Technical Debt and Design Death

Kane Mar and Michael James

At a Scrum gathering in 2006, Ken Schwaber and Jeff Sutherland discussed technical debt and design death.

Although the concepts are nothing new[1], Schwaber and Sutherland identified a number of characteristics that represent significant contributions to the dialogues surrounding these subjects. Additionally, they introduced some very helpful graphs that a company or department could use to gauge the "health" of its software products. As such, we will first discuss the concepts of technical debt and design death as well as their telltale characteristics. Secondly, we will turn to Schwaber and Sutherland's insights to consider technical debt's impact on legacy systems and offer a few coping strategies.

ACCRUING DEBT

Over the course of a project, it is tempting for an organization to become lax regarding software quality. Most commonly, this results when teams are expected to complete too much functionality in the given time or quality is simply not considered a high priority characteristic of the software. To illustrate how dangerous this is to the health of the software as it evolves, let us consider the following scenario involving an "average" project team at work on a company's flagship product (Figure 1). Let's assume the team isn't making as much progress as expected (Figure 2).

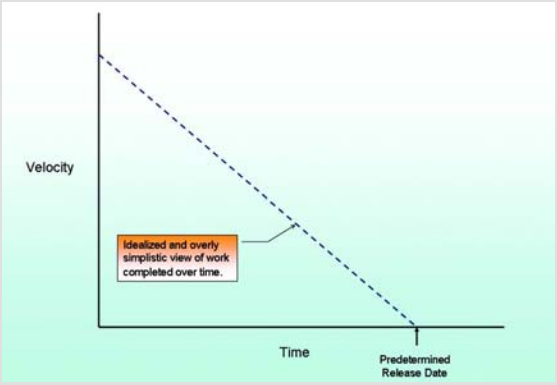

Figure 1: Expectations at the start of a project.

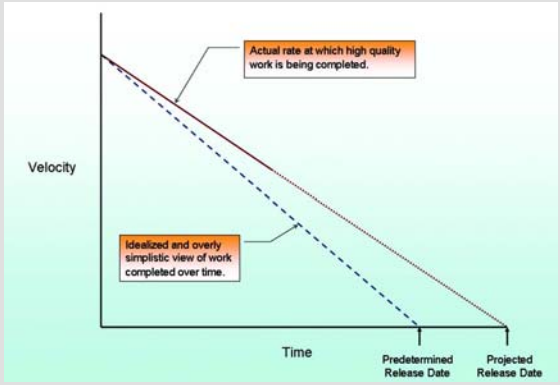

Figure 2: Reality has a way of making its presence known!

Noting the disparity between the projected rate of work and what has actually been accomplished, senior management wants to know why the team's progress has not met expectations. Subtle (or not so subtle) pressure is placed on the team to complete the project by the predetermined release date (Figure 3). When the team makes the deadline, the project manager interprets the work as accelerated productivity, explaining that "the project team has learned to work more efficiently." In actuality, the project team reduced the quality of the software in order to finish the job on time. The difference between what was delivered and what should have been delivered is known as technical debt (Figure 4).

When a second team continues to develop with the same code base, the consequences of technical debt become increasingly evident. As the new team begins work on the next phase of the project, it faces the same time constraints. However, the team now faces a significant disadvantage because the code base it must work in is poorly written, resulting in an even slower rate of

The difference between what was delivered and what should have been delivered is known as technical debt.

progress than the first team. To “motivate” the team to deliver on time, the same management techniques are applied and pressure is brought to bear on the project team to meet the deadline. They finish “on time and within budget,” but, according to the project manager, weren’t nearly as “efficient” as the first team. Rather than address the looming technical debt and repair the code base, the team has begun a cycle of worsening code quality. Put another way, a layer of cruft has been built upon a layer of cruft (Figure 5).

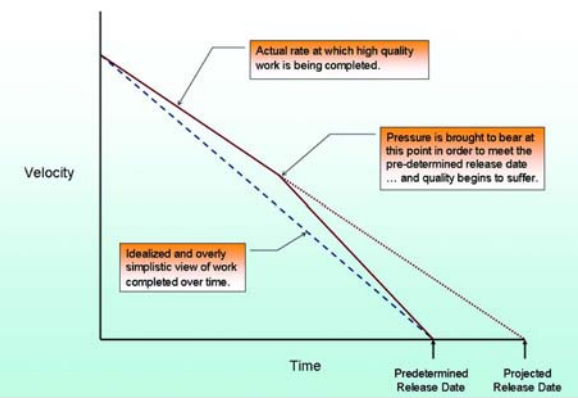

Figure 3: Pressure is placed on the project team “to be more efficient.”

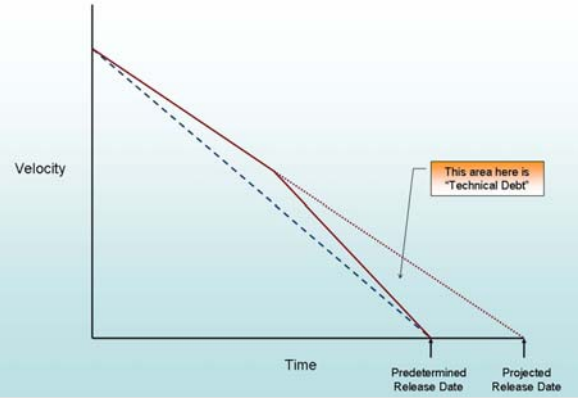

Figure 4: Technical debt.

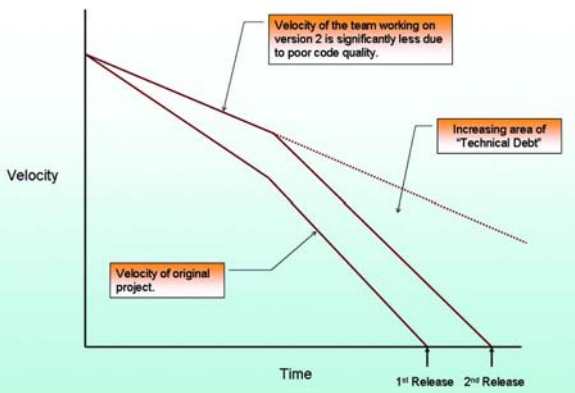

Figure 5: Cruft on top of cruft.

A BAD CASE OF CRUFT

The concept of software complexity as debt was introduced by Ward Cunningham in an experience report for OOPSLA Conference in 1992[2]. It has since gained a large following and has been expanded to include technical debt and design death in addition to other concepts[3].

Technical debt is simply defined as deferred work that is not directly related to new functionality, but necessary for the overall quality of the system[4]. Examples of this include delaying upgrades to necessary tools and frameworks, delaying the refactoring of overly complex system components, and so on. If this technical work is necessary for the health of the system, then what happens when it is ignored for too long? What happens when several layers of cruft accumulate and there are no unit tests to aid refactoring?

DESIGN DEATH

To consider the long-term effects of technical debt, let’s return to the hypothetical scenario—but

two years on in the development cycle. The company's flagship product is now on its fourth version and its scope has expanded significantly from its original iteration. Consequently, the code base is very, very complicated. In fact, there are sections of the code that can only be modified by one or two specialized developers in the entire company. Some classes are as long as several thousand lines of code.

If this is an unthinkable reality in your organization, consider yourself fortunate. I recently met with a client that described its classes as "very complicated." Imagining the worst case scenario, I asked, "How long are they? Three or four thousand lines?" An entire room full of developers burst into laughter. Puzzled, I asked what was so funny. "We're lucky if our classes are three or four thousand lines," one developer explained. "Try ten thousand lines, plus!"

Cruft (n):

1. An unpleasant substance. The dust that gathers under your bed is cruft; the TMRC Dictionary correctly noted that attacking it with a broom only produces more.

2. The results of shoddy construction.

When code reaches that level of debt, the effort required to change any part of it becomes so significant that it is described as "design dead." At the Scrum gathering, Ken Schwaber identified three attributes of code that has entered design death.

1. The code is considered part of a core legacy system, in which its functionality is connected to so many other parts of the system that it's impossible to isolate (and hence reduce) any one component.
2. There is either no testing or minimal testing surrounding the code. Although it may sound redundant, it is necessary to point out that without comprehensive unit tests, it is impossible to refactor the code to a more manageable state.
3. There is highly compartmentalized knowledge regarding the core/legacy system, supported by only one or two people in the company.

Now that we have identified the telltale signs of technical debt and design death, I will steer my discussion to consider coping strategies for companies facing these challenges or even more complicated situations, such as multiple legacy systems.

LEGACY SYSTEMS

It's not uncommon for organizations, including banks and insurance companies, to have a multitude of legacy systems, many of which are "Green Screens" (i.e., "dumb" terminals connected to an IBM mainframe, running some Cobol database application). Even relatively new companies (less than 20 years old) can have legacy systems (although they often use the term "core" rather than "legacy" to describe them).

Legacy System (n):

A computer system or application program which continues to be used because of the cost of replacing or redesigning it and often despite its poor competitiveness and compatibility with modern equivalents. The implication is that the system is large, monolithic, and difficult to modify.

Anyone who works on these legacy (or core) systems will identify with the characteristics of design death Schwaber identifies. They would likely recognize a fourth characteristic, which I would add to his list.

4. The legacy system is in an unknown state.

By that, I mean it can be difficult—sometimes impossible—to determine the state of the system at any given point in time. In such a case of design death, installing the system and recovering it after a failure becomes such a labor-intensive operation which can only be performed by an extremely limited number of an organization's employees that it can seem like a black art.

In thermodynamics, "entropy" refers to the randomness of the components of a system. When the term is applied to software it is considered a measure of disorder. Thus entropy in software is the result of changes made to the code base, including bug fixes, updates to existing functionality, and the addition of new functionality. But over a period of time, these small changes can snowball to create a system that is difficult to change, overly connected to external systems, and lacks clear

delineation of functionality[5].

Because competition reduces the value of existing software over time, companies must constantly add new functionality to their software in order to maintain value. There are countless sources that document how a relentlessly competitive marketplace demands that software (or the service provided by that software) must continually increase in value to remain relevant. It follows that software must be continually changed (i.e., updated) to remain relevant[6]. By necessity, ongoing development of a product increases the entropy of the system—and the cost of change along with it.

As a developer, nothing was more challenging and frustrating than to be added to a project late in its development. Working with code afflicted by technical debt made me feel like an archaeologist trying to clean up an artifact that will crumble unless I work slowly and cautiously.

In the last six months alone, I've talked to at least three companies that are planning to rewrite their systems from the ground up. These are three very different businesses with unique markets and business models. When I asked them why they decided to entirely overhaul their systems, they responded, across the board, that the cost of change was too high. Automated tests (unit tests, acceptance tests, FIT/FITness tests, etc.) help decrease the cost of change and help the system reach a known state, but without an adequate automated testing framework, the task of changing the legacy system becomes increasingly expensive.

Let us consider a fictitious company, which we will call "NFI," that has failed to add value to its product. The graph in Figure 6 shows a gradual decline in revenue. These numbers are also fictional.

Figure 6: NFI has had a consistent decline in revenue.

A MATTER OF CHOICE

The management at NFI knows that functionality must be added to its system in order to remain competitive. The organization has three options:

1. Add functionality to the core system.

This would be prohibitively expensive, as the ongoing addition of new functionality exponentially raises the cost of software. This is rarely a practical or cost-effective solution.

2. Introduce a temporary solution that would allow NFI to use the existing legacy system in addition to the new functionality.

This would not address the underlying problem, but might give the company more time. One way this is commonly executed is by building a Web service layer on top of the existing legacy API. New functionality is then constructed alongside the existing system and they are both integrated at the Web services level. But what happens when new data needs to be added to the legacy data model?

3. Reconstruct the existing functionality using new platforms and technology.

This solution addresses the underlying problem. It is, however, more expensive than option 2, but less expensive than option 1.

So how does NFI decide which option is most suitable for its particular situation?

Even if a manager has difficulty recognizing the value of repaying debt each Sprint, he or she could hardly argue against the potential cost of rewriting a legacy system from scratch.

GRAPHIC EVIDENCE

For the sake of illustration, let us assume that NFI decides to rewrite existing functionality. We can draw a graph of Revenue (declining) and Functionality (increasing) over time (Figure 7).

To measure functionality, I have chosen to use Story Points, but, provided they are consistent, any unit of measurement can be used. In this scenario, I've also made the assumption that any rewritten system is easier to maintain with a lower cost of change. These are reasonable assumptions if some Extreme Programming (XP) practices [such as Continuous Integration (CI), Test Driven Development (TDD), Refactoring, etc.] are used.

Based on the graph in Figure 7, the rewritten functionality will be completed in mid-2004. This situation is a viable option for NFI. If the company has time, then this approach should be investigated. But what happens when the time needed to rewrite the most basic functionality of the legacy system exceeds the time the company is projected to generate positive revenue? (Figure 8.)

Figure 7: Revenue declines as functionality increases.

Figure 8: What happens when the cost of rewriting functionality takes longer?

DECIDING FACTORS

Clearly, a company in this situation has some difficult decisions to make. There might be a temporary solution. For example, perhaps NFI could use the existing system while simultaneously building a new product. Conversely, NFI might decide to borrow money to fund the rewrite or the company might simply return the remaining financial resources to its shareholders. Whatever the final decision, it is possible to present management with a number of potential courses of action by constructing a few simple graphs.

HOW TO SURVIVE TECHNICAL DEBT

As a developer, nothing was more challenging and frustrating than to be added to a project late in its development. Working with code afflicted by technical debt made me feel like an archaeologist trying to clean up an artifact that will crumble unless I work slowly and cautiously. In those situations, I frequently complained that the software was poorly documented, but, in reality, no amount of documentation could have helped, even if I trusted it[7]. I was always terrified that a "fix" I made would just trigger a new regression failure.

The only way to ensure that developers - and organizations - do not find themselves in a similar situation is to incrementally add automated tests. Not only does this increase velocity, it also reduces risk by preventing developers from "flying blindly" as they make changes to the code base. While it's unlikely that an organization can afford—in terms of time or money—to implement all of these tests at once, an organization should prioritize the test

Large amounts of technical debt must be made visible as first class Product Backlog Items, to be split many times and prioritized along with the business stories. If stakeholders can't see why they should be concerned, it is doubtful they will want to allocate a team's effort toward an activity that does not visibly move a project forward.

coverage to cover areas where it is most needed and incrementally add more tests as more business functionality is introduced.

Some teams or organizations may wish to formalize this process by adding a "tax" on each business work item accepted into a Sprint to guarantee that technical debt is "repaid." Acceptance criteria for such a Product Backlog Item might include:

• "Automated tests have been added to all existing code touched by the added code" or

• "We left it cleaner than we found it."

Large amounts of technical debt must be made visible as first class Product Backlog Items, to be split many times and prioritized along with the business stories. This approach will require the developers and ScrumMaster to explain each item's impact on development velocity in terms the business stakeholders can understand, so that they are prioritized appropriately. If stakeholders can't see why they should be concerned, it is doubtful they will want to allocate a team's effort toward an activity that does not visibly move a project forward.

Some parts of a legacy system may never receive test coverage. That might be acceptable if those parts aren't involved in the changes and have proven to work in the past through years of manual testing or live deployment.

MAKE LARGE TECHNICAL DEBT VISIBLE IN THE PRODUCT BACKLOG

The idea that technical debt items appear as first-class Product Backlog Items might appear to contradict the rule of thumb that all PBIs (or stories) possess directly measurable business value. At a recent Scrum Exchange, I challenged participants to share stories that could not be split into vertical slices. A particular participant was very attached to the idea that none of his stories could be split small enough to complete in a single Sprint because his system had fragile global tables inhibiting progress in any direction. Other participants pointed out that this problem itself could be split. This participant was well versed in how he could do this, having read Michael Feathers' book "Working Effectively with Legacy Code". But he couldn't imagine including the technical debt itself on the Product Backlog. Although repaying technical debt on an ongoing basis may not "advance" development of the project itself, it is a precautionary measure that preserves code quality and protects an organization's resources in terms of time and money. Even if a manager has difficulty recognizing the value of repaying debt each Sprint, he or she could hardly argue against the potential cost of rewriting a legacy system from scratch.

ABOUT THE AUTHORS

Kane Mar

Kane Mar is an Agile coach and certified Scrum Trainer, specializing in Scrum and Extreme Programming. Prior to his work in Agile software development, Mar spent 15 years as a developer and project manager for waterfall and RUP projects working with Java, Smalltalk, C, SQL, and PL/SQL.

Michael James

Michael James is a software process mentor and Certified Scrum Trainer, focusing on the engineering practices that enable Agile project management. Having worked in the software industry for more than 20 years as a software developer (formerly "architect"), he has experience in automated testing that predates the Extreme Programming movement; formal, phased, high-ceremony processes based on DOD-STD-2167A; chaotic non-processes of the dot-com era; and Agile processes including Scrum and XP.

ABOUT COLLABNET

CollabNet is the leader in application lifecycle management (ALM) platforms for distributed software

development teams. CollabNet TeamForge is the industry's most open ALM platform, supporting every environment, methodology, and technology. With an integrated suite of easy-to-use tools that share a centralized repository, it is the only ALM platform that enables a culture of collaboration, improving productivity 10-50% and reducing the cost of software development by up to 80%. As the founder of the open source Subversion project, the best version control and software configuration management (SCM) solution for distributed teams, collaborative development is in CollabNet's DNA Millions of users at more than 800 organizations, including Applied Biosystems, Capgemini, Deutsche Bank, Oracle, Reuters, and the U.S. Department of Defense, have transformed the way they develop software with CollabNet For more information, visit www.collab.net.

REFERENCES

1. Cunningham, Ward. "The WyCash Portfolio Management System." http:c2.com/doc/oopsla92.html.
2. Ibid.
3. See http://www.c2.com/cgi/wiki?DesignDebt
4. See http://www.c2.com/cgi/wiki?TechnicalDebt
5. Yoder and Foote. "Big Ball of Mud." http://www.joeyoder.com/papers/patterns/BBOM/mud.html
6. See http://caddigest.com/subjects/pro_engineer/select/011602_cadreport.htm
7. The act of writing documentation can be a way to learn how a system or piece of software is supposed to work. Better yet, writing tests can yield even more information.