TECHNICAL DEBT: THE ULTIMATE 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 deliverable or deadline.

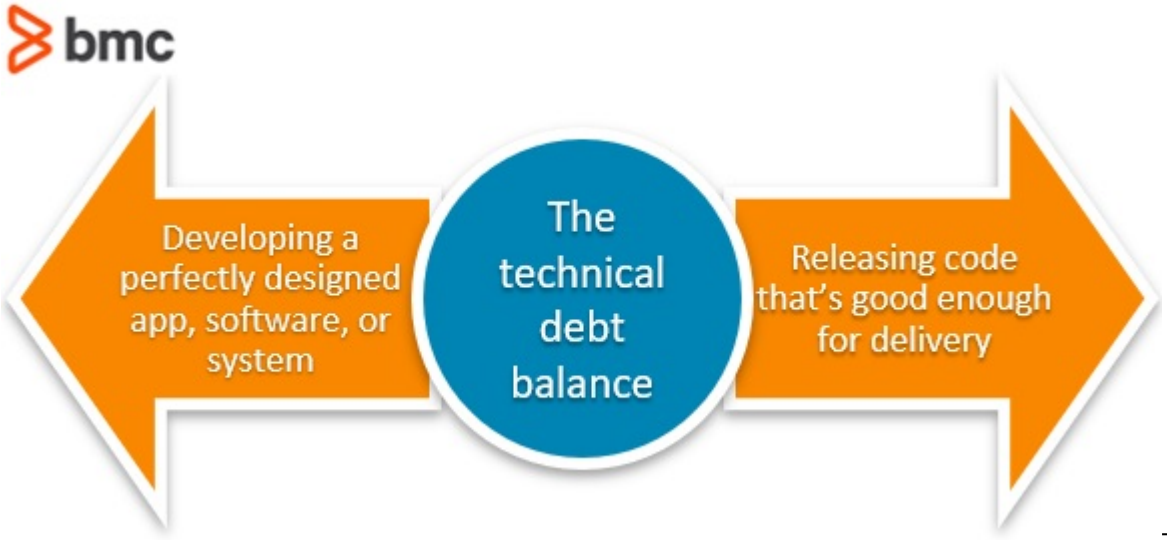
Technical debt is the coding you must do tomorrow because you took a shortcut in order to deliver the software today.

Take a simple Excel macro worksheet (.xlsm) update using Visual Basic (VB) script, for example. New VB code runs beautifully under Excel 2013 but hangs when running under Excel 365. Rather than reconfiguring the macro to run under both Excel 2013 *and* Excel 365, you put the code into production for Excel 2013 only, in order to deliver the enhancement without delay. The service desk starts rolling out Office 365 and—surprise—the macro no longer works. This piece of technical debt just came due: someone must reconfigure the VB code to work correctly under Excel 365.

In this article, we'll take a deep dive into technical debt, exploring all these topics:

Tech debt: when good enough sinks perfect

By nature, developers always have to balance this issue: developing a perfectly designed app, software, system, etc., versus putting out code that is good enough. Clean, well-designed code allows future iterations and innovations to be more easily implemented. However, in a business setting, deadlines and limited resources often prevent developers from producing clean, perfect code before delivering a product.



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

Defining technical debt and cruft

The technical debt concept is that the "debt" represents the extra development work that arises when mediocre code is implemented in the short run, despite it not being the best quality overall solution. The mediocre code—which is usually redundant, poorly designed, useless, or all three—is called **cruft**.

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 today—here he is explaining technical debt:

What is the exact definition of technical debt? The Software Engineering Institute at Carnegie Mellon University notes that technical debt "conceptualizes the tradeoff between the short-term benefit of rapid delivery and long-term value." Another perspective comes from industry analyst Gartner, defining technical debt as the deviation of an application from any nonfunctional requirements.

Here is a general example: Imagine that you are a software developer and you have to add a new functionality to a software or system. There are 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, will take 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, the more software entropy can occur.

Technical debt is like cleaning by hiding everything under the couch—it is tidy for now, but nothing is in its correct spot. You will eventually have to spend time properly cleaning and organizing it down the road in order to find something you are 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 that impact.

Technicalities of technical debt

Technical debt is not all-encompassing. An important distinction is that technical debt does not include any features or functions you intentionally didn't build. (Perhaps, for instance, a feature was initially scoped but later scrapped as deadlines changed.)

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 code refactoring. Refactoring is the restructuring of existing code as part of the development process, without changing the code's external behavior (e.g., refactoring a poorly written piece of code that takes any number and multiplies it by two will still output 2 x 2 = 4, even after the underlying code has been rewritten).

The two main reasons a developer will refactor are to:

- Address poorly written legacy code
- Evolve solutions as a problem is better understood

Good vs bad: Reasons for technical debt

Just like financial debt, there can be good reasons for technical debt—but it is important to know that going in, so debt does not 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 will 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 is not 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 will 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 how much technical debt to incur.

Types of technical debt

As we saw above, technical debt is a normal result of software development—some of it occurs for good reason. And some types of technical debt are worse than others:

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
- Inexperience with new coding techniques
- Rollout issues

For example, a design approach that ends up containing many errors is unintentional technical debt. This type sometimes occurs as the direct result of poor communication within the organization or misalignment between the developers and operations teams.

Unavoidable technical debt

Unavoidable debt 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.

Software entropy

Also known as bit-rot, software entropy occurs over time as the software quality slowly deteriorates, leading to problems with usability, errors, or necessary updates. Entropy occurs when many developers—many of whom may not understand the intended function and design—make incremental changes that increase complexity, violate NFR requirements, or slowly break the code.

The solution to software entropy is refactoring.

Identifying technical debt

Here 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**, which can handled by developing a coding style guide and sticking to it.
- **Non-functional requirement issues** where the code violates an NFR restraint. Examples include slow performance, security hacks, unreliable code results, increasingly difficult to use, loss of compatibility with other software or hardware.

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
- Reduced agility
- Negative customer experience
- Poor security

Accounting for Nonfunctional Requirements (NFRs)

Using the Gartner definition of technical debt necessitates a deep dive into nonfunctional requirements (NFRs). Understanding these nonfunctional requirements is important to identifying technical debt. Every system has NFRs, though they often aren't defined or tracked. NFRs refer to constraints on a software system and include the following seven attributes:

- Compatibility
- Security
- Reliability

- Usability
- Efficiency
- Maintainability
- Portability

There are also important sub-characteristics of NFRs:

- **Explicit requirements** are predefined by key stakeholders, such as a detailed design specification, for example.
- **Implicit requirements** are those that are expected but which have not been stated unequivocally. An example is ensuring application ease of use.

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.

Avoiding (too much) 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 time-to-market versus time working overtime to pay down interest. Or, it can show up 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 and dealing with your cruft is essential. Here is a six-step process for removing cruft and reducing technical debt.

1. **Inventory your cruft**. Start a list of technical debts. Include all instances where the developers know the code is not as clean as it should or needs to be for future development.
2. **Categorize the cruft**. List and group deferred tasks into workable units.
3. **Appraise the cruft**. Note the consequences of ignoring each unit.
4. **Air your dirty cruft**. Keep the list visible to all.
5. **Make space for cruft removal**. Inform stakeholders that rely on delivery releases (marketing, sales, etc.) that you are working on technical debt, so that each new release cannot include only new features.
6. **Remove cruft early and often**. Schedule regular and frequent time to pay off technical debt.

Managing technical debt: Best practices

Here are some ways to manage technical debt.

Assessing debt

When identifying technical debt, you might 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 measure this 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 have 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.

Communicating debt

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.

Paying off debt

There are three options to consider in terms of paying off technical debt:

1. **Waive the requirement altogether**. In other words, the organization decides to live with the system as it is and no longer deems the requirement necessary. (If you cannot waive the requirement, you're stuck with two other options…)
2. **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.
3. **Replace the application**. While this will introduce new technical debt, the idea is to address it quickly and minimize it as much possible.

Minimizing technical debt with Agile practices

Agile practitioners know that "done" is a relative term. In traditional development environments, software is done when it is 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: continuously. This way, debt is not 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. 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.

Reducing 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. By minimizing technical debt, your team can reduce risk, improve agility, and deliver best-in class results.

Additional resources

For more on software development, explore these resources:

- The BMC DevOps Blog
- The BMC Business of IT Blog
- The 12-Factor App Methodology Explained
- What Is Pair Programming?
- Resilience Engineering: An Introduction
- What is 'Dark Debt'? How to Eliminate Dark Debt