

Understanding & Dealing with Technical Debt



GembaSystems

Find the **truth** in Software Development

Introduction

- Tim Snyder
- Developing Software (professionally) since 1986.
- Coaching Development Teams and Stakeholders since 1994.



The Problem and The Metaphor

Ward Cunningham:

“Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite.... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation.”

What does this mean to you?



How does it effect our behavior?

- Interest on technical debt reduces our capacity for new development.
- What does this mean for...
 - new requirements?
 - defects?
 - maintenance?



How do we fall into [technical] debt?

- Two factors influence our accrual of debt
 - Our aversion to “pay now” leads us to overlook that we will “pay later”.
 - The cost of change (to code) is often too high, thus reinforcing the idea of “take the short cut” or “I’ll pay later”.



Pay now? ... or pay later?

- Software Development is a learning experience.
- Refactoring goes hand-in-glove with that learning experience.
- We avoid refactoring because because it is not considered a “value-add” activity
- So we opt for the “shortest path” to new features.



The cost of change is too high!

- The cost of changing code is high, so we don't do it unless we have to.
- But why is change expensive?
 - time to design and develop
 - time to find the things we break, and
 - time to fix the things we break



How do we get out of this debt?

- First step to getting out of a hole:
 - STOP DIGGING!
- First we have to stop or slow down the accumulation of new debt
- Second we have to pay down our existing debt
 - remember, that debt incurs interest



How do we get out of this debt?

- #1 Stop or slow down accumulating new debt.
 - #2 Pay down existing debt.
-
- OK, How do we do that?
 - Confront the cause:
 - Pay now vs. Pay Later
 - High Cost of Change[ing code]



Pay Now vs. Pay Later

Hopefully this metaphor of technical debt will serve as an instrument to encourage us all, our team-mates and our stakeholders to temper our willingness to accept pay later as a viable choice when it comes to refactoring our code as we develop it.



High Cost of Change

Cost	How to minimize the cost
Time to design and develop a feature	Keep the feature's design <u>simple</u> . Use <u>pull</u> to develop only what is needed. Use <u>tests</u> as that <i>pull</i> mechanism.
Time to find the things we break	Tests are the safety net that exposes a break.
Time to fix the things we break	Again, keep the design simple and use tests to assert what the code <i>must</i> do.



High Cost of Change

see a theme?

Cost	How to minimize the cost
Time to design and develop a feature	Keep the feature's design <u>simple</u> . Use <u>pull</u> to develop only what is needed. Use <u>tests</u> as that <i>pull</i> mechanism.
Time to find the things we break	<u>Tests</u> are the safety net that exposes a break.
Time to fix the things we break	Again, keep the design simple and use <u>tests</u> to assert what the code <i>must</i> do.

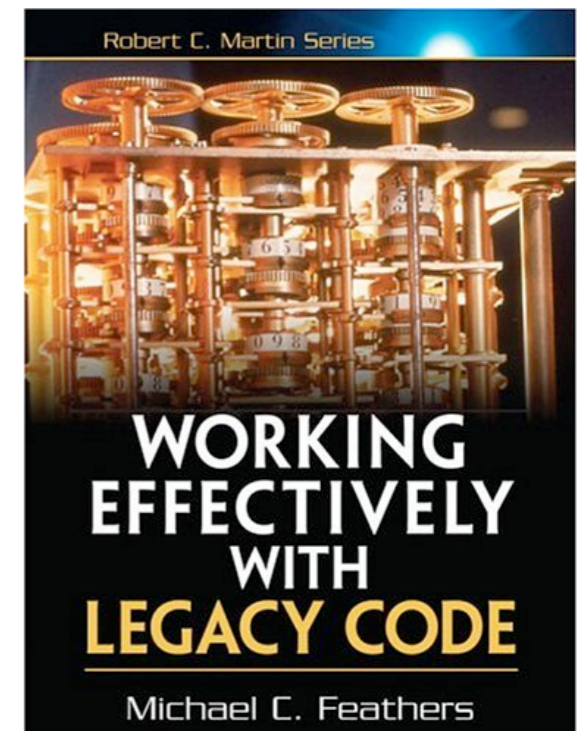


Incrementally adding tests to legacy

The opportunity to add tests to legacy code is at the point when we need to make a change to the code.

Some techniques for incrementally adding tests to legacy code:

- Characterization Tests
- Seams
- Sprouting



Get the code under test

The premise from this point forward is that if we have to change some part of our code, first we get that code under test.

Keep in mind, our intention is to discontinue the practice of adding legacy code; not necessarily to invent tests that verify old requirements.



Characterization Tests

- Can't figure out what the code is *supposed* to do?
- That's okay.
- Test what the code *actually* does and put an assertion in place.
 1. Write a failing test.
 2. Inspect what it returns (debugger).
 3. Modify your tests to pass.



Characterization Tests

- Characterization tests record the actual behavior of legacy code.
- Not all of the behavior. Just enough to ‘characterize’ or understand it.
- This is valuable.
- Make several to test boundary conditions, exception-handling, etc.



Seams

- A place where you can alter behavior without editing in that place.
 - Object seam (polymorphism)
 - Link seam (alternate lib or alternate class path)
 - Preprocessor seam (macros to replace functions)
 - Aspect seam (injected pre/post/redirect code)

- At seams, fake alternatives can be introduced to enable test.



Sprouting

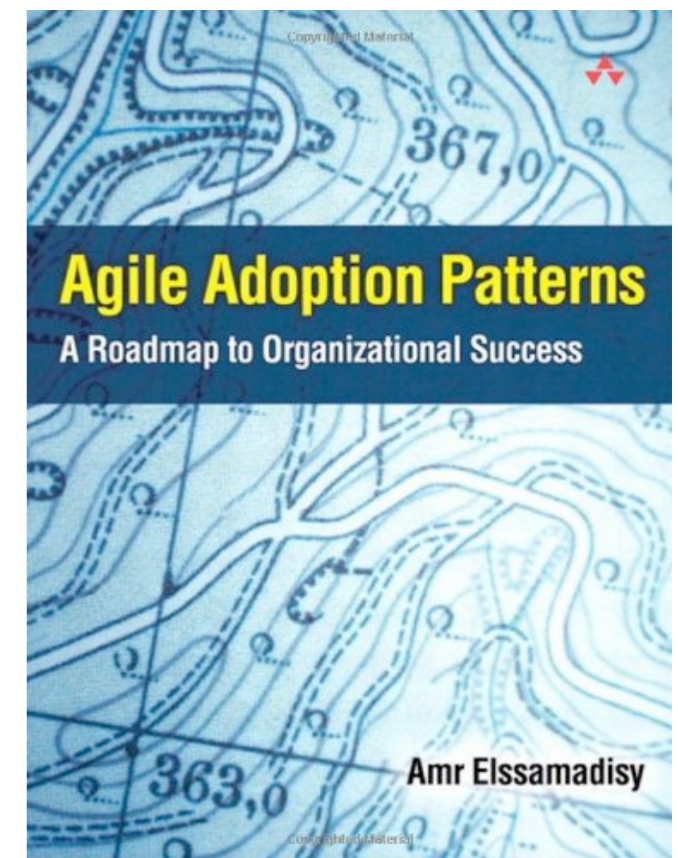
- Sprout a Method (or Class)
- Problem: Need to add code to a method but there are no tests to protect the method from faulty changes.
- Instead of changing the method, sprout a new method (or class) and call it from the old method.
- Use TDD to develop the new method.



Adoption Strategies

from Agile Adoption Patterns

- Commit as a team to the discipline of writing tests. Stop digging.
- Realize that this is first and foremost a human issue and not a tool issue.
- Agree that tests are just as important as production code
- Agree to be patient. It may take anywhere between two to six months for this practice to become habit and for the benefits to become obvious.
- Find an easy to use tool. (NOT test generation.)
- Treat your test code as first-class code.
- Get as much help as you can, this is a tough practice.
 - Bring in experienced consultants.
 - Enlist the help of others at the company with experience.
 - Buy several copies of books on TDD and Unit testing and adopting Agile books. Encourage development team to read and discuss. Maybe even start a regular reading circle.
 - Get involved with online communities and local user groups.
- Adopt collective code ownership to support team development.
- Pair program while learning and to make the discipline easier.
- Start writing tests NOW. Expect a slowdown of up-to 75% (with legacy code). Over time your time will go down to anywhere between 20%-40% of development effort



Summary

- Development is a learning experience.
- Refactoring is part of the cycle. Embrace it.
- Tests enable you to refactor fearlessly.
 - Tests can be added to Legacy code incrementally
 - Characterization Tests
 - Sprouting
 - Seams
- Refactoring reduces technical debt.



I hope you've enjoyed.
thanks for inviting me.

