BEST PRACTICES WHITE PAPER

Managing Technical Debt

Steve McConnell, Chief Software Engineer, Construx Software

Version 1, June 2008

"Technical Debt" refers to delayed technical work that is incurred when technical short cuts are taken, usually in pursuit of calendar-driven software schedules. Just like financial debt, some technical debts can serve valuable business purposes. Other technical debts are simply counterproductive. The ability to take on debt safely, track their debt, manage their debt, and pay down their debt varies among different organizations. Explicit decision making before taking on debt and more explicit tracking of debt are advised.

Contents

Introduction

The term "technical debt" was coined by Ward Cunningham to describe the obligation that a software organization incurs when it chooses a design or construction approach that's expedient in the short term but that increases complexity and is more costly in the long term.

What is Technical Debt? Two Basic Kinds

The first kind of technical debt is the kind that is incurred unintentionally. For example, a design approach just turns out to be error-prone or a junior programmer just writes bad code. This technical debt is the non-strategic result of doing a poor job. In some cases, this kind of debt can be incurred unknowingly, for example, a company might acquire another company that has accumulated significant technical debt that it doesn't identify until after the acquisition. Sometimes, ironically, this debt can be created when a team stumbles in its efforts to rewrite a debt-laden platform and inadvertently creates more debt. We'll call this general category of debt Type I, "Unintentional Debt."

The second kind of technical debt is the kind that is incurred intentionally. This commonly occurs when an organization makes a conscious decision to optimize for the present rather than for the future. "If we don't get this release done on time, there won't be a next release" is a common refrain—and often a compelling one. This leads to decisions like, "We don't have time to reconcile these two databases, so we'll write some glue code that keeps them synchronized for now and reconcile them after we ship." Or "We have some code written by a contractor that doesn't follow our coding standards; we'll clean that up later." Or "We didn't have time to write all the unit tests for the code we wrote the last 2 months of the project. We'll right those tests after the release." (We'll call this Type II, "Intentional Debt.")

The rest of this white paper will focus on the kind of technical debt that's incurred for strategic reasons (Type II, "Intentional Debt").

Short-Term vs. Long-Term Debt

With real debt, a company will maintain both short-term and long-term debt. A company uses short-term debt to cover things like gaps between its receivables (payments from customers) and expenses (payroll). A company takes on short term debt when it has the money; it just doesn't have it now. Short-term debt is expected to be paid off frequently. The technical equivalent seems straightforward. Short-term debt is the debt that's taken on tactically and reactively, usually as a late-stage measure to get a specific release out the door. (We'll call this Type II.A, "Short-Term Debt.")

Long term debt is the debt a company takes on strategically and proactively—investing in new capital equipment, like a new factory, or a new corporate campus. Again, the

technical equivalent seems straightforward: "We don't think we're going to need to support a second platform for at least five years, so this release can be built on the assumption that we're supporting only one platform." (We'll call this Type II.B, "Long-Term Debt.")

The implication is that short-term debt should be paid off quickly, perhaps as the first part of the next release cycle, whereas long-term debt can be carried for a few years or longer.

Incurring Technical Debt

When technical debt is incurred for strategic reasons, the fundamental reason is always that the cost of development work today is seen as more expensive than the cost will be in the future. This can be true for any of several reasons.

Time to Market. When time to market is critical, incurring an extra $1 in development might equate to a loss of $10 in revenue. Even if the development cost for the same work rises to $5 later, incurring the $1 debt now is a good business decision.

Preservation of Startup Capital. A startup company has a fixed amount of seed money, and every dollar counts. If it can delay an expense for a year or two it can pay for that expense out of a greater amount of money later rather than out of precious startup funds now.

Delaying Development Expense. When a system is retired, all of the system's technical debt is retired with it. Once a system has been taken out of production, there's no difference between a "clean and correct" solution and a "quick and dirty" solution. Unlike financial debt, when a system is retired all its technical debt is retired with it. Consequently near the end of a system's service life it becomes increasingly difficult to cost-justify investing in anything other than what's most expedient.

Be Sure You Are Incurring The Right Kind of Technical Debt

Some debt is taken on in large chunks: "We don't have time to implement this the right way; just hack it in and we'll fix it after we ship." Conceptually this is like buying a car—it's a large debt that can be tracked and managed. (We'll call this Type II.A.1, "Focused Short-Term Debt.")

Other debt accumulates from taking hundreds or thousands of small shortcuts—generic variable names, sparse comments, creating one class in a case where you should create two, not following coding conventions, and so on. This kind of debt is like credit card debt. It's easy to incur unintentionally, it adds up faster than companies think, and it's harder to track and manage after it has been incurred. (We'll call this Type II.A.2, "Unfocused Short-Term Debt.")

Both of these kinds of debt are commonly incurred in response to the directive to "Get it out the door as quickly as possible." However, the second kind (II.A.2, Unfocused

Short-Term Debt) doesn't pay off even in the short term of an initial development cycle and should be avoided.

Debt Service

One of the important implications of technical debt is that it must be serviced, i.e., once a project incurs a debt there will be interest charges.

If the debt grows large enough, eventually the company will spend more on servicing its debt than it invests in increasing the value of its other assets. A common example is a legacy code base in which so much work goes into keeping a production system running (i.e., "servicing the debt") that there is little time left over to add new capabilities to the system. With financial debt, analysts talk about the "debt ratio," which is equal to total debt divided by total assets. Higher debt ratios are seen as more risky, which seems true for technical debt, too.

Summary of Kinds of Debt

Non Debt

Feature backlog, deferred features, cut features, and so on. Not all incomplete work is debt. These aren't debt, because they don't require interest payments.

Debt

I. Unintentional Debt. Debt incurred unintentionally due to low quality work

II. Intentional Debt. Debt incurred intentionally

II.A. Short-Term Debt. Short-term debt, usually incurred reactively, for tactical reasons

II.A.1. Focused Short-Term Debt. Individually identifiable shortcuts (like a car loan)

II.A.2. Unfocused Short-Term Debt. Numerous tiny shortcuts (like credit card debt)

II.B. Long-Term Debt. Long-term debt, usually incurred proactively, for strategic reasons

Attitudes Toward Technical Debt

Like financial debt, different companies have different philosophies about the usefulness of debt. Some companies want to avoid taking on any debt at all; others see debt as a useful tool and just want to know how to use debt wisely.

Business staff generally seems to have a higher tolerance for technical debt than technical staff does. Business executives tend to want to understand the tradeoffs involved, whereas some technical staff seem to believe that the only correct amount of technical debt is zero.

The reason most often cited by technical staff for avoiding debt altogether is the challenge of communicating the existence of technical debt to business staff and the challenge of helping business staff remember the implications of the technical debt that has previously been incurred. Everyone agrees that it's a good idea to incur debt late in a release cycle, but business staff can sometimes resist accounting for the time needed to pay off the debt on the next release cycle. The main issue seems to be that, unlike financial debt, technical debt is much less visible, and so people have an easier time ignoring it.

How do You Make an Organization's Debt Load More Transparent?

One problem with technical debt is that project teams incur it intentionally, but the accumulation isn't tracked in a visible way. This is similar to the way an individual might charge numerous items on a vacation and then be surprised at the total bill when the credit card statement arrives at the end of the month.

Here are two approaches that can add transparency to technical debt tracking:

- **Maintain a debt list within the defect tracking system.** Each time a debt is incurred, the tasks needed to pay off that debt are entered into the system along with an estimated effort and schedule. The debt backlog is then tracked, and any unresolved debt more than 90 days old is treated as critical.
- **Maintain the debt list as part of a Scrum product backlog.** Each debt is treated as a Scrum "story," and the estimated effort and schedule to pay off each debt is estimated the same way other stories are estimated in Scrum.

Either of these approaches can be used to increase transparency of the debt load and of the debt service work that needs to occur within or across release cycles.

Each of these approaches also provides a useful safeguard against accumulating the "credit card debt" of a mountain of tiny shortcuts mentioned earlier. You can simply tell the team, "If the shortcut you are considering taking is too minor to add to the debt-service defect list/product backlog, then it's too minor to make a difference; don't take that shortcut. We only want to take shortcuts that we can track and repair later."

Ability to Take on Debt Safely Varies

Different teams will have different technical debt credit ratings. The credit rating reflects a team's ability to pay off technical debt after it has been incurred.

A key factor in ability to pay off technical debt is the level of debt a team takes on unintentionally, i.e., how much of its debt is Type I? The less debt a team creates for itself through unintentional low-quality work, the more debt a team can safely absorb for strategic reasons. This is true regardless of whether we're talking about taking on Type I vs. Type II debt or whether we're talking about taking on Type II.A.1 vs. Type II.A.2 debt.

One company tracks debt vs. team velocity. Once a team's velocity begins to drop as a result of servicing its technical debt, the team focuses on reducing its debt until its velocity recovers. Another approach is to track rework, and use that as a measure of how much debt a team is accumulating.

Retiring Debt

"Working off debt" can be motivational and good for team morale. A good approach when short-term debt has been incurred is to take the first development iteration after a release and devote that to paying off short-term technical debt.

The ability to pay off debt depends at least in part on the kind of software the team is working on. If a team incurs short-term debt on a web application, a new release can easily be rolled up after the team backfills its debt-reduction work. If a team incurs short-term debt in avionics firmware— the pay off of which requires replacing a box on an airplane— that team should have a higher bar for taking on any short-term debt. This is like a minimum payment—if your minimum payment is 3% of your balance, that's no problem. If the minimum payment is $1000 regardless of your balance, you'd think hard about taking on any debt at all.

Communicating about Technical Debt

The technical debt vocabulary provides a way to communicate with non-technical staff in an area that has traditionally suffered from a lack of transparency. Shifting the dialog from a technical vocabulary to a financial vocabulary provides a clearer, more understandable framework for these discussions. Although the technical debt terminology is not currently in widespread use, we've found that it resonates immediately with executives and non-technical stakeholders. It also makes sense to technical staff who are often all-too-aware of the debt load their organization is carrying.

Here are some suggestions for communicating about debt with non-technical stakeholders:

- **Use an organization's maintenance budget as a rough proxy for it's technical debt service load.** However you will need to differentiate between maintenance that keeps a production system running vs. maintenance that extends the capabilities of a production system. Only the first category counts as technical debt.

- **Discuss debt in terms of money rather than in terms of features.** For example, "40% of our current R&D budget is going into supporting previous releases" or "We're currently spending $2.3 million per year servicing our technical debt."
- **Be sure you're taking on the right kind of debt.** Not all debts are equal. Some debts are the result of good business decisions; others are the result of sloppy technical practices or bad communication about what debt the business intends to take on. The only kinds that are really healthy are Types II.A.1 and II.B.
- **Treat the discussion about debt as an ongoing dialog rather than a single discussion.** You might need several discussions before the nuances of the metaphor fully sink in.

Reasons to Pay Down Technical Debt

The more debt a project accumulates, the more a project's rate of progress (i.e., velocity) will slow down. Like business debt, there isn't really an expectation that a project ever pays it off completely. The goal is to keep the debt service at a reasonable level compared to a company's other expenses and investments.

Here are several reasons to pay down technical debt:

- "We'll be able to get future releases out in 5 months instead of 6 [or whatever] if we can spend X-amount of time paying down our technical debt."
- "We can add functionality in area X, which currently we can't do, if we spend Y weeks working on the technical infrastructure. Once we build the infrastructure for X, it will also allow us to A, B, and C fairly easily, which we can't do now without that infrastructure."
- "Our response time on hot fixes is really slow because there are major sections of the code that have become so complex that no one knows how to make changes in those areas safely. We end up with long review, test, and debug cycles even for simple changes, which requires a lot of time to release even a simple hot fix. If we spend X time paying down the debt, we'll be able to reduce our average hot fix time by Y."
- "Some of the 'shortcuts' we've been taking are starting to become visible to the customer, and the number of customer-reported defects has been increasing. Until we pay off some of the debt, we're going to have an increasingly difficult time assuring the quality of our system prior to release."

These are just a few examples of the kinds of indirect costs a team incurs when it has a high technical debt load. If a team's level of technical debt is such that it can't promise specific business benefits along these lines from paying off the debt, then it's always possible that the debt isn't worth paying off.

Focused Debt-Reduction Projects

Construx hasn't ever seen a case where diverting a large staff for a period of months to focus on debt reduction makes sense. We've seen companies attempt such initiatives many times, and they nearly always end up being unfocused boondoggles that don't produce enough business value to justify their time or cost.

Instead, we recommend breaking the debt reduction payments into much smaller pieces and then including some percentage of debt reduction work into the team's normal work flow. Software teams will often say that "It will be more efficient to do the debt reduction all at once," but it really isn't because doing it all at once removes the "business value grounding" from the work.

Technical Debt Decision Making

When a team gets to a point where it is debating taking on technical debt, people normally consider two possible paths, one of which is the "good but expensive" path and the other of which is the "quick and dirty" path. When teams reach that decision point, they often estimate the good path and the quick path. Those estimates will help inform which path the team should choose at that moment. But there are three more issues that should considered.

The first additional issue to be considered is, "How much it will cost to backfill the good path after you've already gone down the quick path?" Backfilling the good path will typically be more expensive than just following the good path in the first place because the work will include ripping out the quick code, making sure you didn't introduce any errors while doing that, then adding the good code and going through the normal test & QA processes. The "ripping out" part makes it cost more to implement the good path later than it would have cost to implement it in the first place. And of course you've already incurred the cost of the quick path, so the real cost is the sum of the quick path + the good path + the cost to rip out the quick path.

If the code is really well designed the "ripping out" cost can be minimal, but we've found that to be the exception.

The second additional issue that should be considered is the interest payment on the technical debt. I.e., if you choose the quick path now, how much does that slow down other work until you're able to retrofit the good path? The size of the "interest payment" depends very much on the specific case. Sometimes the "interest" is really just the cost of ripping out the quick code and of implementing the good code, which isn't really interest, per se. It's more like a late payment fee. Other times the quick and dirty approach does create ongoing interest payments by making related work in that same area take longer.

This leads us to the third issue that should be considered: Is there a path that is quicker than the good path and that won't affect the rest of the system? In other words, is there

a quick path that can be isolated from the rest of the system in such a way that it doesn't create any ongoing interest payment/make other work more difficult? Teams often turn the technical debt decision into a simplistic "two option" decision—good path vs. quick and dirty path. Pushing through to a third option is important because often the best path is the one that is fairly quick, albeit not as quick as the pure quick and dirty path, and whose adverse affects are better contained than those of the pure quick and dirty path.

General Example of Technical Debt Decision Making

With those three options, the decision table for deciding which kind of technical debt to take on could look something like this (assuming for purposes of illustration a labor cost of $600/staff day):

Option 1: Good Path

Immediate cost of Good Solution: 10 staff days
Deferred cost to retrofit Good Solution: 0 staff days

Option 1 cost now:	$6,000
Option 1 cost later:	$0
Option 1 lifetime cost:	$6,000

Option 2: Pure Quick & Dirty Path

Immediate cost of Quick & Dirty solution with possible interest payment: 2 staff days
Deferred cost to retrofit Good Solution: 12 staff days
Estimated cost of "interest payments": 0.5 staff days/month

Option 2 cost now:	$1,200
Option 2 ongoing cost (interest):	$600-$1,800 (assuming good solution is implemented within 6 months)
Option 2 cost later:	$7,200
Option 2 lifetime cost:	$9,000-$10,200

Option 3: Quick but not Dirty path

Immediate cost of Quick & Dirty solution with no interest payment: 3 staff days
Deferred cost to retrofit Good Solution: 12 staff days

Option 3 cost now:	$1,800
Option 3 ongoing cost (interest):	$0
Option 3 cost later:	$7,200
Option 3 lifetime cost:	$9,000

In this example, either Option 2 or Option 3 is an attractive short-term alternative to Option 1. That is, either $1200 or $1800 is a fraction of the cost/effort of $6000. But if you select Option 2 you saddle yourself with an obligation to revise the code later—either you reimplement the good solution, which costs more, or you keep paying interest, which costs more. When you select Option 3 you introduce the possibility of choosing never to pay off the technical debt, because there isn't any ongoing penalty, and so there isn't any urgency to pay off the debt.

Bottom line: When facing the prospect of taking on technical debt, be sure to generate more than two design options. Don't oversimplify technical debt decision making to just the two extremes.

Specific Example of Technical Debt Decision Making

Here's a more specific example that illustrates the same idea as the preceding general example.

Suppose that you're developing an application that requires, among other things, 5 reports. Your plan is to design a set of custom report writer classes that are specific to the nature of the application you're working on and that will make generation of additional reports later on easier. For the sake of expediency, you're being pressured to write the reports using your database's built-in report generator instead. That makes writing the initial set of reports easier (mostly), but once that initial set of reports are done later reports will be more difficult.

Here's how the options listed earlier might play out in this example:

Option 1: Good Path

Write all the custom classes, test them, and implement the 5 reports using those classes.

Immediate cost of Good Solution: 10 staff days

Deferred cost to retrofit Good Solution: 0 staff days

Option 1 cost now:	$6,000
Option 1 cost later:	$0
Option 1 lifetime cost:	$6,000

Option 2: Pure Quick & Dirty Path

Use your database's built-in report-writer. This makes generation of 4 of the 5 reports very fast. The fifth report is harder than it otherwise would have been. Going forward, you expect that each report after the first 5 will be harder to write than it otherwise would have been. Moreover, if you ever do implement the originally planned custom

report writer, you'll have to go back and regenerate each report in your custom report writer.

Immediate cost of Quick & Dirty solution with possible interest payment: 2 staff days

Deferred cost to retrofit Good Solution: 12 staff days

Estimated cost of "interest payments": 0.5 staff days per report beyond the first 5 (because later reports written using the database's report writer will be harder to write than they otherwise would have been)

Option 2 cost now:	$1,200
Option 2 ongoing cost (interest):	$300/additional report
Option 2 cost later:	$7,200
Option 2 lifetime cost:	9,000 + $300/additional report

Option 3: Quick but not Dirty path

Use your database's built-in report writer, BUT you wrap the database code in a translation layer that presents the same interface that the planned custom report writing code would present. Writing the translation layer takes two extra days, BUT the retrofit cost will be lower because the code for each report will not need to be changed when you swap out the database's report writer and swap in your own custom code.

Immediate cost of Quick solution with no interest payment: 4 staff days

Deferred cost to retrofit Good Solution: 9 staff days (1 day less than Option 1 because the interface-creation work is done during the Quick phase)

Estimated cost of "interest payments": Zero (because later reports can be written to the custom-report interface)

Option 3 cost now:	$2,400
Option 3 ongoing cost (interest):	$0
Option 3 cost later:	$5,400
Option 3 lifetime cost:	$7,800

In the short term, Option 3 costs more than Option 2 ($2400 vs. $1200). In the long term, it costs less ($7,800 vs. $9,000+$300/report). But most significantly with Option 3 you don't incur any ongoing interest payment that forces you to go back and implement the originally-planned custom code; you can delay that decision indefinitely.

It's useful to generate several different design options, rather than just the "quickest and dirtiest" on the one hand vs. the "most pure" on the other hand. Often a hybrid approach ends up being the best option.

About Construx

This white paper was created by Construx Software Builders, Inc. Construx Software is the market leader in software development best practices training and consulting. Construx was founded in 1996 by Steve McConnell, respected author and thought leader on software development best practices. Steve's books *Code Complete*, *Rapid Development*, and other titles are some of the most accessible books on software development with more than a million copies in print in 20 languages. Steve's passion for advancing the art and science of software engineering is shared by Construx's team of seasoned consultants. Their depth of knowledge and expertise has helped hundreds of companies solve their software challenges by identifying and adopting practices that have been proven to produce high quality software—faster, and with greater predictability.

Steve McConnell, CEO/Chief Software Engineer

steve.mcconnell@construx.com
+1(425) 636-0100

Jenny Stuart, VP Consulting

jenny.stuart@construx.com
+1(425) 636-0108

Matt Peloquin, CTO

matt.peloquin@construx.com
+1(425) 636-0104

Steve Tockey, Principal Consultant

steve.tockey@construx.com
+1(425) 636-0106

Mark Nygren, COO/VP Sales

mark.nygren@construx.com
+1(425) 636-0110

For more information about Construx's support for software development best practices, please see our website at www.construx.com, contact us at consulting@construx.com, or call us at +1(866) 296-6300.

Construx Software Builders, Inc.
10900 NE 8th Street, Suite 1350
Bellevue, WA 98004
U.S.A.