SV/SG

Silicon Valley Software Group

TECHNICAL DEBT

A framework for business leaders and engineers to make joint decisions on technical debt.

TECHNICAL DEBT

Bitter battles are often waged over technical debt. Business-minded stakeholders accuse engineers of being perfectionists for wanting to fix it, while engineers accuse the business people of "not getting it". Both perspectives are valid: a company needs a steady stream of new features to grow, yet a product that sees continued traffic growth will need to be re-architected at some point. This white paper offers a framework for these two perspectives to meet, share the same language, and thus drive decisions that are supported by all.

Inspired by Dan Hackner's article Managing Technical Debt, this paper offers an in-depth analysis of technical debt, as well as strategies to reduce it in combination with new feature development. We will cover:

- The definition of technical debt and why it occurs
- The risk and cost associated with accumulating debt
- Best practices in planning and communication
- Day-to-day considerations in dealing with technical debt

We will address in-depth the various ways—some legitimate, some not—that technical debt is created. In this paper we position ourselves at a point in time, like during quarterly product roadmap review, when we need to make decisions about future releases and allocation of engineering resources (we will discuss strategies to prevent day-to-day accumulation of technical debt in another paper).

WHAT IS TECHNICAL DEBT AND HOW IS IT CREATED?

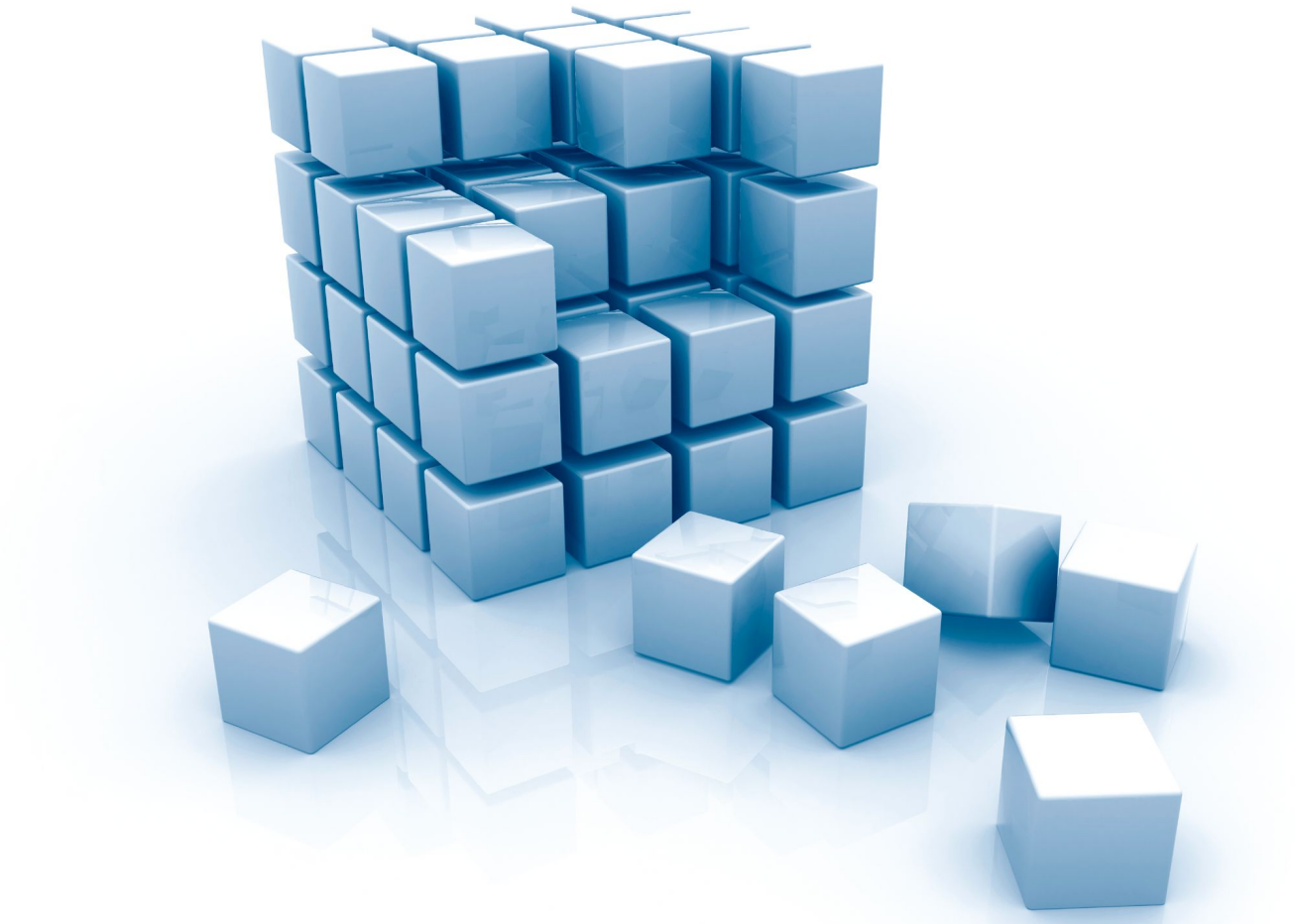
Martin Fowler, a leading voice on enterprise software, explains how technical debt can arise from making certain decisions up front:

"Doing things the quick and dirty way sets us up with a technical debt, which is similar to a financial debt. Like a financial debt, the technical debt incurs interest payments, which come in the form of the extra effort that we have to do in future development because of the quick and dirty design choice. We can choose to continue paying the interest, or we can pay down the principal by refactoring the quick and dirty design into the better design. Although it costs to pay down the principal, we gain by reduced interest payments in the future."

> *Technical Debt comes from multiple sources, not just fixing code written quick and dirty, but also from a UX paradigm that becomes dated, an architecture that can no longer support new features or performance requirements, frameworks that are no longer supported.*

"Doing things the quick and dirty way" is an important source of technical debt, but by far not the only one. Other sources of technical debt have "natural causes" that are unrelated to poor decisions made by the engineering team. It can stem from inheriting a product with legacy code, or outdated infrastructure that slows down velocity and creates painful performance problems. It can also happen when code simply ages—the user interface or data models become inadequate as more features are added, or the architecture must be restructured to meet new requirements. Similarly, when software built to run on a handful of servers for a relatively small customer base sees traffic grow by 10x, or 100x, it reaches a point where it needs to be refactored or even re-architected.

Whatever the cause, technical debt must be accounted for and reduced, in order to avoid reaching a dangerous tipping point where its weight reaches a level that affects the financial health of the business.



THE RISK AND COST OF TECHNICAL DEBT

Technical debt creates a burden that can cripple a company's velocity and growth rate if left unattended. When we take on any amount of technical debt, whether willfully taken on or inherited, we are taking on a liability that impacts us in two different ways: a burden to pay and a risk that must be managed.

The Burden of Paying Back the Debt

On one hand, if we "pay now" (i.e. if we pay the principal) any money we spend paying down technical debt cannot be spent on new features or enhancements. On the other, if we "pay later", the interest and possibly the principal continues to accumulate. Left unchecked, the accumulation of technical debt can eventually render us "bankrupt"—at a point where we can no longer respond to unplanned events quickly enough to stay competitive.

> ### Example: When Old Technology Gets Pushed Out En Masse
>
> In September of 2015, Chrome started blocking Flash ads from auto-playing on websites. This came on the back of first Safari's, then Mozilla's, then Amazon's decision to do the same. This was the "writing on the wall" that browsers would sooner or later stop supporting Flash in favor of HTML5. If your website or application was written in Flash, this decision instantly created a huge amount of technical debt for the product.

This is an example where technical debt is not caused by poor decisions, lack of planning, or expediency to release a product to customers, but rather by external factors. Other examples are Apple moving to Swift as its iOS development language of choice (instead of Objective-C), or Facebook deciding to shutter the Parse development platform.

Risk Caused by Technical Debt

Debt not only creates a burden that needs to be paid back but it also creates risk. Because risk is complicated and often unintuitive to grasp, it is often overlooked. However, managing risk from technical debt is best done from a preventative standpoint—like with health care, early intervention can prevent a serious emergency.

There are two kinds of risk to monitor:

- ## Aggregative

  An insidious risk that accumulates over time but does not exhibit a notable turning point that puts the company in severe danger. However, as it builds up, it eventually prohibits the company from responding quickly to new market opportunities, competitive pressures, or security attacks.

  The best example of this is unmaintained code. It can go on for several years without being a serious problem, but finally grows to a size where implementing a new feature requires more time fixing existing code than writing new code.

- ## Time-driven

  A risk that increases significantly with the passing of time but often retains the same cost to fix, much like changing the oil in a car: the cost is the same whether you can change the oil after 5,000 or 20,000 miles. Yet, at a certain point, neglecting to fix the issue can result in catastrophic setbacks.

  A good example of time-driven risk is the result of underinvesting in datacenter monitoring tools, which often get dismissed when companies are running on a small, simple infrastructure. But as a SaaS product grows and uses more servers, diagnosing issues is more complex. There is a high risk of experiencing major performance issues and irreparably damaging the health of the company.

Making Decisions About Risk

When we think about risk, we tend to focus more on two out of three important variables: the likelihood of the negative event and the cost of a remedy.

Yet a third variable must be considered so we do not dismiss the rare events that can ruin the health of an otherwise successful company—the cost of a catastrophic negative event. We can calculate risk by "expected cost", or the probability of the negative event times its cost, and compare it against the cost of the remedy. For example, a startup that has just launched its product should not invest in a disaster recovery plan that will mitigate risk if the datacenter is down for an extended period of time. As the company reaches $100 million in revenue, the equation flips.

> Catastrophic risk is difficult to evaluate, yet it cannot be ignored.

Worse, some or all of its customers may never come back after a security exploit. I experienced it firsthand with a consumer mobile app where a security attack on one of our partners forced us to reset the passwords of all our customers. Even after 3 months, one third of our customers had never bothered to create their new password.

Risk that could lead to the death of the company is non-negotiable. It is too dangerous to let warranties on critical networking, security, and storage subsystems become outdated. While the cost of paying for an expensive disaster recovery plan or other preemptive security measures may seem high, it is not worth losing everything the company has worked for.

BEST PRACTICES IN TECHNICAL DEBT MANAGEMENT

Once we understand the risk and cost of technical debt, we can use this knowledge to outline a detailed plan for dealing with it. However, this not enough. Because developing new features and reducing technical debt draw from the same pool of engineers, we need to develop a mechanism to resolve this competition for resources.

Finding a Common Language

Reducing technical debt competes for the same engineering resources as the development of new features, enhancements, and bug fixes requested by customers. From a CEO's perspective, new features take priority because they create new revenue. On the other hand, fixing the technical debt on a feature that is already shipped is perceived as 100% cost and 0% benefit. Yet failing to address technical debt in time could create massive discontent among existing customers (if the debt causes an outage of the site, or if the page load time becomes too slow).

In order for engineers and other executives to speak the same language, and for them to be able to compare costs and benefits of new features versus reducing technical debt, the best tool is to use the return on investment. More specifically, the case for reducing technical debt must demonstrate that it either:

- Increases revenues
- Reduces costs
- Minimizes the risk of a catastrophic event

Example: Framing a Technology Decision by the Numbers

The DBA team has demonstrated that certain tables in the database have become very large, causing certain queries to become very slow. Implementation of a cache layer, or sharding, for example, will bring queries execution time and thus response time to the end user back up to standards.

However, from a customer's perspective this does not count as a new feature—it is simply expected behavior. One way to make the argument is: *our analytics show that engagement drops by 20% with each 1ms increase in response time, and that 10% of the customers exhibiting low engagement churn each month. By our calculations, we can eliminate churn of at least one hundred users each month at an LTV of $500,000.*

Engineers who discuss technical debt often present their arguments in terms of purity of code, or beauty of the new implementation. However, this is akin to speaking a different language than the business team. This is why we, as engineering leaders, need to translate the technical solution into customer benefits and return on investment—which can then be compared to those brought by prospective new features.

5 Essential Questions for Technical Debt Decision-Makers

QUESTION	PRESENTING THE CASE
Will the debt lead to loss of revenue (e.g. customer churn, low upsell)?	Quantify this revenue loss, and compare with cost to pay down the debt.
Does the debt carry a negative risk to the viability or the success of the company?	Multiply the cost of a negative event by its likelihood and compare to cost of a remedy.
Does the debt impact our velocity to develop new features?	Quantify the revenues associated with each option.
Does the debt lead to high operational costs?	Quantify them and compare to the cost of fixing it.
Does the debt affect our ability to scale?	Measure the lost revenue anticipated.

Using the Right Time Frame

For a discussion on paying down technical debt to be productive, all parties need to agree on a meaningful time frame to measure return on investment. Otherwise, the answer is guaranteed to be “NO” because, by default, the assumption is that work must take place within the next quarter. Over the time span of a quarter, technical debt work is all cost and no reward.

Instead, when over one to two years for example, the benefits of the investment accumulate and lead to a meaningful and positive ROI.

Example: An Agile Approach of Technical Debt

A development team might have a set amount of debt to repay—fixing this technical debt might cost 5 story points for 2 quarters (12 2-week Sprints) for a total of 60 story points-Sprints. However, once it is fixed, velocity will increase by 3 story points every Sprint. This results in a break-even point at 20 Sprints (within 3 quarters). Over a 2-year period, the team is net positive 48 story points (-5*12 + 3*36), i.e. 1/2 story point per Sprint. It demonstrates a strong rationale for fixing the technical debt.

Using Debt to Drive Growth

Sometimes taking on technical debt is a deliberate strategy used to accelerate growth. In this case, it is crucial to understand why we are doing it, and to create a concrete plan for paying it back.

There are several reasons to purposely take on technical debt:

Getting to Market Faster

When we face pressure to ship code rapidly, we may benefit from taking on some technical debt. Releasing an MVP in time for an important deadline, for example, might be worth a faster implementation that only handles the “happy paths”. In another instance, we might catch wind of a feature that a competitor plans to release and attempt to beat them to it.

Responding to a Feature Request from an Important Prospect

Sometimes a prospect requests a new feature as a dealbreaker. More often than not, these requests come with a tight deadline (right before the end of the quarter), and the team does just enough to meet the particular prospect’s requirements, but not enough for the feature to be shared right away with the all other customers. This technical debt needs to be identified, and ideally resolved promptly, in part because the sales team will advertise it to all their prospects as if it was “done”.

Prioritizing Different Types of Technical Debt

Some types of technical debt pose more of a threat to the health of our company than others. We present various scenarios in general order of priority (see chart below). Naturally, the ranking will vary depending on the specifics of the product and company, and the following framework presents a good starting point to analyze our technical debt.

RISK LEVEL
Very high

STRATEGY
Pay now

Out-of-Support Packages

For packages that affect the security or the overall uptime of the system, timely updates are non-negotiable. Failing to update can mean the end of your business.

For others—like frameworks and libraries included in the product—it is still dangerous to fall behind in the update cycle. Once behind, the cost of addressing compatibility issues becomes a burden that gets heavier at each release. Eventually you'll be faced with an unpalatable choice: migrate to new tools that don't support your old packages or miss out on productivity enhancements gained by adopting new tools.

The remedy here is easy and effective: upgrade all packages used in the product during the first sprint of each release. The effort is minimal compared to doing it later, and there is no cost of validation since it is included in the testing of the release.

RISK LEVEL
Very high

STRATEGY
Pay now

Absorbing a One-Off Request from an Important Prospect

One-off requests can be both rewarding and dangerous. Rewarding when they bring in a game-changing customer, but dangerous because the code is typically written in a hurry. The result often only meets a single customer's narrow requirements and therefore is not deployable to all users.

As a consequence we need to pay up this debt as quickly as possible so that the feature can be deployed to all users.

Quick Tip: Developing code for just one customer is usually a very bad idea: Read Rich Mironov's "Four Laws Of Software Economics" and "The One Cost Engineers and Product Managers Don't Consider" by Kris Gale.

RISK LEVEL
High

STRATEGY
Pay now

Difficult-to-Maintain Code

When code is hard to maintain, it creates significantly more work when we need to add new features down the road. While it is ideal to have working, clean, tested and documented code, it is not always possible.

We should dedicate time at the start of the next release to clean up the code. It will be quick because it is still fresh in engineers' minds, and we can reap the benefits for a long time as functionality continues to expand.

RISK LEVEL
High

STRATEGY
Pay now

Poor Automated Test Coverage

Coding and testing are two inseparable tasks in building a product. When a product lacks proper test coverage, the first we hear of a bug or an issue may be straight from a user.

> Untested code is worse than having no product at all—it is a liability.

Once a product has reached a certain size (e.g. after a year), each new release requires regression testing to ensure that the old features still work. Performing these

regression tests manually, without the benefit of automated tests, is time-consuming and error-prone. Furthermore, the cost of not having proper automated tests, and thus poor test coverage, is paid multiple times because each new release requires several cycles of manual regression testing.

The math in this case is straightforward: compare the cost of creating the automated tests against the time it will take to run the tests manually over the next releases for the next two years. Then add the risk of missing bugs because manual tests are typically less thorough than automated tests. The result is almost always in favor of writing automated tests.

The code that proves that a product works is as valuable as the product itself.

Automated tests need to be developed concurrently with product code at all times.

RISK LEVEL
High

STRATEGY
Refinance

Re-architecting the Product

Any architecture has a shelf life. For any product that we build, we know that the architecture will need to change at some point—due to product evolution, higher performance requirements, availability of new frameworks—and it is necessary to plan for this. It is not a matter of “if” we will need to re-architect, it is a matter of “when”.

We can plan for this in a couple of ways: one is to "budget" this re-architecture on the product roadmap. As a rule of thumb, any two-year product roadmap should reserve at least one major re-architecture project in that time frame. Similarly, whenever we raise a new round of venture funding, or other investment, we should make sure to include a budget for re-architecture projects that will be necessary to achieve the business objectives that underlie the funding.

RISK LEVEL
High

STRATEGY
Refinance

Legacy Code

Legacy code can be defined as old code that has been developed by engineers who are no longer on the project. It typically runs on old infrastructure and is more vulnerable to security exploits and performance issues than newer code. In almost all cases, legacy code lacks automated tests and has poorly documented test cases.

Because it requires reverse-engineering each area of code that needs updating, adding a new feature may require more time spent reverse-engineering than actually writing code for the new functionality. Furthermore, without tests in place, there is a risk of breaking other existing features without knowing about it until after the update has shipped.

> Retaining legacy code is akin to building up debt on a high-interest credit card.

If we wait too long, we end up paying two or three times the principal for the small conveniences early on. It is essential to the health of the company to assess options realistically. Assuming that legacy code will continue to run unmaintained without major incidents is asking for trouble. Either we invest in progressively bringing up the code to adequate quality so that we continue selling it, or we must plan its end-of-life.

RISK LEVEL
Medium

STRATEGY
Do the math and revisit regularly

Operationally Cost-Ineffective

When code works but is costly to operate in the datacenter because of past architectural choices, it becomes technical debt. For example:

- It requires too much CPU or RAM
- It does not auto-scale, or cannot be clustered (which forces provisioning of servers for peak load)
- It does not provide proper metrics or alerts to the Ops team, leading to a high Ops headcount
- It uses commercial software packages that should be replaced by open source solutions
- It uses inefficient open source solutions when commercial software would be better

Under these circumstances, we can compare expected engineering effort against savings in operations. We must also factor in the cost of missed opportunity, when engineers working on cost reductions could have developed new, revenue-driving features. Calculations

must be revisited periodically as the company grows to check whether they still lead to the same conclusion.

RISK LEVEL
Medium

STRATEGY
Pay now in increments

Lack of Documentation

When code lacks proper documentation, it becomes harder and harder to work with it or update it later on. Alleviating this type of technical debt can increase ROI by accelerating the onboarding of new hires, diminishing the risk of human single points of knowledge (and thus single points of failure), and increasing velocity on future enhancements.

After each release, developers should refresh code documentation. For older code without documentation, it should be treated as legacy code and given an improvement plan over time.

RISK LEVEL
Low-Med

STRATEGY
Pay soon

Code Known to Break at Some Point

As engineers code and release a feature, they may identify deficiencies that are not harmful now (corner use cases, performance), but might become so later under certain circumstances (if the number of users double, for instance).

To deal with this type of technical debt, we must add an enhancement request to the backlog and tackle it before the risk becomes material.

RISK LEVEL
Low

STRATEGY
Get funding to pay it back as soon as possible

Strategic Technical Debt

Sometimes taking on technical debt can be a powerful strategy. For example, we may build an MVP (Minimum Viable Product) to attack new markets. This MVP is, by design, as simple as possible so that it can be built fast, and thus carries technical debt compared to what the mainstream product will be. Because of this, we cannot forget that this product will need to be "hardened" once this new market takes off.

As a consequence, we also need to have a plan to fund this hardening, which can be a lot more effort than the MVP itself. It can be funded either through profits from the new sales or a new round of investment.

RISK LEVEL
Low

STRATEGY
In most cases, put in the backlog to pay later

Non-Optimal Implementation

Engineers often figure out the "right way" to implement a functionality right after they complete it. However, realizing there is a better alternative does not always lead to technical debt.

For example, if a developer thinks of a way to make an app 10x faster once they have finished the first implementation but the current version still meets user standards, this improvement is not technical debt and should be tracked in the backlog as "enhancement ideas".

We must assess whether our implementation meets today's performance and usability standards. If it does, then improvement ideas are not technical debt, but enhancement suggestions.

MANAGING TECHNICAL DEBT DAY-TO-DAY

Here are some common scenarios on how to apply the recommendations presented above.

Do We Need a Dedicated Team for Technical Debt?

Some companies have teams dedicated to maintenance, often called "sustaining engineering." While this may work for some large enterprises, it often creates a couple of problems: Firstly, the engineers working on new features may feel that they are superior to those working on the maintenance backlog. Secondly, the "1st class" engineers may be disincentivized to focus on quality knowing that the "2nd class" team will fix their bugs later.

In addition, the amount of resources allocated toward bug fixes and minor enhancements should vary from release to release to match the current needs of the business.

> Sustaining engineering teams may lead to "moral hazard", whereby the development team relaxes its standards of quality.

The same applies to reducing technical debt. Some preventive technical debt reduction should be performed at each release—upgrading all packages, frameworks, and tools to the current version—while the rest should be prioritized in the backlog alongside new features, based on the benefits each brings to customers and the company.

Managing Technical Debt in the Backlog

The backlog review is when engineers need to speak up for prioritizing technical debt. More than anyone, they understand the impact of ignoring important items in the backlog—but to get their point across, they need to translate their requests into a compelling business case for the rest of the stakeholders.

Compare the two arguments for rewriting a module:

> Argument #1: "We need to rewrite this module because it is very poorly written."

> Argument #2: "Once we rewrite this very poorly written module, it will take us half the time currently forecasted for these five items on the backlog—in other words, if we spend one week on the rewrite, we gain two weeks on these five items."

The first argument will never win against a new feature or enhancement requested by customers. However, the second shows real value: an extra two weeks of engineering resources.

Strong Arguments for Prioritizing Technical Debt Reduction

- **ROI**: Fixing it will increase velocity on upcoming roadmap items, evaluated over a period of 2-3 years
- **Risk**: The product will break if we do not fix it. Even if we cannot forecast the breaking point, we know that the likelihood is increasing over time, and the penalty in terms of cost and customer pain is too great to risk
- **Security**: It will make us more vulnerable to security breaches and could cause catastrophic damage
- **Usability**: It is hurting usability, and we can measure this with tools like Mixpanel or Amplitude. There is no point in building new features if the existing functionality does not engage our current users

Dealing with Outdated Architecture

In a typical growth scenario, a startup begins with a simple implementation of its product. In its search for product-market fit, the product evolves and new features are added. Eventually the initial architecture no longer fits customer needs—**this is completely normal and must be anticipated and planned for**, by reserving funding in the future as well as time on the roadmap.

Engineers will intuitively know when the company has reached this point as new features will require more work retrofitting existing code than implementing the new functionality itself.

For companies raising funds, it is highly recommended to raise enough new money to not only implement new features, but also to re-architect the code. By actively managing this process, we can avoid a perpetual state of crises and quick-fixes.

Fixing Buggy Code

There are many reasons for software to contain buggy modules. The code may have been designed poorly at the onset, enhanced too quickly over time, or not properly tested before release. While taking the time to fix this code is unattractive, the payoff for refactoring over time is well worth it. The key is to frame the decision in the context of a two- to three-year ROI.

Handling Debt from the Previous Release

Rare is the release where engineers have time to write clean, well-documented code that follows best practices and is fully supported by automated tests. It is important to follow a process where code from each release is refreshed before the next to avoid an unhealthy accumulation of technical debt from mistakes or shortcuts during implementation.

> We should dedicate time to bring the most recent code up to standard at the beginning of each new release. The few days that we spend will be paid back over the next few years by the absence of bugs, faster regression testing and the readability of the code when we add new features.

GAINING CONTROL OF YOUR TECHNICAL DEBT

Technical debt is created the moment a product is first released, and typically continues to accrue during each subsequent release. This is a natural process. If neglected it has the potential to irreversibly damage customer relationships, throw a wrench in the product roadmap, or even bring the entire operation to a standstill. Like financial debt, the longer we wait to deal with it, the more debilitating it can become.

On the other hand, when we take a proactive approach to technical debt and use a rational decision-making process, we can allocate the right engineering resources to paying it down when it makes sense for the business.

> Whatever the size of our company, managing technical debt—just like managing financial debt—is a "life skill" that we all need to master as business and engineering leaders.

By applying the analysis framework presented here, we will communicate effectively and make ROI-based decisions. This will reduce risk, deliver more value to our customers, and increase our competitiveness in the market.

THE AUTHOR

Bernard Fraenkel

PRACTICE LEAD IN SAN FRANCISCO

Bernard Fraenkel has over 20 years of experience leading engineering teams that deliver mission-critical software applications for the enterprise. In the past few years, his teams have delivered SaaS and mobile applications in markets as diverse as digital marketing, mobile banking, distributed storage, education, augmented reality, social messaging, email, web hosting, e-commerce, neuroscience and distance learning.

Silicon Valley Software Group (SVSG)

Silicon Valley Software Group (SVSG) is a team of Silicon Valley CTOs dedicated to leading organizations around the world to the forefront of innovation. SVSG CTOs bring high-level and hands-on technology expertise into organizations by providing guidance on emerging technology trends, incorporating best practices into product development, and navigating growth at every stage.

SVSG consultants are recognized authorities in their fields. Each has experience as a Chief Technology Officer and has founded one or more successful ventures of their own. Many have launched products for major organizations including Google, Microsoft, Yahoo Japan, The United States Government, The Government of Malaysia, TRUSTe, US Open Data, UpWork, Deutsche Bank, MIT, UC Berkeley, Stanford, and The Robotics Institute.

SV/SG

1161 Mission St.
San Francisco, CA 94103

+1 844 946 SVSG

www.svsg.co