How Much of a Problem is Technical Debt and What Should We Do About it?

April 2013

Scope of this Report
Delivering Software systems and services has always been a balancing act of sorts, between delivering on time and delivering “best practices”. Every so often compromises are made on each end and whether intentional or unintentional, technical debt is created in the process. If left unmanaged, technical debt can hinder, debilitate, and even render an entire organization obsolete. This report identifies some root causes of technical debt then outlines what can be done to identify and manage it.

What is technical debt and how can it be identified?

First, we must clearly define technical debt. Several definitions exist, but we will be using the one espoused by Ward Cunningham, who first coined the metaphor (Fowler, 2004). The metaphor is meant to help non-technical stakeholders understand the long term implications between choosing, within a single iteration, to code things in a ‘quick and easy’ way to get it ‘out the door’ versus coding things more deliberately and in a way that will be easily and readily adaptable to integrate seamlessly with future iterations and changes. The quick and easy way gets things done faster for this iteration, but in the next iteration that software components lack of ‘future proofing’ can cause some overhead and require extra time from the developers once it is time to make changes or add functionality to the system. Eventually, this overhead can build up to a point where it forces projects past their completion dates and cost projections. The debt can be so cumbersome that it puts a system into a state where it would be less expensive and time intensive to completely rebuild the system.

Douglas Smith and Ipek Ozkaya, both of the SEI, assert that technical debt has financial implications, just like monetary debt payments. Developers can choose to pay interest on their technical debt in the form of additional time and effort required to understand and modify poor structured code. Conversely, developers can pay down the debt by refactoring poorly designed code to reduce future effort. Ozkaya suggested that understanding the financial model implied by the “debt” metaphor can help establish the structural aspect of debt. These financial implications suggest the following questions that development teams must consider:

- What is the “interest rate” that organization signs up for when incurring technical debt?
- Can this interest rate be controlled?
- What is the period of the loan?
- What is it we’re borrowing? Time? Or, other opportunities we need to bring to bear when managing timeline of loan?
- How do we create a realistic repayment strategy?

For a breakdown of different types of Technical Debt, an excellent source is the “taxonomy of technical debt” created by Steve McConnell. McConnell discuss the difference between so-called “good” and “bad” debt but Martin Fowler summarizes it best in the following table:
The above table shows that there are times when it might be prudent to deliberately take on technical debt (such as the need to be first to market) and times when it is reckless to do so deliberately (thinking that time for long term design considerations is not worth spending now). A real life example given by John Kern involves winning a million dollar contract. A client has requested a new feature and if they get that feature, they will purchase other products from the same vendor for a price tag of one million dollars. The vendor is able to ‘hack’ together a solution that will make the client happy right now and earns the contract but builds up an amount of technical debt, since it is not properly tested or documented. The vendor recognizes this debt and makes plans during the next iteration or cycle to ‘pay down the principle’ and eliminate the debt. In this case, the technical debt has earned the vendor a million dollar contract and will not have long reaching consequences as the accumulated debt will be handled immediately. If the solution where hacked together with no plans to pay down the principle, it would be considered bad debt because the interest payments would be hanging over the vendor every time that module is touched.

Finally, is Technical Debt the same as coding defects or bugs? We are inclined to think not although there may be an overlap for lower severity bugs. The insidious nature of technical debt is derived from the fact that the code usually works, it just gets harder and harder to maintain or change.

How much of a problem is Technical Debt?

Most developers and some Application Development and Maintenance (ADM) managers are aware of the moral hazard of Technical Debt but few, including the DCG consulting team, can cite concrete examples of disasters due to technical debt. Instead, we find ourselves recalling examples where technical debt was one factor amongst several contributing to significant, and usually costly, decisions, to retire or re-platform applications. The economic cost in such examples is very dependent on the situation. For example, in a simple decision to re-platform and application due to technical debt and, say, diminishing programming expertise in the old language we can imply that the cost of the technical debt is broadly equivalent to the cost of the project to re-platform. However, if the decision to retire an application is due to its relative technical debt compared to another with similar functionality in a merger/acquisition scenario then arguably the economic cost of the technical debt at that point is only what the acquiring company paid for the asset (which may be zero).
So perhaps the true answer to ‘how much of a problem is technical debt’ may well depend upon who you ask and when.

If RIM (makers of Blackberry) was asked 10-15 years ago, they might have said it is not a problem for their organization. If they were asked today, they might say that losing their majority share of the mobile OS market is the problem with technical debt. They had such an enormous technical debt built up that by the time they released a tablet it didn’t have an e-mail application – a feature which is widely considered to be a primary function of most useful mobile computing devices. This was due in large part to their outdated software architecture and unwillingness to take care of their principle balance of technical debt before the interest payments of technical debt got out of control, costing the company enormous sums of revenue and consumer confidence which still have yet to return.

The accumulation of additional debt is generally due to a combination of any of the following causes, (taken from Wikipedia) and the degree to which these causes are present in a given organization is a reasonable indicator of the likelihood of technical debt in the current code base:

- **Business pressures** where the business considers getting something released sooner before all of the necessary changes are complete, builds up technical debt comprising those uncompleted changes
- **Lack of process or understanding** where businesses are blind to the concept of technical debt, and make decisions without considering the implications
- **Lack of building loosely coupled components** where functions are hard-coded; when business needs change, the software is inflexible.
- **Lack of test suite** which encourages quick and less risky Band-Aid’s to fix bugs.
- **Lack of documentation** where code is created without necessary supporting documentation. That work to create the supporting documentation represents a debt that must be paid.
- **Lack of collaboration** where knowledge isn’t shared around the organization and business efficiency suffers.
- **Parallel Development** at the same time on two or more branches can cause the buildup of technical debt because of the work that will eventually be required to merge the changes into a single source base. The more changes that are done in isolation, the more debt that is piled up.
- **Delayed Refactoring** As the requirements for a project evolve, it may become clear that parts of the code have gotten unwieldy and must be refactored in order to support future requirements. The longer that refactoring is delayed, and the more code is written to use the current form, the more debt that piles up that must be paid at the time the refactoring is finally done.

**What should we do about Technical Debt?**

So what should we do about technical debt? So far we have learned that unmanaged technical debt is an enormous liability on an organization and that the difference between good technical debt and bad technical debt is how it is managed. Given these facts, the high level solution for handling technical debt becomes clear: management of the debt. An organization must take ownership of the debt and the best way to accomplish this is via management strategies which share strong similarities to both risk management and financial portfolio management.

The key to taking ownership of technical debt is organizational awareness. This is achieved by ensuring that anyone, from the coders to project managers, is able to identify technical debt and understand how it impacts projects. Three general strategies are needed to accomplish this task:
1. Create a system for tracking and cataloging issues.
2. Analyze the cost/benefit (including future man hours) when considering taking on additional debt.
3. Use your newly created documentation to communicate the ramifications of technical debt to the stakeholders. If the stakeholders are aware of the cost/benefit analysis, it will be easier to encourage an internal process for project planning which earmarks time during each development cycle specifically for technical debt reduction/mitigation.

What Technical Debt Management Techniques are available?

Backlog/Baseline Methods

The backlog/baseline methods are built around the idea that you must have some data about the debt in order to manage it. A developer backlog can be built using tags within the code (Malmsten, 2013). When this tagged data is combined with the estimated amount of work days needed to eliminate each of the individually tagged debts, it can then be used to help guide decision making about the acceptable amounts of principle/interest present in a given system. There are also open source baseline tools available to compare against industry standards, such as www.techdebt.org which allows benchmarking of many languages.

To make backlogging useful and move to the next steps, buy-in must be achieved. Management must understand why time that is not spent pushing a short term project out as fast as possible is beneficial in the long term. Developers must understand why their aim should always be to leave the code better than they found it and apply this knowledge. One way to help convince both segments is to show them why it is beneficial to their own segments. Generally speaking, “it all boils down to data: track your velocity or your cumulative flow over time to be able to show concrete data that it will take you longer and longer to push out features if the coding issues are not dealt with.” (Szynkarski, 2013) If you can show this to management, they will listen. Developers are usually easier to convince, as they will be the ones who have to work off the debt later.

Portfolio Prioritization

These strategies can be accomplished by establishing a portfolio prioritization methodology. Prioritizing the portfolio requires combining four strategies: evolving a mature definition of done, preventing unacceptable technical debt, prioritizing & quantifying existing debt, and aligning debt reduction initiatives with both short and long term business goals (Garnett, 2013).

If the definition of “done” in terms of a software deliverable is lacking, there will be a larger gap between what is expected and what is delivered. This knowledge gap needs to be minimized in order to reduce the debt that will be incurred as a part of the project.

Prioritizing and quantifying the portfolio requires utilizing the baseline and backlog data created for every application within a comprehensive organizational portfolio to help analyze and demonstrate the debt in a way that any party can analyze and use to find an efficient way of fulfilling the business objectives of the software - in both the short and long terms - through well informed technical debt management (Garnett, 2013). This generates buy-in from all parties as it reduces costs and simplifies work.
Classifying the portfolio applications with their business objectives in mind (Source: Garnett 2013):

Software Code Inspection and Analysis

Measuring and understanding a problem goes a long way towards fixing it. To this end, we recommend the use of code analysis techniques to identifying negative patterns in the code that could be technical debt or something more serious such as security weaknesses. The best code analysis is a blend of clever analytical tools and experienced minds to interpret the results and recommend solutions.

Conclusion

Technical debt is unavoidable but manageable if you acknowledge its existence and put the right steps in place to deal with it. The cost of ignoring technical debt is hard to quantify. Many organizations get away with it forever but for some it can be a costly risk which can occur at the worst possible time such as when someone launches an iPhone™!

Sources


