Technical debt
The awful truth and what to do about it
Agile 2016

Tom Grant
Senior consultant, Net Objectives

Declan Whelan
Agile coach, LeanIntuit
Technical debt is the temptation built into software innovation.
The good news

- Software development encounters fewer constraints than other forms of invention
  - Lower material costs
  - Lower testing costs
  - Fewer regulations
  - Instantaneous rapid market delivery
  - New capabilities and maintenance sliced as thinly as we want
  - Almost no barrier for market entry
The bad news

• Software development encounters fewer constraints than other forms of invention
  • It’s easy to cut corners
  • The incremental cost of a cut corner is always low
  • We can always fill up our schedules with more to do
  • If we don’t, someone else usually pressures us to produce more
  • It’s hard to go back and do the work the right way
  • Cut corners impose a rising cost on innovation
We’re tempted to cut corners

• And we eventually pay for our sins
What is technical debt?
A textbook definition

• What is technical debt?
  • The increased difficulty in writing new code, or maintaining code, that is the natural result of...
    • Shortcuts
    • Bad coding practices
    • Hacks
    • Other times when you wish, in hindsight, you had coded more carefully
A practical definition

• The feeling that, over time, writing or maintaining code is like running through mud.
I'M GETTING TONS OF CODING DONE! WHEEEEEE!

Changing the code takes sooo loooong...

You now

Future you
“Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with refactoring. The danger occurs when the debt is not repaid. Every minute spent on code that is not quite right for the programming task of the moment counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unfactored implementation, object-oriented or otherwise.”

Ward Cunningham
how do you create it?

• Very easily, and here are a few common examples
  • “I didn’t have the time to write a simpler class…”
  • “No time to re-think the design of the class, just keep adding new stuff to it!”
  • No time to think about how someone else will fix or extend the code
  • No code review leads to culture of sloppiness
  • Didn’t think of the impact of generic error handling
What TD is not

<table>
<thead>
<tr>
<th>QUALITY DEBT</th>
<th>Unidentified defects, unknown defect trends or levels, lack of clarity about quality targets.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCESS DEBT</td>
<td>Poor processes within the team, value stream, and larger organization.</td>
</tr>
<tr>
<td>FEATURE DEBT</td>
<td>Missing capabilities in the code that lower its value.</td>
</tr>
<tr>
<td>UX DEBT</td>
<td>A poor user experience, lowering the code’s value.</td>
</tr>
<tr>
<td>SKILL DEBT</td>
<td>Missing or weak skills within the team.</td>
</tr>
</tbody>
</table>

TD may contribute to these problems. These problems may contribute to TD.
What is the impact of technical debt?
On the individual

- Harder to add new software value
- Harder to fix problems
- Harder to work in any random part of the code
- Harder to get motivated about working in the code
- Is this the career I wanted?
On the team

• Lower velocity
• Greater variance in velocity
• Harder to break down stories
• Less fluidity in task assignment
• Less flow within the team
• Harder to make reliable plans
• Lower team morale, higher turnover
On the organization

- Reduced value of software assets
- Harder to manage the portfolio of these assets
- Harder to change team or team membership
- Reduced flow in the software value stream
- Slower and less reliable responsiveness to critical problems
- Greater friction between team and other groups
What makes it hard to control technical debt?
Technical debt gets out of control quickly

• Deadline pressures and uncontrolled demand
  • Creates incentives for cutting corners
  • Don’t think through decisions
Technical debt gets out of control quickly

- Unmeasured impact
- Hard for non-technical people to grasp importance
- If no time to avoid technical debt, there’s no time to assess it
- Often lack data, especially about deep patterns
Technical debt gets out of control quickly

• Lack of experience
  • Not aware of the level of risk
• Not aware of the sources of risk
Technical debt gets out of control quickly

- No awareness among customers and executives
  - Don’t know how they contribute
  - Easier to blame the people that it is to address the problem
What's really bad about TD

• Chances are, someone else left it for you
What can we do about technical debt?
Exercise

- Dice Of Debt game
  - Get into groups
  - Play the game (20 minutes)
  - Please leave us the score sheet at the end
  - Please take the chart with you
Discussion

• Based on the game, what is the best strategy for dealing with technical debt?
• Is the strategy you used in the game at all similar to the strategy you follow in real life?
• If not, why not?
You can’t manage what you don’t measure
How do I measure it?

• Individual sources
  • What you get from code analysis tools
  • Examples
    • Cyclomatic complexity
    • Copied code

• Aggregate effects
  • Affect on the team and the organization
  • Examples
    • Velocity
    • Reliability
    • Innovation
Find the measure that works for your team

Approaches like the SQALE model help assess amount and impact
The OODA Loop – John Boyd

[Diagram of the OODA Loop]

- Observe
  - Implicit Guidance & Control
  - Unfolding Circumstances
  - Outside Information
  - Unfolding Interaction With Environment
  - Observations
  - Feedback

- Orient
  - Cultural Traditions
  - Genetic Heritage
  - Analysis & Synthesis
  - New Information
  - Previous Experiences
  - Feedback

- Decide
  - Implicit Guidance & Control
  - Decision (Hypothesis)
  - Feedback

- Act
  - Action (Test)
  - Feedback
  - Unfolding Interaction With Environment

https://en.wikipedia.org/wiki/OODA_loop
### Address the individual sources: the code

<table>
<thead>
<tr>
<th>Name</th>
<th>Good practice</th>
<th>Affects</th>
<th>Rationale</th>
<th>Remediation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tested-CCOV</td>
<td>A file has an acceptable level of code coverage.</td>
<td>Reliability</td>
<td>Unit tests verify that the code performs as expected without errors.</td>
<td>Write tests in order to cover uncovered lines. Test variable values.</td>
</tr>
<tr>
<td>Clear-INVL</td>
<td>A &quot;for&quot; loop iterator is not modified in the body of the loop.</td>
<td>Reliability</td>
<td>Modifying the loop iterator inside the loop may lead to unreliable behavior. Code is also more difficult to understand.</td>
<td>Restructure the code.</td>
</tr>
<tr>
<td>Clear-DEST</td>
<td>All &quot;if&quot;/&quot;for&quot;/&quot;while&quot; structures are delimited by curly braces.</td>
<td>Changeability</td>
<td>Using curly braces for control structures helps to better understand the code.</td>
<td>Enclose the core of the structure with curly braces.</td>
</tr>
<tr>
<td>Clear-CLDO</td>
<td>Public classes and public methods are documented.</td>
<td>Maintainability</td>
<td>Code is easier to understand</td>
<td>Identify public classes and public methods without documentation. Write additional meaningful comments.</td>
</tr>
</tbody>
</table>
Address the individual sources: the team

• Many project management-level measures, such as…
  • Follow the Boy Scout Rule
  • Make code review part of the done criteria
  • Dedicate some percentage of each sprint to TD reduction and prevention
  • Do static code analysis at some interval
  • Dedicate slack to refactoring
Address the individual sources: the team

• Establish a clear working agreement within the team
  • Which of these measures will you adopt?
  • How will you adopt them?
  • EX: How often do we do code analysis?
Make TD impact part of planning

- Portfolio
  - What investments are we willing to make?
  - How far are we willing to pursue them?
- Project/product
  - What effect is TD having on…
    - Release planning?
    - Sprint planning?
    - Re-planning?
  - What are the rules for deliberately assuming TD?
    - Risks of taking on TD
    - Risks of not taking on TD
  - How is TD prevention and reduction built into our plans?
Apply Systems Thinking

• What factors contribute to TD in your system?
  • Hiring, project deadlines etc.
  • Consider system archetypes
    e.g. Shifting the Burden

• Use general advice for system archetypes
  • e.g. if a quick fix is necessary then plan for fundamental fix as well

![Diagram of Shifting the Burden](image)

- Quick Fix
- Change
- Delay

- Technical Debt
- Fundamental Fix
- Technical Debt
- Fundamental Fix

$t$
Make the case for fixing technical debt

• Play out the options
  • What will the impact be on team productivity in 2 years?
  • How will TD affect our ability to make portfolio-level adjustments?
  • How will TD affect our ability to work with outsourcing partners?
  • How will TD affect our corporate strategy?
The conclusion should be obvious

Short-term investment in technical debt reduction and prevention allows for increased productivity in the future.

Technical debt just keeps accumulating. Our productivity plummets.
What will you do next Monday?

- Take a minute to write down what you will do next Monday to tame technical debt
- Pick a partner
- Discuss your plans with that person
The Agile Alliance can help!

<table>
<thead>
<tr>
<th>GENERAL INFO</th>
<th>White paper</th>
<th>Understand the nature of technical debt</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>A2ADAM spreadsheet</td>
<td>Assess the sources and burden of technical debt</td>
</tr>
<tr>
<td>TEAM</td>
<td>Project management white paper</td>
<td>Take steps proven to be effective</td>
</tr>
<tr>
<td>JUSTIFICATION</td>
<td>Dice Of Debt game</td>
<td>Educate people about the importance of dealing with technical debt</td>
</tr>
</tbody>
</table>

https://www.agilealliance.org/resources/initiatives/technical-debt/
QUESTIONS?