mercuryworks

IDENTIFYING AND MANAGING TECHNICAL DEBT

Technical debt has been a phenomenon since the first mainframe punchcard. However, with our economy being dominated by businesses viewing themselves as “software companies” (and with “product development” referring to software more often than any durable good), an increasingly broad set of stakeholders feel the effects of technical debt.

Combine that with this decade’s current and impending Great Resignation–where widespread turnover is primed to occur due to employee burnout and dissatisfaction in their companies’ ability to handle the next crisis. Managing software teams’ time efficiently and helping them avoid redundant, unnecessary work becomes not only a best practice, but an important advantage in attracting and retaining top talent.

78% of developers agree

that maintenance of legacy systems and technical debt have a significant impact on their personal morale.

Source: The Developer Coefficient, Stripe, 2018

WHAT IS TECHNICAL DEBT?

Ward Cunningham, hugely influential computer scientist and co-creator of Extreme Programming, coined the term “technical debt” nearly three decades ago:

> *“Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite… The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation.”*

In more pragmatic terms, technical debt can be defined as quick, dirty, or otherwise inadvisable development that leads to more work later. This work becomes more and more critical as time wears on so that the shortcuts taken won’t lead to bugs or make future changes hard to implement.

TECHNICAL DEBT SMELLS

As a technical manager, product manager, or Agile product owner, how can you get a feel for the level of technical debt in your digital products? How can you gauge whether technical debt is increasing or decreasing?

Kent Beck coined the term "code smell" as a surface-level indication (in other words, you're likely to notice it if you're near it) that usually corresponds to a deeper problem with the code. Just as food that smells a little funny might not guarantee that it is spoiled, code smells are usually not as obvious as outright bugs, but if you're tuned in you'll definitely notice them.

THE FOLLOWING ARE SOME CLASSIC TECHNICAL DEBT SMELLS:

- **Simple Functional Changes or Additions Take a Long Time**
  For instance, you have determined that you need to add a "salutation" and "suffix" field to your CRM application and your development team greets you with pained looks and/or racks several days of work to make the change. The amount of work to unpack and remediate debt-ridden logic while building out your additions is adding major unexpected cost to your project.

- **Bug Fixes Take a Long Time**
  A problem comes in from the field that your web application is rendering a key page in poor form on smartphones. The bug takes three days to remedy, your team is putting in late hours to get it fixed, and senior development staff is called in to assist.

- **Code Changes for Small Additions Frequently Lead to Bugs**
  During the addition of the "salutation" and "suffix" fields to your CRM application, three reports break and the data update feature breaks. Seemingly unrelated features regress to broken states for no apparent reason.

- **Developers Actively Avoid Being Assigned to the Code**
  Your development team actively avoids being assigned to the team working on your apps. Word spreads quickly among the corps when an app's technical debt has grown out of control and soon it will be difficult to get your developers of choice to voluntarily work on your app.

The Application Slows Down

You use your app for day-to-day work and the slowdown from how it worked a year ago is palpable. Technical debt has a way of gumming up the works in an application. Workarounds mount in your backlog and inefficient legacy code dominates the application, bringing it to a crawl.

Any of the above sound familiar? If so, it's probably time to dig deeper and involve technical staff to evaluate your software for the presence of technical debt.

WHAT CAUSES TECHNICAL DEBT?

While it may appear that technical debt is most often a result of developer carelessness or ignorance, the fact of the matter is that the situation is much more nuanced. Holding such a viewpoint would be like blaming the automotive assembly line worker for your car's oil changes and brake pad replacement: certainly a car could be built that would not require that maintenance, but the cost would be so high no one would buy such a car.

It helps to understand the different types of technical debt, each of which have their own likely set of causes:

NAÏVE, RECKLESS, OR UNINTENTIONAL TECHNICAL DEBT

These are different names for a form of technical debt that accrues due to irresponsible behavior or immature practices on the part of the people involved. In our experience it is very rare to find this kind of technical debt stemming from conscious irresponsible development behavior: no one intentionally sets out to write faulty or sloppy code. However, it is very common to find this kind of debt originating from developers not trained in current and robust development techniques. It can also arise when little architectural planning has taken place or the toolchain the development team uses is immature.

LIKELY CAUSE

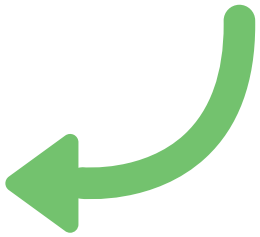
The truth is that not all teams are equipped or prepared to consider the implications of technical debt. A lack of understanding of DevOps principles and process results in unintentional fallout. The technical debt experienced here comes from isolated work efforts where teams don't collaborate and share knowledge effectively, inexperienced development staff on teams without a strong mentorship culture, or uncoordinated outsourced teams.

MYOPIC TECHNICAL DEBT

Myopia is another type of technical debt that can take root during initial development of a major new digital product. When developers or product owners do not look far enough into the future to realize the type of versatility that will be needed for a feature, the team can often author short-sighted code that will need to be re-visited later to provide the needed flexibility.

This type of technical debt can manifest in development practices that are out of touch with modern DevOps practices that prioritize ease of scaling. One example is choosing to hard code the options in a select list when the end user may want to be able to add options to it in a dynamic fashion. Choices like tight code coupling and lack of modular components mean that the software is not flexible enough to adapt to changes in business needs. Lack of testing and documentation can also fall into the myopic bucket, making regressions more difficult to identify and debug.

LIKELY CAUSE



UNAVOIDABLE TECHNICAL DEBT

A form of technical debt that is usually unpredictable and unpreventable and accrues through no fault of the team building the product. An example of the “unavoidable” flavor of this kind of technical debt would be the use of ASP.NET forms just on the brink of Microsoft’s release of the MVC framework.

LIKELY CAUSE

In this case, the developer may have had an idea that a new standard in .NET client-side development was coming, but within a small number of years the development community moved away from ASP.NET and thus the resulting application was laden with unavoidable technical debt.

STRATEGIC TECHNICAL DEBT

Sometimes taking on technical debt for strategic and economic reasons is a sensible business choice. This is largely a result of conscious decisions around needed speed-to-market or plans to remedy technical debt after meeting a market-imposed (often regulatory in nature) deadline.

LIKELY CAUSE

Technical debt here builds up as an outcome of rushed and/or uncompleted changes and should be dealt with as soon as possible to mitigate future negative impact.

NOT ALL TECHNICAL DEBT IS BAD

Between the significant costs to teams and users and the impacts on productivity, efficiency, performance, and morale, it would be easy to assume that technical debt is always the wrong answer. But sometimes there are sound reasons to plan for the creation of technical debt. There aren't many, but a few reasons that do exist are very good ones.

Strategically allowing or even encouraging the creation of technical debt is a very valid strategy in certain circumstances. One specific example is a product that is planned to have a short life. Since it's not going to live long, it does not have to be built to withstand a strong legacy; we'll only need to make debt payments for a very short period of time and then the app will be shut down. In this case it makes sense to build the product quickly, just good enough to last the short time and then retire it, along with the debt, as soon as possible. It does not make economic sense to gold-plate a short-term solution.

The same can be said for a product nearing its end of life. Eric Ries contends that in a Lean world, the value prototypes bring are to validate theories, give learning, and stir interest/establish the proof of concept. Since prototypes are intended to be thrown away, this approach is a special case of the "don't worry about technical debt for products with a short life".

A survey of 3350 IT professionals found that legacy system integration was the top factor slowing down product delivery times.

Source: The State of Application Development, Outsystems, 2019/2020

FROM THE TRENCHES: SHORTCUTTING FOR A QUICK RELEASE

We once had a client with a key ecommerce project in flight.

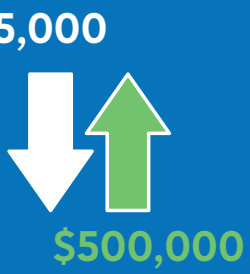
The client's key industry tradeshow necessitated that we ship the product early to make the most of the increased visibility and seasonal demand surrounding the show.

In order to meet the shortened timeline, we took several code shortcuts that we knew were not a good idea for long-term operation of the software but would hasten development to the point we would be able to launch just prior to the show.

As a result, the client captured $500K in revenue that they would have otherwise missed.

Importantly, in the sprint following the launch we came back to the code base, remedied the shortcuts, and bolstered automated tests so as to not leave behind technical debt.

Cleaning up the technical debt after main development increased costs by $5000, but it enabled the client to realize $500K in additional profits, yielding a huge return on investment while still maintaining the integrity of the software for the long term.



One very defensible rationale for intentional technical debt is for a "throwaway prototype." You may have significant interest in a big new application but have not quite achieved full budgetary support to build it in full. You may need a proof of concept for a very reduced budget to determine viability of the application to solve business problems and to get feedback from potential users. As such, a quick prototype with non-production tooling, lower-cost staff, and non-enterprise approaches can be taken to provide the appearance of a near-complete application such that it can be put in front of customers.

This approach gives the opportunity for very realistic feedback on needed features, the things that have the most value, and pointers on what the usability should be like. This information can help you make a better decision on scope and project sponsorship. Then this prototype can be thrown in the trash and a fully-funded enterprise solution can be developed with these learnings - what you have learned and benefited from in this process will greatly exceed the cost of the thrown-away prototype. Whatever you do, do not build upon your rough prototype for the real solution! If you do, you will enter an entirely unprecedented world of technical pain.

CONSEQUENCES OF TECHNICAL DEBT

If your workarounds are working for now, why should you worry about technical debt? Putting an end to the code smells is alone probably reason enough. But leaving technical debt unattended (even if everything is working okay today) can saddle your team with huge costs and significant headaches down the road.

INTEREST PAYMENTS

Technical debt often takes the form of sub-optimal and often rough workarounds within code that developers have used either out of necessity to meet a deadline or out of a lack of skill. Every time developers work with your app, every time a support request comes in, and every time you make small adjustments, your developers will need to make a hard choice: add to technical debt (making the situation worse) or re-write portions of the software to unwind the technical debt and make progress. This is completely analogous to financial debt: either make interest payments on your loan or sink further into debt.

As these aspects of the code base build up, there can come a time far enough into the application's life in which the application is built up of more workaround code than advisably built professional code. At this point, developers will spend more time understanding and working around the various jury rigs than they will contributing to rigorous solutions. Once this happens you are at a tipping point. It will become completely unpredictable how long code changes and functional additions are likely to take.

INCREASED TIME TO DELIVERY

Predominant technical debt and code rot leads to developer and architect discussions and in-depth planning to make changes to the code and still deliver quality product. There will be times when, despite this planning, developers run into pockets of the code base that are more entrenched in outdated technologies or unsustainable implementations than expected and the actual timeframe can stretch out even further.

SIGNIFICANT NUMBER OF DEFECTS

In modern software development, developers and engineers rely on extensive unit, integration, and functional tests to run during each commit and catch any instances of regression - times when working on a new feature breaks an old feature. Code bases with a large extent of technical debt are often absent these kinds of automated tests - and as a result, regressions occur at a higher rate than normally expected.

RISING DEVELOPMENT AND SUPPORT COSTS

Because of the many ways in which high technical debt applications result in lengthier planning, existing code re-writes and sub-optimal coding approaches, the time (and associated cost) for in-house staff to work with the application goes up, along with the cost of frequent out-of-house consulting required at times to remediate particularly thorny code bases.

PRODUCT ATROPHY AND DWINDLING TEAM MORALE

Product owners are not interested in working with the software application, and they avoid activating their teams to work with the application. Thus, rot continues to build. If teams do still work on paying down the debt (or orchestrating complex workarounds that rack up even more), they are less motivated as they are often rewarded with buggy releases, budget overages, and delayed timelines.

DECREASED PREDICTABILITY

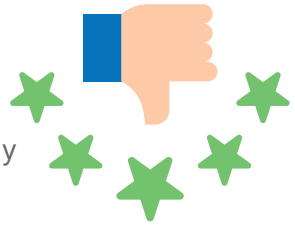
Just as developers, engineers and product owners receive constant surprises from technical debt-ridden software, the business receives less precise signals as to timeline and cost for maintenance and incremental development. Upper management and product stakeholders begin to lose faith in the application and the team as predictability and their ability to make proper business decisions goes out the window.

UNDERPERFORMANCE

While we have spent a great bit of time covering the ways in which working with technical debt-ridden applications makes developer and engineer lives difficult, costly, and drawn-out, technical debt also poses very real consequences to the users of applications. Due to antiquated technologies, algorithmic workarounds, and rushed work, the actual response time, availability, and server/cloud resources of the application will suffer.

DECREASED CUSTOMER SATISFACTION

Rolling together all of the impacts of technical debt above, you're going to see customer satisfaction slip. As such you will soon hear from your end users that they have felt the bite of technical debt.



Bad code costs developers an average of
4 hours each week,
which amounts to
$85 billion annually in lost opportunity costs.

Source: The Developer Coefficient, Stripe, 2018

mercuryworks

TOOLS TO MEASURE TECHNICAL DEBT

If you're at the point where you can determine that technical debt is having an impact on your operations, what's next? Measuring the severity of that impact, especially over time, with monitoring and assessment tools.

More than hanging your hate on an arbitrary level of technical debt, make sure you're using these tools to watch trends and changes from check-in to check-in. Relative changes from one version of the code base to another is a more actionable assessment of the degradation or improvement in the condition of the code.

Here are some of the methods and tools available on the market for this purpose:

SQALE

Software Quality Assessment based on Lifecycle Expectations (SQALE) is an analytical method to assess a software application's source code. It is a generic method, independent of the language and source code analysis tools, that normalizes best practice software development techniques across languages. A SQALE score is comprised of 8 indices that measure key factors such as code reusability and changeability, all contributing to an application's technical debt.

SonarQube

SonarQube is an open-source software implementation of the SQALE tests to yield technical debt scores for a given code base. At time of writing SonarQube works against the world's most popular software languages and platforms including Java, C#, Objective-C, JavaScript and PHP. SonarQube is most often implemented as part of a continuous deployment pipeline to assess code quality upon every code commit and build of an application.

Additionally, an organization can write custom rule sets for SonarQube to apply to put more (or less) emphasis on particularly problematic types of technical debt their teams are prone to.

STATIC ANALYSIS TOOLS

There are several other software products that are intended to measure many of the factors contributing to technical debt. Examples include cyclomatic complexity (a measure of the number of possible routes through your software logic), static analysis (automated review of source code to identify lacking best practices) and coupling (the extent to which one software routine depends in turn on another to complete its intent). Software development tools such as Crucible and FxCop are often implemented to keep a lid on these ill effects.

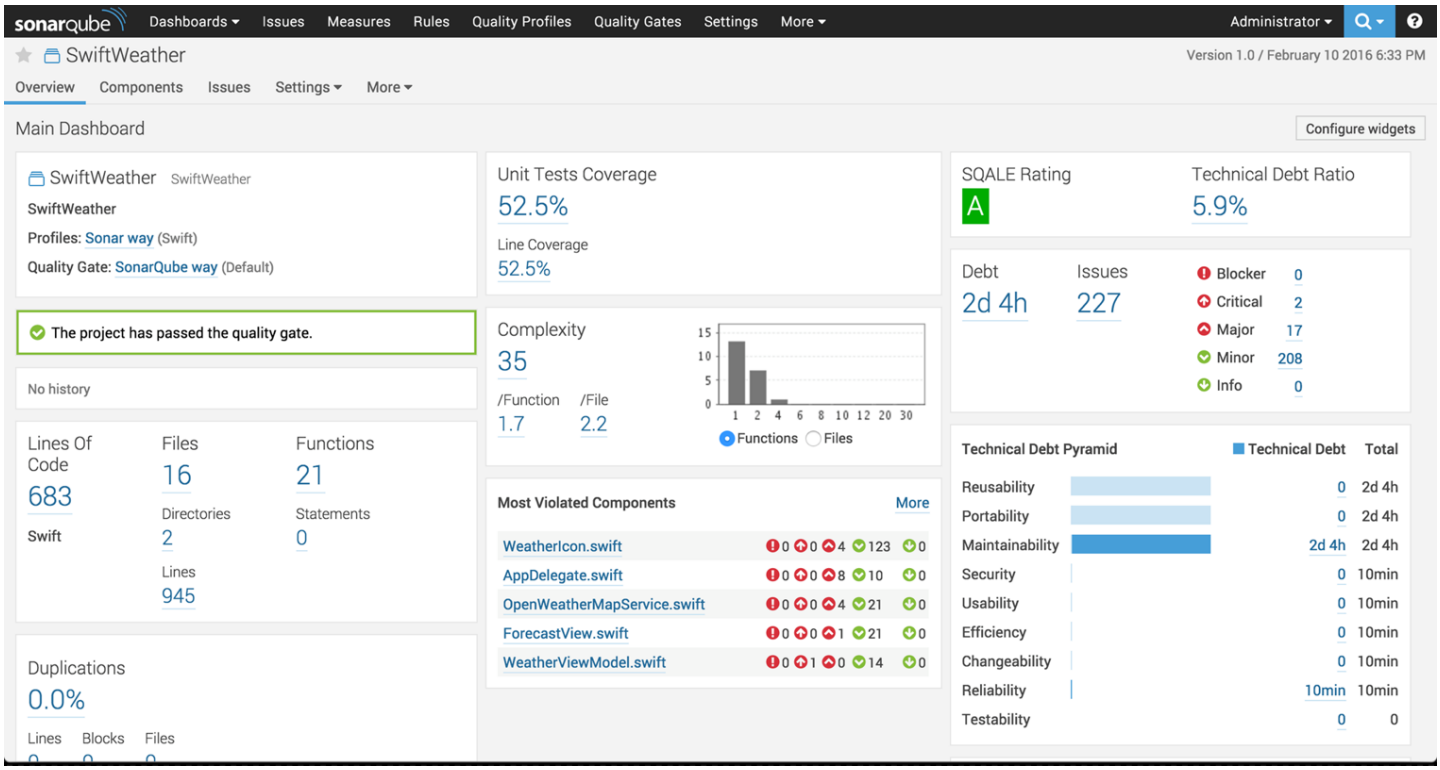


Figure 1: A sample SonarQube dashboard

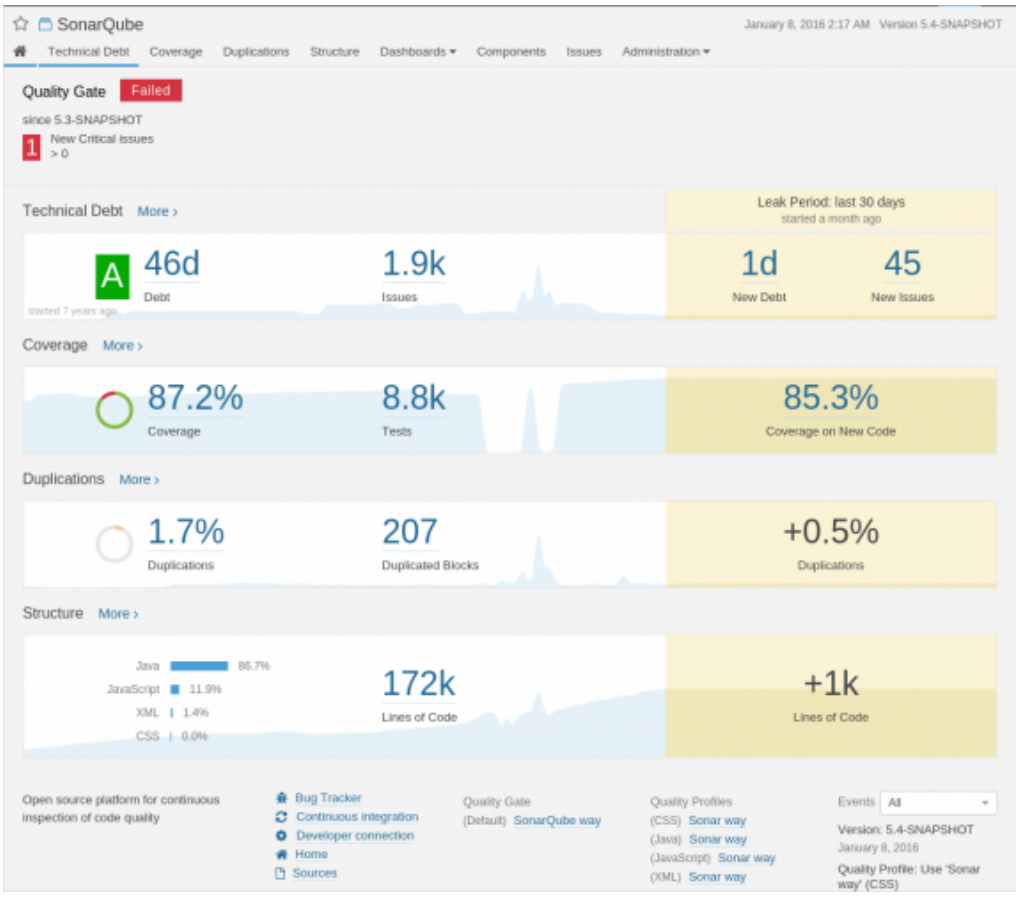


Figure 2: A sample snapshot of a quality gate failed by SonarQube

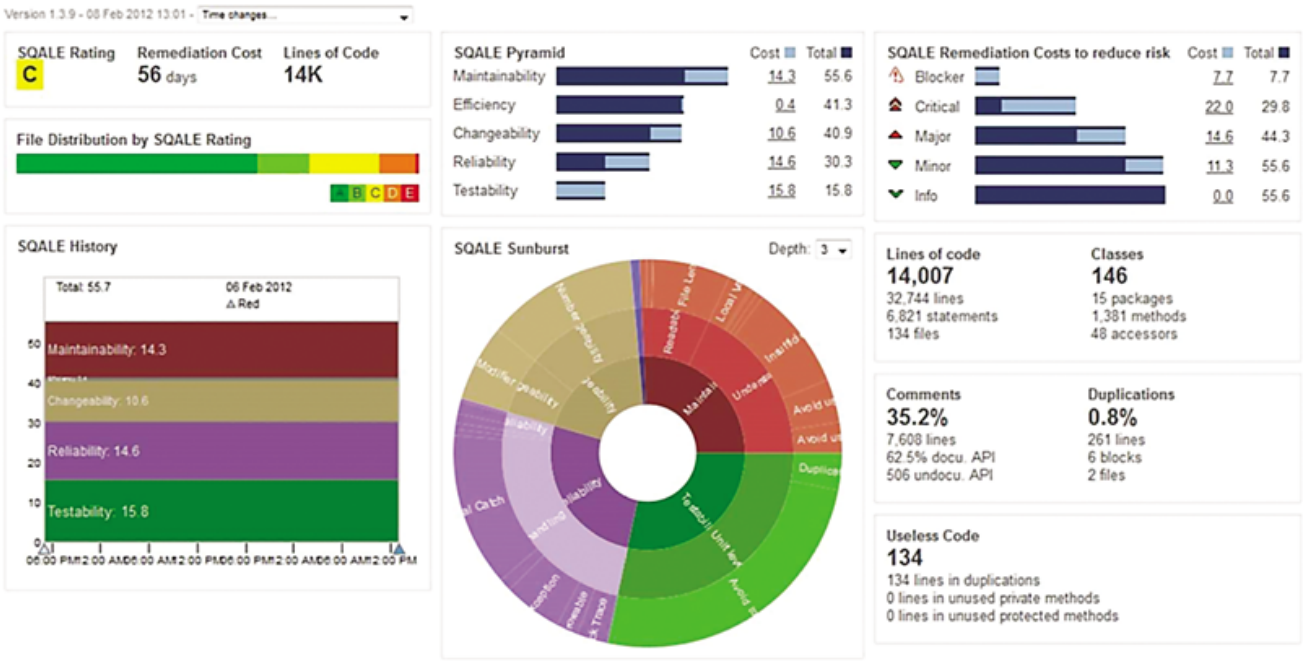
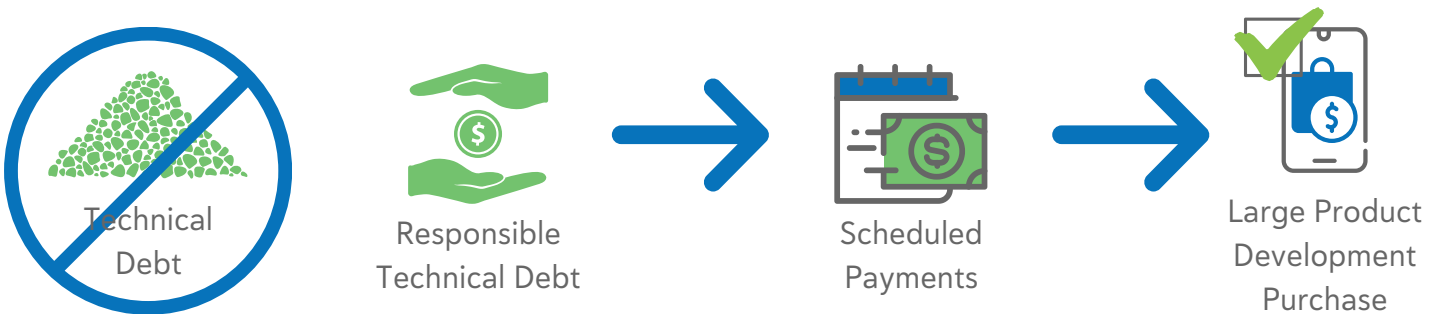


Figure 3: A deeper dive into a SonarQube run that exposed significant technical debt

DEALING WITH TECHNICAL DEBT

The question is not so much whether or not you are going to take on technical debt but rather how quickly, what kind, and what you are going to do about it in the long term. Just like consumer debt, technical debt will show the magic of compound interest but in reverse - just as small investments to a 401k over time will build up a nice nest egg for retirement, a bit of technical debt added to consistently over time will result in a nearly insurmountable pile of technical debt. Taking on a responsible amount of technical debt and making scheduled payments will keep your leverage low and still enable you to make those large purchases that are necessary for large product development.



PREVENTING UNPLANNED TECHNICAL DEBT

Developing software without a concerted long-term quality plan is the fastest road to chaos and debt. But tragedies of lost productivity and mounting costs can be prevented by building solid DevOps and Agile principles into the development process. Here are some of the industry standards that can help keep debt as low as possible:

DEFINE AND SET COMMON STANDARDS

One of the most important social compacts among a team of application developers is a solid Definition of Done - a Definition of Done (DoD) is a checklist that enables a team to say that a feature is shippable. Key elements of a DoD include an agreement on extent and type of tests, acceptable code style, degree of peer review, amount of documentation, and acceptable frameworks and versions. A rigorous DoD will make the entire team aware of the level of quality expected of committed code - and that not meeting the DoD will likely result in incremental technical debt that will require more work.

FROM THE TRENCHES: DON'T FORGET THE TESTS

Unit, integration and functional tests are all very important to develop in tandem with the actual software that drives your released product. Some developers are versed in Test Driven Development (TDD) and follow it in day-to-day coding. However, when things get rushed and project deadlines are constrained the development of tests is often the first thing that is skipped "to save time". Having good test coverage, running tests on every code commit and stopping development until the built-up set of tests pass is a key practice to keep technical debt from mounting in your software.

COLLABORATE

Pair programming and peer reviews are both highly encouraged Agile methods of vetting the approaches, consistency, and elegance of code. Both mechanisms enable more senior developers to mentor apprentice developers and for apprentice developers to make senior developers aware of new technologies. Having a second pair of eyes and hands on the code base tends to bring down the level of technical debt since inadvisable code is likely to jump out to someone new to the code who did not author it.

AUTOMATE WHERE POSSIBLE

There is a practical limit to the amount of pair programming and peer reviews that can be done on an ongoing basis in most shops. For this reason, MercuryWorks has found automated pipelines with quality checks to be a reasonable base level proxy and a timesaver to reduce the amount of naïve technical debt introduced into a product's code base and increase code consistency.

Tools like Grunt, Gulp and SonarQube, along with associated code linters, will help point out sloppy code or approaches counter to the organization's DoD - and since it is done by a machine upon every check-in, there is essentially no overhead added to the development effort.

KEEP YOUR PLANNED DEBT RESPONSIBLE

Just as you should avoid large levels of debt on high-interest credit cards, developers should strive to avoid knowingly take on high-interest technical debt. High interest technical debt is the kind of technical debt that is present in a module that is often modified or depended on by other modules and is either brittle/difficult to change or throws a lot of exceptions and performance problems. Rather than having code routines that carry out single operations, we have now spread the debt throughout the application. It will cost the team money every day as we add features.

Conversely, those parts of the app that aren't frequently touched can be prioritized lower, even if their implementation is particularly poor.

FROM THE TRENCHES: GREEN STAFF AND NEW TECHNOLOGY PILE ON THE DEBT

Several years ago we empowered a group of apprentice developers to develop a software product using what was, at that time, a new release of Microsoft's Entity Framework. They were given a lot of leeway to build things with advisable design patterns, and sprint demo after sprint demo, they delivered and showcased some outstanding product.

After several months of development, we successfully released to production. Almost immediately there were problems reported from the field including timeouts and flaky functional performance. It turns out Entity Framework defaults to "lazy loading" which the developer did not override-resulting in extremely slow SQL queries and application performance grinding.

We stopped new feature development and spent a sprint modifying LINQ queries to load data directly into Data Transfer Objects which shored up performance. This team can now develop functionality soundly within the code base.

With consumer debt, a poor enough credit rating will render you unable to buy a new car or finance a home and remedial credit action is needed to get that new car or home. Such is life with technical debt in software also. The software team needs to stop new feature development and remove the technical debt so that not only to return stability to the product but to allow additional features ton be added without immediate breaks to prior functionality.

Consider making these kinds of lump sum payments against the software mortgage that is your technical debt so that the application's credit rating returns to a reasonable level and typical spending habits can resume. Know, though, that many experts consider this a "condition critical" move only and not one that should be leaned on regularly.

HOW TO HANDLE TECHNICAL DEBT

At MercuryWorks, we lean on "Uncle Bob" Martin's method for handling technical debt inspired by the old Boy Scout rule: "Always leave the campground cleaner than you found it." By intentionally improving the environment for the next group of campers (developers) the team can move into the next sprint knowing that they do not need to dedicate the first part of the sprint cleaning up a technical mess from the prior sprint.

Team members must be empowered and culturally encouraged to clean up a technical mess they identify while working on adding new features. This debt service should only take place to a reasonable threshold, though, rather than completely re-factoring and re-implementing a routine while simultaneously developing the committed new feature. While such refactoring is admirable it's not often realistic - do enough to make the situation better and flag any extreme problems as a future user story in the product backlog.

Which brings us to the topic of "balloon payments."

In his book Essential Scrum, Kenneth Rubin warns against the practice of regularly carrying out an entire sprint for refactoring or technical debt reduction work (a balloon payment) on an application. In Rubin's opinion, the regular use of balloon payments subtly encourages the team to accumulate technical debt, avoid dealing with technical debt as they find it and rather postpone to a concerted balloon payment sprint (which may never come). This practice can also lead to regressions because many areas of the solution that were not recently developed are touched in a very short amount of time during the "bug hunt."

Instead, Rubin recommends that the team handle technical debt while performing client-valuable work within each sprint. One example of doing this is addressing trouble tickets that come in from the field during a sprint. When technical debt is found to be at the root of the ticket it should be flagged as such and then the team should determine if can be worked into the current sprint.

Another in-sprint approach is that when a feature is being added or extended to take note of any technical debt (particularly if it is impeding progress) and refactor or otherwise reduce debt while constructing the new feature. Not only is this likely to be a more efficient approach but should also improve the bandwidth committed to thinking about removing the technical debt in the most optimal fashion.

Finally, another ongoing method to actively reduce technical debt is to write stories and add them to the overall product backlog on a regular basis and assign a percent of the product budget to address this identified technical debt. In this way the Product Owner can occasionally work high-priority technical debt stories into sprint backlogs.

MERCURYWORKS CAN HELP

MercuryWorks is a leading digital application and professional services firm obsessed with web applications, native applications, DevOps strategy, and legacy application modernization. For more than 23 years, MercuryWorks has been providing clients with creative web-based solutions. During that time, we have designed and developed thousands of web applications and websites to meet our clients' diverse and demanding business needs. Our solutions stand the test of time by addressing our clients' immediate needs with an eye towards future growth.

Solutions implemented by Mercury New Media include custom web application design and development, mobile application development, and an integrated line of business solutions. We accomplish this through a deep discovery process to determine the most appropriate mix of technologies, which often include some of the following Mercury New Media areas of expertise: HTML5, CSS3, RWD, jQuery, Bootstrap, Angular, React, .NET, Azure Web Services, and Microsoft Power Platform.

Have questions regarding your web application's technical debt and what can be done about it?

CALL US:
813.551.3144

EMAIL US:
INFO@MERCURYWORKS.COM

mercuryworks