

Technical Debt

Your project has it.

About me, Richard Brewster

Software Developer since 1994

Worked in telecommunications, online games, education, air quality, and mobile phone application domains.

Currently at WillowTree Apps, Inc., Charlottesville, VA

Specialize in server-side Java and relational database technology.

Used agile approaches to software development since 2002.

Debt - Whoa!

Is debt bad? A mortgage or car loan isn't bad, is it?

Neither is Technical Debt just bad.

Your project has it.

What is it?

Do you need to address it?

Maintenance

Ever get handed a project and told to 'maintain' it?

What was the biggest problem for you?

Understanding it!

How long did you have to look at code before you understood it?

How easy was it to execute the code to demonstrate and prove it out?

How confident were you that making a change would not break some part of the system unexpectedly?

Debt = Principal + Interest

Technical debt is a metaphor.

Ward Cunningham coined the term around 1992 to describe costs incurred when hurrying to get software features delivered. Time saved now has to be paid back plus interest in the future. Future increased costs will be unavoidable.

Why technical debt is not avoidable

- We want something fast.
- Our knowledge is always limited.

These are the reasons we incur technical debt. We can't really avoid it.

BUT... We can look at how we work and how it impacts the future.

AND... We can make decisions about spending to manage debt we cannot avoid incurring.

Big Design Up Front

The Waterfall model of software development is an attempt to avoid technical debt.

1. Gather all requirements (and hand off)
2. Complete the analysis (and hand off)
3. Design the software (and hand off)
4. Write code (and hand off)
5. Test (and ship)

The trouble with this is not just that it takes a lot of time. It is the assumption that knowledge is more or less complete at each cascade.

Learning Happens

Technical debt can be viewed as a lack of understanding.

- Of requirements
- Of domain analysis
- Of software design
- Of software implementation
- Of testing techniques

Knowledge naturally increases over a project lifetime.

Addressing technical debt is about finding ways to build new knowledge into the *system* quickly, and to keep it there.

Technical Debt Impact

The most important consideration:

HOW LONG IS YOUR PROJECT LIFE?

A small project with a main delivery and a few upgrades scheduled over a few months cannot incur much technical debt.

A project lasting years with frequent updates incurs huge technical debt.

Why so huge? Can't it be mitigated?

Change and Knowledge

Software change *adds knowledge* to the system.

- Domain knowledge (models, features)
- Implementation knowledge (bug fixes)

BUT: Adding knowledge is not linear in cost.

Cost of Change is Exponential

Alistair Cockburn in "Agile Software Development, The Cooperative Game, 2nd Ed." point outs a non-intuitive fact about the cost of software change.

"The cost for updating software looks like the interest growth on a financial loan: $(1 + c + u)^n$. Here, c is the necessary complexity added to the code through new business rules, u is the cost for understanding the unnecessary intertwining across the changes, and n is the number of changes put into the code." (p. 251)

Because n is in the exponent, the cost of change spirals upward over project life.

Debt Slowing Activities

So you want to address technical debt. What can you do?

Think about how knowledge enters your system during

- Requirements and analysis
- Design and implementation
- Testing

Let's look at each of these in turn.

Requirements

Anticipate and welcome changing requirements.

Look for ways of prioritizing and iterating requirements, with continual refinement of details as scheduling of implementation approaches.

How understanding propagates in a team is big topic. But the faster it gets into heads, the better. Think about how this happens on your team and how to speed it up.

My favorite author on this topic is Alistair Cockburn.

Continually Improving Design

Design can always be improved. The term 'refactoring' means to improve the design of existing code without altering its external behavior. Martin Fowler wrote a whole book on it.

Why change something that works?

- Make it easier to understand
- Make it easier to change
- Make it easier to test

There's no need to refactor a piece of code that nobody will need to understand or change or test in the future.

Automated tests support good design

An automated test is one that can be run without human intervention and automatically brings problems to the attention of humans. Its value lies in *regression*: telling you when some part of the system breaks that you were not expecting when you made a change.

Some of the categories of automated testing include:

- Unit tests for small pieces of code, such as with JUnit
- Integration tests, as of a website with Selenium
- Acceptance tests, such as with the Fit framework

Building Knowledge In

Refactoring and automated testing work together to build knowledge into the system.

This assures that when people try to understand the system in the future, one of the ways they will find effective is looking at code and at tests. It will save them time.

(They will use other ways, too, such as talking to people and reading documents.)

Measure Software with Tools

Tools exist for analyzing software complexity, looking for potential bugs, and finding code that needs to be improved. My current favorite one for Java is an open source project, Sonar.

<http://www.sonarsource.org/>

Sonar is very easy to install and use and produces a rich graphical presentation of the analysis results.

Like all code analyzers, it requires some study to learn how to use it well. Sonar even has a Technical Debt plugin.

Summary

- All projects accrue technical debt, which amounts to higher *cost of change*.
- The cost of change *spirals exponentially* over time, no matter what you do.
- But certain good practices can reduce the *rate* of debt accumulation by building knowledge into the system.
- These practices pay back their own cost and can *extend* the lifetime of a system.
- Nevertheless, projects still eventually become too expensive to change (too hard to understand).

Links

Martin Fowler on Technical Debt:

<http://martinfowler.com/bliki/TechnicalDebt.html>

Steve McConnell:

<http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx>

Rob Myers, with a link to Ward Cunningham's video:

<http://powersoftwo.agileinstitute.com/2009/03/ward-cunninghams-debt-metaphor-isnt.html>