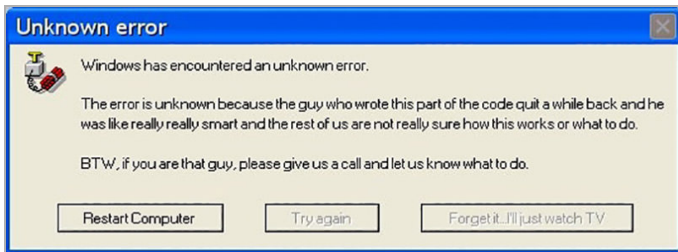
Software Quality Mortgage

by Jason Cohen on September 8, 2008

Quick software releases have long-term costs. Stakeholders and engineers alike must be prepared to repay the mortgage known as "tech debt."

Your code is a mess. Years of squeezing in "must-have" features for big customers have stretched the code beyond its original design. Core modules are riddled with landmines; we're afraid to make any changes, even with unit tests. Tacit assumptions shared by the two founders aren't obvious to the next ten hires.



When companies are new and unknown, still seeking their niche, the most important thing is to get the software out the door fast, bugs and all.

It's the right thing for the company and the right thing for the software, but there comes a day when your emphasis shifts from "time-to-market" to reducing tech support calls and not pissing off tens of thousands of existing users with a dud release.

But now you have all this crappy code.

I call this phenomenon the "Quality Mortgage." The analogy to a home mortgage is apt.

A responsible, hard-working person cannot afford to purchase a home outright and therefore enters into debt. If shippable, salable software is the house you want today, your debt is the quality and maintainability of your code. Sure you could build a close-enough-to-bug-free application if given ten years to work on it, but you're taking out a mortgage to build v1.0 in six months.

But eventually you have to pay back the debt. With interest. You pay interest in the form of bugs. Bugs everywhere, many preventable had you given time to have unit tests, good design, manual tests, and use-cases. And fixing those surface bugs doesn't fix the underlying problems in the code. This is perfectly analogous to those first years of the mortgage where you're paying interest without reducing the principal. But this is still the right choice at first—fix the most heinous bugs and keep going.

Over time you can pay back the principal, slowly. You can refactor one file while adding a feature. You can add complete unit test coverage for a handful of core methods. You can write a manual test plan for a particularly complex dialog box. This is all good! But at this rate it's still going to take ten years to pay it back.

Or maybe you'll never pay it back. Because unlike a house your software is constantly expanding with new features and reused for purposes beyond its original conception. Without fixing the underlying mess or the process that brought you that mess, you'll never catch up. It's more like an interest-only mortgage.

At some point you can't tolerate this anymore. It's time to pay down the principal in earnest. But this requires allocating time for major rework.

Winning the right to refactor can be tough politically, especially with non-technical stakeholders. Here's how to combat the common arguments against spending time refactoring:

> **Clean-up is invisible to users; we need to add new features.**
> The bugs constantly produced by messy code are visible to users too. Time spent fixing those bugs could have been spent adding features. The longer we stay in quality debt, the more time it takes to add each new feature.

We don't have time for clean-up.
You'd rather spend your time fixing bugs *generated* by the problem rather than fixing the problem? That's like whacking weeds every weekend instead of pulling them up by the roots. Prevention is sixteen times[1] more valuable than cure.

[1] "An ounce of prevention is worth a pound of cure."—Ben Franklin

Developers got themselves into this mess; they should get themselves out of it on their own time.
Had developers not gotten releases out the door as fast as they did, had they not responded so swiftly to early adopter feedback, even when the product morphed into a beast quite different from its original conception, we wouldn't have our current customers and revenue. We'd be working for another company, not complaining about the software we built.

Attention CEO's: Finger-pointing impedes resolution. Instead, challenge your developers to reduce bug reports. This is easily measured, so you can track time versus results. Remember, developers prefer implementing new features to fixing bugs, so if they're begging for time to fix bugs, it's serious.

A fine line separates debt as a lever for acceleration and an insurmountable drag. The quality mortgage is a necessary evil in early software development, despite its eventual problems. Just plan on paying it back.

P.S. After writing this I found Martin Fowler making the same point.

Printed from: *A Smart Bear*
https://longform.asmartbear.com/software-quality-mortgage/
© 2007-2024 Jason Cohen @asmartbear