Technology Spotlight: Technical Debt

In software development, the term “technical debt" (according to The Software Engineering Institute at Carnegie Mellon University) is a concept that “conceptualizes the tradeoff between the short-term benefit of rapid delivery and long-term value." Technical debt is a consequence of the balance between developing software quickly versus developing code that conforms to all best practices. The phrase was coined in the 1990s and remained largely a developer-centric concept, but interest in technical debt and its impacts has become important for even non-developers to understand. Although it may have negative connotations, not all debt is bad debt. Technical debt can be incurred deliberately or inadvertently, and if it accumulates, it can negatively affect your ability to operate and maintain your code. And just like any other kind of debt, you can't make it go away by ignoring it—you need a plan to handle it.

In the Agile software delivery model, which emphasizes delivering the minimum viable product (MVP) as quickly as possible, development teams often deliberately incur technical debt. A simple example is a drop-down list with hard-coded values for the MVP release; the technical debt incurred is the work it will take to go back in later iterations and make that list editable. In this example, the trade-off is a relatively simple decision, as it is unlikely that architectural changes will be required to support the future change. The decisions may not always be so obvious. Which is why having a plan matters! The best way to manage technical debt is to track it the same way you track all other development work. When a trade-off decision is made, the related technical debt should be added to your Agile work tracking system (such as Visual Studio Team Services or Jira) and then managed, prioritized, estimated and planned just like any other work item or feature.

Unintentional technical debt is incurred when the organization attempts to build an MVP on a weak architecture. This is usually the result of inadequate or incomplete design. In this situation, the MVP may still be successful, but the product will reach a point where adding new features will require a significant investment in re-factoring the architecture. This is the “Goldilocks" of software design—you want an architecture that is sufficient to support future growth, but not overdesigned and difficult to work with. The best way to achieve this is to ensure you thoroughly understand the business domain before you start designing. The better you understand the end user's needs, the less likely you will be surprised by “unexpected" requirements.

Occasionally, you may hear “technical debt" used to describe vast quantities of legacy code, or ill-conceived or poorly written code. This is a misnomer. Technical debt is not an excuse for sloppy code, it refers to code and design that is sound, but may require some level of re-factoring to meet the ongoing needs of the project. Code that is buggy, poorly structured, or poorly written is not technical debt—there is an appropriately different term for that: “bad code."