Technical debt

Everyone does their best to write excellent code from scratch. There probably isn't a programmer out there who intentionally writes unclean code to the detriment of the project. But at what point does clean code become unclean?

The metaphor of "technical debt" in regards to unclean code was originally suggested by Ward Cunningham.

If you get a loan from a bank, this allows you to make purchases faster. You pay extra for expediting the process - you don't just pay off the principal, but also the additional interest on the loan. Needless to say, you can even rack up so much interest that the amount of interest exceeds your total income, making full repayment impossible.

The same thing can happen with code. You can temporarily speed up without writing tests for new features, but this will gradually slow your progress every day until you eventually pay off the debt by writing tests.

Causes of technical debt

Business pressure

Sometimes business circumstances might force you to roll out features before they're completely finished. In this case, patches and kludges will appear in the code to hide the unfinished parts of the project.

Lack of understanding of the consequences of technical debt

Sometimes your employer might not understand that technical debt has "interest" insofar as it slows down the pace of development as debt accumulates. This can make it too difficult to dedicate the team's time to refactoring because management doesn't see the value of it.

Failing to combat the strict coherence of components

This is when the project resembles a monolith rather than the product of individual modules. In this case, any changes to one part of the project will affect others. Team development is made more difficult because it's difficult to isolate the work of individual members.

Lack of tests

The lack of immediate feedback encourages quick, but risky workarounds or kludges. In worst cases, these changes are implemented and deployed right into the production without any prior testing. The consequences can be catastrophic. For example, an innocent-looking hotfix might send a weird test email to thousands of customers or even worse, flush or corrupt an entire database.

Lack of documentation

This slows down the introduction of new people to the project and can grind development to a halt if key people leave the project.

Lack of interaction between team members

If the knowledge base isn't distributed throughout the company, people will end up working with an outdated understanding of processes and information about the project. This situation can be exacerbated when junior developers are incorrectly trained by their mentors.

Long-term simultaneous development in several branches

This can lead to the accumulation of technical debt, which is then increased when changes are merged. The more changes made in isolation, the greater the total technical debt.

Delayed refactoring

The project's requirements are constantly changing and at some point it may become obvious that parts of the code are obsolete, have become cumbersome, and must be redesigned to meet new requirements.

On the other hand, the project's programmers are writing new code every day that works with the obsolete parts. Therefore, the longer refactoring is delayed, the more dependent code will have to be reworked in the future.

Lack of compliance monitoring

This happens when everyone working on the project writes code as they see fit (i.e. the same way they wrote the last project).

Incompetence

This is when the developer just doesn't know how to write decent code.

READ NEXT

When to refactor →