who have come together to offer content, consulting, and training to our clients. These experts are committed to delivering top-level, critical, and objective advice. They have done, and are doing, groundbreaking work in organizations worldwide, helping companies deal with issues in the core areas of software development and Agile project management, enterprise architecture, business technology trends and strategies, enterprise risk management, metrics, and sourcing.

Cutter offers a different value proposition than other IT research firms: We give you Access to the Experts. You get practitioners' points of view, derived from hands-on experience with the same critical issues you are facing, not the perspective of a desk-bound analyst who can only make predictions and observations on what's happening in the marketplace. With Cutter Consortium, you get the best practices and lessons learned from the world's leading experts, experts who are implementing these techniques at companies like yours right now.

Cutter's clients are able to tap into its expertise in a variety of formats, including content via online advisory services and journals, mentoring, workshops, training, and consulting. And by customizing our information products and training/consulting services, you get the solutions you need, while staying within your budget.

Cutter Consortium's philosophy is that there is no single right solution for all enterprises, or all departments within one enterprise, or even all projects within a department. Cutter believes that the complexity of the business technology issues confronting corporations today demands multiple detailed perspectives from which a company can view its opportunities and risks in order to make the right strategic and tactical decisions. The simplistic pronouncements other analyst firms make do not take into account the unique situation of each organization. This is another reason to present the several sides to each issue: to enable clients to determine the course of action that best fits their unique situation.

For more information, contact Cutter Consortium at +1 781 648 8700 or sales@cutter.com.

The Cutter Business Technology Council

The Cutter Business Technology Council was established by Cutter Consortium to help spot emerging trends in IT, digital technology, and the marketplace. Its members are IT specialists whose ideas have become important building blocks of today's wide-band, digitally connected, global economy. This brain trust includes:

- Rob Austin
- Ron Blitstein
- Tom DeMarco
- Lynne Ellyn
- Vince Kellen
- Tim Lister
- Lou Mazzucchelli
- Ken Orr
- Robert D. Scott