Technical Debt: A Beginner’s Guide

Technical debt sounds like a financial term, which is where this programming theory has its roots. When it comes to software development, technical debt is the idea that certain necessary work gets delayed during the development of a software project in order to hit a particular deliverable or deadline.

Developers inherently deal with this issue: the idea of developing a perfectly designed app, software, system, etc. Clean, well-designed code allows future iterations and innovations to be more easily implemented. However, in a business setting, time and resources often mean that developers don’t have the ability to produce clean, perfect code before delivering a product.

This is where technical debt comes into play. The concept recognizes a trade-off between perfect products and the short
timelines often required for product delivery. While it can be useful to incur some debt, as with money matters, there’s a limit to how much debt you want to carry.

In this article, we’ll take a look at how technical debt is affecting your IT team’s workload, and ways to manage the debt.

**Defining technical debt**

First up, let’s talk about the ins and outs of the theory. The concept was first discussed in the early 1990s by Ward Cunningham, one of the founders of Agile programming. Cunningham continues to speak on this topic. The concept is that the “debt” represents the extra development work that arises when code is implemented in the short run, despite it not being the best overall solution.

**Get the IT Buyer’s Guide for Service & Operations**

This IT Buyer’s Guide helps you understand what to look for and how to choose the right tools for IT service and operations.

Get the Buyer’s Guide

What is the exact definition of technical debt? The Software Engineering Institute at Carnegie Mellon University notes that technical debt “conceptualizes the trade off between the
Another perspective comes from industry analyst, Gartner, which defines technical debt as the deviation of an application from any nonfunctional requirements. Here’s a general example: Imagine that you’re a software developer and you have to add a new functionality to a software or system. There’s two paths to choose:

- The easier route – made up of messier code or design, will get you there faster.
- The harder route – made up of cleaner code and design that takes a lot more time.

Like monetary debt, technical debt can accumulate “interest”. In this concept, the interest is the increasing difficulty it can be to implement changes later on, especially as a software project dominoes through multiple phases. The longer technical debt is ignored or unaddressed, software entropy can occur.

(It is like cleaning by hiding everything under the couch – it’s tidy for now, but nothing is in its correct spot. You’ll eventually have to spend time properly cleaning and organizing it down the road in order to find something you’re seeking.)

Regardless of how you choose to define it, recognizing the importance of the core concept provides the best starting point for reducing risk. This means honing the ability to acknowledge the fact that issues related to the source code have a negative impact on productivity and devising a plan to address same.

**Technicalities of technical debt**

Technical debt does not include ignoring or delaying things like functionalities, which were intentionally not built.

Instead, the debt is incurred when IT chooses to delay better coding and building internal pieces that should be there, for various reasons. But this choice to ignore certain pieces is
understood that it will impede future development and delivery if left undone — and if the team doesn’t go back to resolve the mediocre code or address known bugs, the result is the cumulative interest.

While technical debt can refer to any part of software development, it is commonly associated with extreme programming, particularly refactoring. Refactoring is the restructuring of existing code as part of the development process. The two main reasons for refactoring are poorly written legacy code and evolving solutions as a problem is better understood.

**Identifying Technical Debt**

Below are some warning signs that a project has created technical debt:

- **Code smells** are much more subtle than logic errors and indicate problems that are more likely to impact overall performance quality than cause a crash.
- Higher levels of **complexity** — when technologies overlap each other.
- **Product bugs** that will cause an entire system crash.
- Issues with **coding style**. Developing a coding style guide and sticking to it, will help to overcome this issue.

The key thing to note at this stage is that, if left unaddressed, these red flags can result in higher total cost of ownership, longer time to market and reduced agility. They can also negatively impact customer experience and security.

**Types of Technical Debt**

Now that we know what technical debt looks and smells like, it’s time to categorize its occurrence.
Planned Technical Debt – This type of technical debt occurs when the organization makes an informed decision to generate some technical debt with the full understanding of the consequences (risks and costs). In the case of planned technical debt, it is critical to be as precise as possible in terms of defining the compromises the organization intends to make. An example could be: “In order to meet the new release deadline of November, we have decided to forego writing unit tests in the final three weeks of the project. We will write these tests after the release.”

Because these decisions can accumulate quickly over time, it is imperative to maintain a record of them. Doing so will increase the likelihood that technical debt will get addressed and paid down quicker. Otherwise, it will quickly be forgotten, potentially costing the organization big time in the long run.

Unintentional Technical Debt – This refers to unplanned technical debt that arises due to poor practices. For example, a design approach that ends up containing many errors. This type of technical debt sometimes occurs as the direct result of poor communication within the organization or when the goals of Development and Operations are misaligned.

Unavoidable Technical Debt – This occurs due to changes in the business and the progress of technology over time that present better solutions. It typically arises when scope changes are requested mid-project, that result in an immediate cost such as adding a new feature to an existing design to better support mobile delivery. In short, technical debt is created when new business requirements make old code obsolete.

Good vs. bad: reasons for technical
Just like financial debt, there can be good reasons for technical debt — but it’s important to know that going in, so debt doesn’t get ahead of the team, slowing down progress and future deliveries.

A good reason for incurring technical debt is often because a delivery is more important than the internal cleanliness of code or smoothness of functionality. If the product works for the user, despite not being the best or cleanest product, then delivery may behoove your company (in terms of time-to-market, revenue, etc.). If, however, business needs require perfect design, you’ll take your time to deliver a cleaner product.

A bad reason for incurring technical debt is because the team chose to focus on other areas that are more innovative or interesting but less important.

Even if you have a good reason for incurring debt, choosing the messier option that ensures a quicker delivery isn’t the end of this design choice. Instead, the team must return — at some point — to this piece of design to add more functionality or revise it. The longer the team waits to deal with an issue, the more likely it can cause more issues or get buried in code. This is the interest: the time in the future you’ll have to spend cleaning it up to get it to jive with new changes you have to make.

Dev teams must be balanced. Looking at many sides of the issue to determine whether or not to incur technical debt.

Avoiding racking up technical debt

While financial debt is easy to track, technical debt is not inherently a metric. With some tweaking, the theory of technical debt can be translated to certain metrics, such as money spent versus time working overtime to pay down debt.
interest. It can also appear as less productivity from a team – which is also difficult to measure.

Experts recommend tracking your technical debt to keep it from getting too unwieldy. As the debts can survive multiple development cycles, tracking it is essential. Here’s how:

- Start a list of technical debts. (This includes all instances where the developers know the code isn’t as clean as it should or needs to be for future development.)
- List and group deferred tasks into workable units.
- Note the consequences of ignoring each unit.
- Keep the list visible.
- Inform teams that rely on delivery releases – like marketing, sales, etc. – that you’re working on technical debt, so that each new release cannot include only new features.
- Schedule regular and frequent time to pay off technical debt.

Taking Nonfunctional Requirements (NFR) Into Account

Using the Gartner definition of technical debt necessitates a deep dive into nonfunctional requirements or NFRs. Every system has NFRs though they may not always be defined or tracked. NFRs refer to constraints on a software system and include the following seven attributes: compatibility, security, reliability, usability, efficiency, maintainability and portability.

In contrast, functional requirements outline what the system should do and are much easier to measure and analyze for example in the case of epics. Achieving both functional and nonfunctional requirements are critical to ensuring project success.
Shifting focus back to NFRs, there are also important sub-characteristics that exist, namely explicit and implicit requirements. As you may have guessed, explicit requirements are predefined by key stakeholders, for example, a detailed design specification. On the other hand, implicit requirements are those that are expected but which have not been stated unequivocally. An example of an implicit requirement would be ensuring the ease of use of an application.

Having an understanding of these nonfunctional requirements will be important when it comes to identifying technical debt.

Managing Technical Debt: Best Practices

Assessment

In identifying technical debt you might have come across some key signals. For example, your product’s performance ratings could be on the decline or your developers may be taking much longer to iterate. But how do you measure this? What is the true cost of technical debt?

One way to come up with this measure is to look at the number of days developers would need to spend reducing technical debt by performing activities such as refactoring or replacing the application. Once you attached a dollar amount to these functions, you could then compare this data to other milestones, like number of remaining days before the release date. This will provide an excellent cost/benefit analysis and aid in communicating more effectively with the rest of the organization.

In addition to providing a current status update, it is also important to generate estimates of how you expect technical debt to change over time.
Communication

One of the most important steps to take in managing technical debt is to acknowledge that it exists in the first place and share that discovery with key stakeholders. It should be the responsibility of IT management to set the tone and communicate to non-IT managers about the true cost of technical debt. The head of IT must also explain the importance of paying down technical debt sooner rather later.

Implementation

There are three options to consider in terms of managing technical debt.

- Waive the requirement altogether. In other words, the organization decides to live with the system as it is and no longer deems the requirement as necessary. If you cannot waive the requirement, then you will need to refactor or replace the application.
- Refactor the application. This option is aimed at reducing complexity, removing duplicates and improving the structure of the code. Refactoring is the only way to improve a code’s internal structure without changing the behavior of the program.
- Replace the application. While this will introduce new technical debt, the idea is to address it quickly and minimize it as much possible.

Implementing Agile practices to minimize technical debt

Users of the popular Agile approach to programming know that “done” is a relative term. In traditional development environments, software is done when it’s shipped to the users. But in Agile environments, iterations of work are frequently delivered to the user, improving bugs and issues that develop in each release. There is no “done”.

Agile relies on reducing the scope of a release to ensure quality work, instead of promoting a large number of functions in a release.

Because Agile embraces increments and iterations, instead of finished projects, implementing Agile theories can be a good way to stay on top of technical debt. Always thinking in short bursts of work makes it easier for IT teams to tackle smaller groups of technical debt: in an ongoing fashion. This way, debt isn’t forgotten for bigger and better projects and phases, thus avoiding long-term interest.

To alleviate technical debt, using Agile methodologies like test automation and Continuous integration (CI) can ensure that your IT team is always working on technical debt without. Weekly sprints will also allow Agile teams to clean up data and outdated assumptions. By implementing these theories in the long run, many believe that technical debt can be avoided entirely.

**The Way Forward: Reducing Technical Debt in New Initiatives**

The best way to reduce technical debt in new projects is to include technical debt in the conversation early on. You will be able to account for the impact of short term decisions on long-term ROI and create a plan for paying off debt at the beginning of a project. Remember, just like financial debt, technical debt gets more expensive the longer you hold it. The idea is to minimize technical debt so your team can reduce risk, improve agility and deliver best-in class results in today’s competitive digital economy.