cm FIRST
Rethink Modernization

// WHITEPAPER

Technical Debt

What Is It Costing Your Company?

Technical debt may be a new concept but it is an old problem that increases business risk and cost to your company.

Technical debt is the cost of programming choices and decisions that were made consciously to meet a business objective, unconsciously because of lack of knowledge or experience, or historically because they made sense initially but are not best practices today. All development will result in some amount of technical debt—the challenge is to manage it, reduce it and develop practices to keep it at a level that does not impact performance and availability of your critical business services. Software solutions now exist to help make paying down your debt a much less onerous process.

The Problem

"The savvy developer treats technical debt just as the entrepreneur does financial debt. They use it. It speeds delivery."

In 1992, Ward Cunningham, inventor of "wikis" and a signatory to the Agile Manifesto Doctrine, coined the term "technical debt" to describe the application design trade-offs organizations make every day. The metaphor is apt; executives often make conscious decisions to deliver new business functionality as quickly as possible. As with financial debt, they weigh the benefits of faster time-to-market and increased profits against the probability of sub-optimal code. In some cases, speed is required simply to meet a government mandate. Whatever the cause, whether it is in service of maintaining a competitive edge or meeting compliance requirements, technical debt is incurred whenever corners have to be cut in design, coding and testing. Like financial debt, interest accrues. In this case, 'interest' means impact on availability as well as higher support and application maintenance costs. Another term for technical debt is 'deficit programming,' a term coined by David Panariti.

Some of these choices were made in the early days of computing where it seemed safe to assume that applications written today would be supplanted with new and better code tomorrow. Surviving the Y2K challenge involved a substantial interest payment to remediate applications where shortcuts were applied. These shortcuts saved substantial money, resources and time when all these

Contents

elements were in short supply. No one expected the code containing all these shortcuts to last 20+ years, but it did. Now, we know better. Code lasts a long time.

Another cost-saving technique initiated in the ‘90’s also increased the technical debt. Assumptions were made that coding languages were now so powerful and easy to learn that junior programmers could be trusted to create business-critical applications from detailed specifications. But even senior programmers have been known to have some bad coding habits. Facing time pressure and learning ‘on the job,’ junior programmers were even more likely to make mistakes, take shortcuts and sometimes copy old code from other applications to get the job done. Another area of concern was with outsourcing. Outsourcing development added to the problem. With programmers in other countries, it becomes difficult to vet the code or the ability of the coders. Inside a company, technical management can prescribe standards and institute oversight; with an outsourcer, both objectives may be difficult to achieve.

The 21st century debt includes quick turnaround to link applications on disparate platforms. The value can be substantial, but the interest continues to accumulate. In addition, technical debt compounds over time. As applications are modified by a variety of people, each making assumptions about the underlying code, more problems are inevitably introduced. Unless a piece of code is very short-lived, very basic or completely documented, technical debt is a concern you have to monitor and manage.

How big is the technical debt problem? Software engineering expert, Capers Jones notes, “Poor software quality has become one of the most expensive topics in human history: $150 billion per year in U.S.; $500 billion per year worldwide.” He also noted that programmers were “less than 50% efficient in finding bugs in their own software.” Looking at just a single industry, “Gartner estimates that the global insurance industry has accumulated more than $8 billion of IT debt during the past 10 years.”

Types of Technical Debt

Technical debt comes in a variety of forms, but without understanding the sources, remediation is impossible. The most fundamental type comes when an application is initially designed. Often, the full scope of a business service is poorly understood at design time, so while the design might be ideal in the initial implementation, it does not adapt to the many changes required as the application matures. In other cases, poor design can be the result of a misunderstanding between the architects, the business and the development team. Still another cause could be compromises made as an agile project evolves.

Code debt groups a number of problems into one bucket. The most obvious is badly written code. This happens primarily because of coding inexperience. The next, paradoxically, arises from very senior programmers. Overly complex code may work very efficiently, but when it is complex, updates to it without a clear understanding of the complexity may result in problems. In the case where modules were written much earlier, good code is no long optimal because new language facilities and options may

have superseded the original capabilities. And some languages no longer have enough practitioners who understand them—the language itself may have become obsolete.

A common problem is that people under time pressure copy what's working and move it somewhere else. When an issues arises or the code otherwise needs modification, how do you find all the places it has been copied? Capers Jones notes that 21% of COBOL code is unreachable, meaning that it is never executed. But in many cases, developers are not aware of this and changes made may cause untested code to become reachable, or unreachable code to be copied.

Two unexpected areas of technical debt come from lack of documentation and lack of testing protocols. Few developers enjoy writing documentation, so in many cases and for many applications, documentation is scanty or non-existent. As new teams come in, they have to start from scratch figuring out the purpose of various modules and the underlying design. Testing is an issue—code that is not regularly tested may be a problem just waiting to happen.

Code fraught with these kinds of problems becomes brittle. Like taffy as it hardens, making changes to older code can be difficult without the risk that it will break.

Compounding Factors

Organizational structure may tend to help to increase or limit the potential for accruing technical debt. In many shops, the development team has no interaction with the lines of business. Deadlines are determined by the business without development involvement or input. In many cases, the business has no idea of whether their requirements are feasible or achievable. Development personnel also may not hear directly from the customer what his needs are. Yet, development groups will strive to meet those deadlines making their best efforts to deliver the business services required. In too many places, groups with a stake in the overall success of the application (performance analysts, DBAs, storage support, network technicians) only see the finished product and thus, cannot offer their expertise to ensure an optimal solution. Outsourcing only makes this more complicated. Particularly when outsourced staff is on the other side of the globe, the required communications most likely will not occur practically ensuring a lower quality result.

The lack of a process improvement program can also lead to increased technical debt. With fewer staff members and increased pressure to get code out the door, little time is left to go back and review code beyond fixing outright bugs and trying to eliminate outages. Even those companies dedicated to quality may find they don't know how to identify their code debt, let alone find the time to remedy it.

The Cost of Technical Debt

It can be easy to dismiss this problem by saying "If it isn't broken, why fix it?" The problem with technical debt is that your suite of business applications is like a house. A superficial glance might give the

impression of a stable structure, but if the foundation has developed cracks, the roof is about to leak or the wiring is brittle and about to fail, it is broken, even if you can't tell. Even if your code is only a few years old, you may have started to accrue debt, putting stress on your business service 'infrastructure.'

A quick view of the impact can be seen in Figure 1. Not only does quality continue to degrade, but so does customer satisfaction. Your ability to serve new markets and new customers depends on the ease of successfully and safely changing application code.

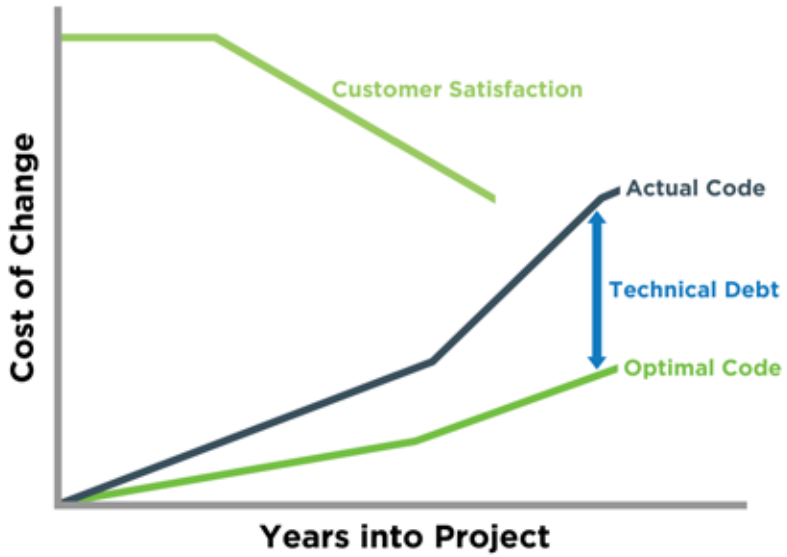

Figure 1 – The Cost of Technical Debt

For companies trying to manage risk, knowing that the foundation of your business—your code—may have serious weaknesses should cause concern. The risks and costs are many beginning with the very real possibility that the next change to a program could cause an outage. Most businesses have a good idea of how much an hour of downtime costs them in lost revenue, but with the increasing 'need for speed' in an always-on world, lost customers could multiply the impact of an outage. Allied to this is the fact that fixing a problem in complex, poorly understood code can be much more time-consuming, lengthening the duration of the outage. Many companies don't realize that their code is essentially undocumented until a major outage occurs. With constant employee turnover, both through employee choice and layoffs, new people are encountering applications that have been touched by many people at different skill levels. This means that even regular maintenance, whether to add new features, to fix issues or to apply compliance features, will be slower, more difficult and more error-prone. When attempting to interconnect applications to build a new capability, those interconnections can be problematic as well. Again, the more technical debt, the more of a black box the application has become. And what you don't know about your code can hurt you and your business.

Other costs include impacts to performance. As code becomes more complex, it often performs poorly. Often, no one has taken a look at the application holistically since early in the design phase, so inefficiencies are inevitable. Inefficient applications cost you in more ways than response time and slowdowns (which also cost you customer loyalty). Inefficiencies inevitably translate to increased system resource demand. And while the cost of computer resources across all platforms has been dramatically reduced, in many cases, upgrades in hardware lead to increased software costs.

One area not always considered in code quality is security exposure. Poorly written code can lead to exploitable vulnerabilities that may go unnoticed until a hacker discovers them. Code written by a disgruntled employee or unscrupulous outsourcer may contain 'backdoors,' inviting exploitation at a later date. Either incompetence or dishonesty can lead to another type of technical debt with potentially even more serious results. In some studies, security exposures of all types were the number one type of technical debt.

Most companies make a great effort to reduce risk and operate in a prudent fashion. But too many are unaware of the cracks in the foundation of their application "building." Technical debt has the potential to cause serious impact to your revenue and to your customer base if not addressed.

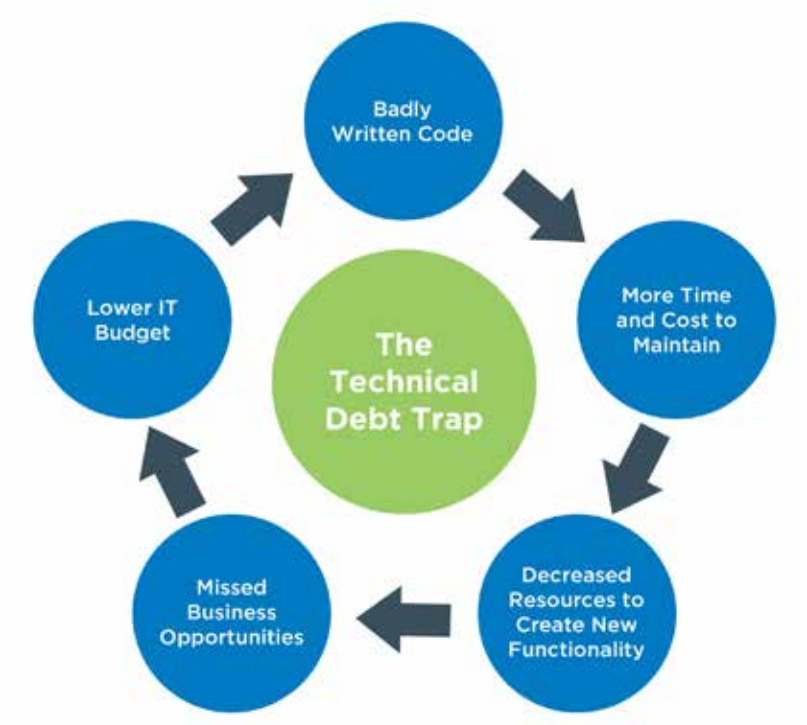

Figure 2 – The Technical Debt Circle of Doom

Figure 2 highlights the trap companies can find themselves in when they do not invest in addressing the technical debt issue.

Paying it Down

As with financial debt, the first step is to acknowledge that there is a problem. Senior management backing is essential, first to set new directions for the company in this area and next to fund the effort. For many companies, the difficult part is to find the problems in the code; fortunately, this does not have to be a painful, manual effort anymore. You don't have the time for that nor do you have the resources. In fact, it is fair to say that it is impossible without strong, automated, analysis tools. Software is available to help you identify the code that will need refactoring, automatically and quickly.

Look for a top-quality code comprehension solution that can analyze source code portfolios across all the hardware and operating systems in your data center. Make sure to include all production systems and cover not just application code, but also your JCL and other pseudo-code. The solution should provide the kind of meta-analytics that makes the next step, refactoring, a lot easier. Code maintainability and quality metrics are key to helping you determine inefficiencies, but you also want to look at data lineage, application and database connectivity, and code complexity information. In addition, if you are exploiting Big Data or various implementations of cloud, make sure your code analysis solution supports those as well. Your analysis software should be fast and flexible enough to return results quickly and affordably. Taking only periodic snapshots or analyzing only a portion of your code base limits the effectiveness of the solution.

The best software will take advantage of the processing power you have, exploiting multi-core and multi-engine processing, parallelizing the effort. Near real-time analysis will give you a true 'state of your code,' which is critical in prioritizing the effort. The software should be affordable and work across your entire code base, including legacy applications. Without this, you could risk serious problems down the road, particularly with older applications. Then, you will have an enterprise metadata repository of technical debt which will allow you to prioritize and plan your effort. Look for a solution that will provide access to the data in a clear, understandable fashion so that business people as well as technology professionals can understand it and use it to make business decisions.

Focus on debt that is either costing you a great deal in inefficiency or leading to greater risk because of complexity or outage potential. Another factor is to focus on obsolescent technology. Moving away from platforms with reduced support and personnel may prove to be a significant win. The initial list may seem daunting, but by using business priorities and weighing the importance of significantly reducing risk and data center costs, the value will rapidly become apparent. Gartner notes they typically see 10-20% of the overall development budget allocated to fixing technical debt. And yet, this might not be enough.

A good practice is to fix debt as part of the maintenance/development cycle. Before touching a program to enhance it, review the debt repository and fix any problems in the program before adding requested enhancements. This starts to reduce complexity and improves the success rate of added code. You will never get rid of all debt (just as your company will probably never be financially debt-free), but managing it and

limiting it reduces corporate risk.

Each of these actions should be part of a major project that is managed just like any other project with dates, deliverables and specific task assignments. This helps ensure that refactoring has the proper weight and importance in development work. It's easy to backslide and incur more debt when feeling pressured to enhance code or write new applications.

Ideally, this is the time to set the groundwork for development standards that ensure better work in the future. Research and make standard the latest design practices. Ensure personnel have the right training and experience to do excellent work. Focus on simple, elegant solutions and ensure that project teams have the time to make these happen. Engage performance experts with developers to ensure code is written to perform from the beginning. Match testing to development to ensure that all code written will be tested before production implementation.

As part of continuous service and process improvement, it can be helpful to establish peer review of code as well as implementing a development mentor program to ensure continued knowledge transfer. Look for ways to document code whether mandating it is the responsibility of the developer or by using software tools to accomplish this goal. And finally, realize that the small increase in time needed for a quality software project will more than pay for itself in reduced technical debt, both 'principal and interest.'

Conclusion

For too many years, development was a 'write and forget' practice. But as with any kind of debt, the longer you go without paying the interest, let alone the principal, the more expensive the note becomes. In this case, technical debt translates to greatly increased business risk as companies devolve most of their business functionality onto IT systems. Ignoring it doesn't just cost you money—it can cost you customers and your reputation. The good news is that once you acknowledge that you have the problem, the solution isn't as daunting as it once may have been. Software can help you identify, manage and ameliorate your technical debt. And once you begin to address it, you can also put into place measures to reduce the debt you incur in the future.

About CM First Offerings

CM First's powerful automation tools, augmented by professional services staff with many decades of software engineering and DevOps experience, ensure successful outcomes for even the most demanding modernization projects. Our products and expertise have helped over 400 customers in the public and private sectors reach their desired future state faster and more cost effectively than by using conventional approaches.

CM First software quickly analyzes, documents and re-platforms legacy code bases with minimal errors and rework, including those that are too large and complex for humans to tackle in any reasonable timeframe. The output is immediately usable by all team members, regardless of experience and knowledge of legacy software languages, accelerating application maintenance and modernization projects.

For more information, visit **cmfirstgroup.com**

Request a Demo Today

Contact us for more information or to schedule a demo. Call 888-866-6179 or email us: **info@cmfirstgroup.com**

cm FIRST
Rethink Modernization

CM First Group
888-866-6179
cmfirstgroup.com

7000 North Mopac Expressway
Plaza 7000, 2nd Floor
Austin, Texas 78731