Strategic management of Technical Debt

Philippe Kruchten

April 4th, 2017

ICSA 2017, Göteborg
Philippe Kruchten, Ph.D., P.Eng., CSDP

Professor of Software Engineering
NSERC Chair in Design Engineering
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, BC Canada
pbk@ece.ubc.ca

Founder and president
Kruchten Engineering Services Ltd
Vancouver, BC Canada
philippe@kruchten.com @pbpk
Outline

• What is technical debt?
• The technical debt landscape
• Limits of the metaphor
• Managing technical debt
• Tools and techniques
• Friction in software development
• Practical steps
Technical Debt

• Concept introduced by Ward Cunningham
• Often mentioned, rarely studied
• All experienced software developers “feel” it.
• Drags long-lived projects and products down
Origin of the metaphor

- Ward Cunningham, at OOPSLA 1992

“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, object-oriented or otherwise.”

Cunningham, OOPSLA 1992
Technical Debt (S. McConnell)

- Implemented features (visible and invisible) = assets = non-debt
- Type 1: unintentional, non-strategic; poor design decisions, poor coding
- Type 2: intentional and strategic: optimize for the present, not for the future.
  - 2.A short-term: paid off quickly (refactorings, etc.)
    - Large chunks: easy to track
    - Many small bits: cannot track
  - 2.B long-term

McConnell 2007
Technical Debt (M. Fowler)

Reckless

“We don’t have time for design”

Prudent

“We must ship now and deal with consequences”

Deliberate

Inadvertent

“What’s Layering?”

“Now we know how we should have done it”

Fowler 2009, 2010
Time is Money (I. Gat)

• Convert this in monetary terms:
  “Think of the amount of money the borrowed time represents – the grand total required to eliminate all issues found in the code”
Tech Debt (Jim Highsmith)

- Once on far right of curve, all choices are hard
- If nothing is done, it just gets worse
- In applications with high technical debt, estimating is nearly impossible
- Only 3 strategies
  1. Do nothing, it gets worse
  2. Replace, high cost/risk
  3. Incremental refactoring, commitment to invest

Source: Highsmith, 2009
Technical Debt (S. McConnell)

• TD: A design or construction approach that is expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now

McConnell 2011
Technical Debt Definition

In software-intensive systems, technical debt is the collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible.

Technical debt presents an actual or contingent liability that impacts internal system qualities, primarily maintainability and evolvability.

Example

First, more capabilities

Then, more infrastructure

First more infrastructure

Then, more capabilities

need to monitor technical debt to gain insight into life-cycle efficiency

underestimated re-architecting costs

neglected cost of delay to market

Ozkaya, SEI, 2010
Making Hard Choices about Technical Debt

The Hard Choices game is a simulation of the software development cycle meant to communicate the concepts of uncertainty, risk, options, and technical debt.

In the quest to become market leader, players race to release a quality product to the marketplace.

Software Engineering Institute | Carnegie Mellon

HARD CHOICES

Do you take the time to gather more tools or do you take a shortcut?
Playing the Hard Choices Board Game

• The goal of the game is to accumulate the most points.
• Players accumulate points by crossing END ahead of their competitors and collecting Tool Cards.
• The person with the most points wins.

• Market Leader Points. The first player to cross END gets 7 points, second gets 3 points, and third gets 1 point. The last player remaining on the board gets no points.
• Tool Points. You get 1 point for each Tool Card.
Rules of Play

• Start of Play
  – Roll the die to determine who goes first. Play proceeds clockwise.

• Player Movement
  – During a turn, roll the die: Move your piece the number of spaces indicated on the die minus the number of penalties incurred, determined by the number of your Bridge Cards.
  – You may move in either direction, or in both directions, within a turn. This increases your opportunity to land on a Tool Square.
  – Once a player has reached the end, no one can move backwards.

• End of Play
  – To enter the END cell you may roll anything equal or greater than the number of remaining squares.
  – The game ends when one player remains on the board.
Special Squares

• Hard Choices Squares
  – When crossing a Hard Choices Square, you must decide whether to go over the Shortcut Bridge or to go the long way and try to collect more Tool Cards.

• Bridges and Bridge Cards
  – Bridges count as one movement.
  – When crossing a Shortcut Bridge, you must collect a Bridge Card. Each Bridge Card subtracts one from subsequent rolls of the die.
  – You may get rid of a Bridge Card by skipping a turn anytime during the game.

• Tool Squares and Tool Cards
  – When landing on a Tool Square, you may elect to take a Tool Card.
  – You may only collect one Tool Card for a given square.
  – If you have a Tool Card, you may elect not to take another. Instead you may play the card (returning it to the deck) and get a free turn.
Debrief After the Game

• What just happened?
  – How did you experience the game?
  – What strategies did you employ?

• So what?
  – How do your experiences in the game relate to the strategies you employ during software development in the face of uncertainty?
  – How does this relate to the choices you make—in investing effort to gain an advantage or paying a price to take shortcuts?
  – What are their implications?

• Now what?
  – What will you take away from the experience?
  – What might you do differently as a result?
Outline

• What is technical debt?
• **The technical debt landscape**
• Limits of the metaphor
• Managing technical debt
• Tools and techniques
• Friction in software development
• Further research on technical debt
New features
Additional functionality

Technological gap

Architecture
Architectural debt
Structural debt
Test debt
Documentation debt

Low internal quality
Code complexity
Code smells
Coding style violations

Visible

Mostly invisible

Defects
Low external quality

Visible

Evolution issues: evolvability

Quality issues: maintainability

Zürich, May 2012
- Architectural debt
- Structural debt
- Test debt
- Documentation debt
- Code complexity
- Code smells
- Coding style violations
- Low internal quality
- Low external quality
- Defects

Issues: evolvability

Quality issues: maintainability
TD: negative value, invisible

- Visible
  - New features
  - Architectural, Structural features

- Invisible
  - Technical Debt

- Positive Value
  - Added functionality

- Negative Value
  - Defects
Interests

• In presence of technical debt, cost of adding new features is higher; velocity is lower.

• When repaying (fixing), additional cost for retrofitting already implemented features

• Technical debt not repaid => lead to increased cost, forever

• Cost of fixing (repaying) increases over time

M. Fowler, 2009
Where the metaphor breaks

• Initial investment at T0 in an environment E0. Now in T2, E has changed to E2, a mismatch, has occurred, which creates a debt.
  – The debt is created by the change of environment. The right decision in the right environment at some time may lead to technical debt.

• Prudent, inadvertent
Where the metaphor breaks...

- Technical debt depends on the future
- Technical debt cannot be measured
- You can walk away from technical debt
- Technical debt should not be completely eliminated
- Technical debt cannot be handled in isolation
- Technical debt can be a wise investment
Technical Debt (1)
Technical Debt (2)
Technical Debt (3)
Potential vs. actual debt

• Potential debt
  – Type 1: OK to do with tools (see Gat & co. approach)
  – Type 2: structural, architectural, or technological gap: Much harder

• Actual debt
  – When you know the way forward
TD litmus test

- If you are not incurring any interest, then it probably is not a debt

McConnell 2013
Technical debt as an investment?
TD and Real Options

Market loves it
+ $4M

Market hates it
+ $1M

\[ \text{NPV (} P_1 \text{)} = -2M + 0.5 \times 4M + 0.5 \times 1M = 0.5M \]

Source: K. Sullivan, 2010
at TD Workshop SEI 6/2-3
TD and Real Options (2)

\[
\text{NPV (P}_2\text{)} = -1M + 0.5 \times 3M + 0.5 \times 1M = 1M
\]

Taking Technical Debt has increased system value.

Source: K. Sullivan, 2010
TD and Real Options (3)

NPV ($P_3$) = $-1M + 0.67 \times 2.5M + 0.33 \times 1M = 1M$

More realistically:
Debt + interest
High chances of success
TD and Real Options (3)

NPV ($P_3$) = $-1M + 0.67 \times 2.5M + 0.33 \times 1M = 1M$

More realistically:
Debt + interest
High chances of success
Not debt really, but options with different values...
Do we want to invest in architecture, in test, etc...

Source: K. Sullivan, 2010
Outline

• What is technical debt?
• The technical debt landscape
• Limits of the metaphor
• Managing technical debt
• Tools and techniques
• Friction in software development
• Further research on technical debt
How do people “tackle” technical debt
Tackling Technical Debt

Attitude, approaches found:
1. Ignorance is bliss
2. The elephant in the room
3. Big scary $$$$$ numbers
4. Five star ranking
5. Constant reduction
6. We’re agile, so we are immune!
Ignorance is bliss

You’re just slower, and slower, but you do not know it, or do not know why

Velocity

accumulated technical debt impacts ability to deliver
The elephant in the room

- Many in the org. know about technical tech.
- Indifference: it’s someone else’s problem
- Organization broken down in small silos
- No real whole product mentality
- Short-term focus
Big scary $$$$$ numbers

- Code smells: 167 person days
- Missing test: 298 person days
- Design: 670 person days
- Documentation: 67 person days

**Totals**

- Work: 1,202 person x days
- Cost: $577,000
Static analysis + Consulting

• Cutter Consortium: Gat, et al.
  – Use of Sonar, etc.
  – Focused on code analysis
  – TD = total value of fixing the code base

• CAST software

• ThoughtWorks

Debt analysis engagements
Debt reduction engagements
Issues

• Fits the metaphor, indeed.
• Looks very objective... but...
• Subjective in:
  – What is counted
  – What tool to use
  – Cost to fix

Not all fixes have the same resulting value.
Sunk cost are irrelevant, look into the future only.
What does it mean to be “Debt free”??
Five star ranking

- Define some *maintainability* index
- Benchmark relative to other software in the same category
- Re-assess regularly (e.g., weekly)
- Look at trends, correlate changes with recent changes in code base

- SIG (Software Improvement Group), Amsterdam
- Powerful tool behind
Constant debt reduction

• Make technical debt a visible item on the backlog
• Make it visible outside of the software dev. organization
• Incorporate debt reduction as a regular activity
• Use buffer in longer term planning for yet unidentified technical debt
• Lie (?)
Buffer for debt repayment
A later release
We are agile, so we’re immune!

In some cases we are agile and therefore we run faster into technical debt
Agile mottos

• “Defer decision to the last responsible moment”
• “YAGNI” = You Ain’t Gonna Need It
  – But when you do, it is technical debt
  – Technical debt often is the accumulation of too many YAGNI decisions
• “We’ll refactor this later”
• “Deliver value, early”
• Again the tension between the yellow stuff and the green stuff
• You’re still agile because you aren’t slowed down by TD yet.
Story of a failure

- Large re-engineering of a complex distributed world-wide system; 2 millions LOC in C, C++, Cobol and VB
- Multiple sites, dozens of data repositories, hundreds of users, 24 hours operation, mission-critical ($billions)
- XP+Scrum, 1-week iterations, 30 then up to 50 developers
- Rapid progress, early success, features are demo-able
- Direct access to “customer”, etc.
- A *poster project for scalable agile development*
Hitting the wall

- After 4 ½ months, difficulties to keep with the 1-week iterations
- Refactoring takes longer than one iteration
- Scrap and rework ratio increases dramatically
- No externally visible progress anymore
- Iterations stretched to 3 weeks
- Staff turn-over increases
- Project comes to a halt
- Lots of code, no clear architecture, no obvious way forward
Gat’s Tech Debt vicious cycle

1. Take Technical Debt
2. Technical Debt Accrues
3. Fail to Pay back Technical debt
4. Reduced Development Team Velocity
5. (more) Relentless Pressure

Israel Gat, 2010
Value, Quality, Constraints

• Value = extrinsic quality
  – Metric: Net present value
• Quality = intrinsic quality
  – Metric: Technical debt
• Constraints = cost, schedule, scope
  – Metric: Cost

Highsmith 2010
Cognitive biases and TD

- Escalation of commitment
  - Aka, too much invested to quit
- Sunk cost fallacy
  - aka. throwing good money after bad
- Anchoring
- Confirmatory bias
Timeline

- **Incurred**
- **Symptom**
- **Rework**
- **Payoff**

**What is the debt?**
Technical debt item description
Risk analysis, Development state analysis

**How does debt accumulate?**
Static and architecture analysis

**When to pay back debt?**
Architecture-focused release planning
Outline

• What is technical debt?
• The technical debt landscape
• Limits of the metaphor
• Managing technical debt
• Tools and techniques
• Friction in software development
• Further research on technical debt
Tools and Techniques

Some examples
Tools for Technical Debt Analysis

- Vendors include
  - CAST
  - Inspearit
  - SonarSource (SonarQube)
  - Thoughtworks
  - Software Improvement Group (SIG)
  - Lattix
  - Hello2morrow
  - Tocéa (Scertify)
  - Xdepend
  - Klocwork
  - JetBrains
• Real Option theory
• Dependency Structure Matrix
  – Propagation cost
• Sonarqube
• SQALE
• Scertify
SQALE

• SQALE = Software Quality Assessment based on Lifecycle Expectation
• Jean-Louis Letouzey and Thierry Coq
• Inspearit
  – (previously known as Det Norske Veritas France)
SQALE

1. Quality Model
2. Analysis Models
3. Indices
4. Indicators

Source Code
Analysis tools

Findings Table
Costs Tables

Source code related requirements

« Right Code » Definition

© inspearit-2012
An **analytic view** provided by orthogonal characteristics. One understands impact of each Non-Conformity and improvement on quality characteristic and life cycle issues.

An **external view** that represents the perceived quality evaluated by consolidation of the hierarchy of characteristics.
Costs

“Right Code” Definition

© inspearit-2012
Reduce business impact
Maximize RoI
SonarQube
SonarQube and SQALE
SQALE-like dashboard with SonarQube

<table>
<thead>
<tr>
<th></th>
<th>Releasability</th>
<th>Reliability</th>
<th>Security</th>
<th>Maintainability</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABAP Sample Project</td>
<td>Passed</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>282</td>
</tr>
<tr>
<td>abilian-core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abilian-crm-core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abilian-sbe</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activiti</td>
<td>Failed</td>
<td>E</td>
<td>D</td>
<td>B</td>
<td>78k</td>
</tr>
<tr>
<td>AisLib application framework</td>
<td>Passed</td>
<td>E</td>
<td>D</td>
<td>A</td>
<td>12k</td>
</tr>
<tr>
<td>Ambrose</td>
<td>Passed</td>
<td>E</td>
<td>A</td>
<td>A</td>
<td>4.1k</td>
</tr>
<tr>
<td>Angular Dependency Injection v2</td>
<td>Passed</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>594</td>
</tr>
<tr>
<td>AngularJS</td>
<td>Failed</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td>119k</td>
</tr>
</tbody>
</table>
Structural level

• Dependency analysis
  – Propagation cost

• Interview the designers
  – TD and its causes are in their heads
### Dependency Structure Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>Strength of B’s dependency on A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Strength of A’s dependency on B</td>
<td>Strength of C’s dependency on B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Dependencies for MS-Lite

<table>
<thead>
<tr>
<th></th>
<th>Alarm Notification</th>
<th>LR Engine</th>
<th>Alarm Engine</th>
<th>Alarm Manager</th>
<th>Logon</th>
<th>Client Updates</th>
<th>Rule Processor</th>
<th>Adapter Manager</th>
<th>User Session</th>
<th>Data Access</th>
<th>Cache</th>
<th>FSS Adapter</th>
<th>Data Persistence</th>
<th>Publish Subsc</th>
</tr>
</thead>
<tbody>
<tr>
<td>US 01</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>US 02</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>US 03</td>
<td>3</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>US 04</td>
<td>4</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>US 05</td>
<td>5</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>US 06</td>
<td>6</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>US 07</td>
<td>7</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>US 08</td>
<td>8</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>US 09</td>
<td>9</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACT 14</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACT 15</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACT 16</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACT 17</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ACT 18</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACT 19</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACT 20</td>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACT 21</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dependency Structure Matrix

![Dependency Structure Matrix](image-url)
## Dependencies in Release Planning

| Dependencies between stories & supporting architectural elements | Understanding the dependencies between stories and architectural elements enables staged implementation of technical infrastructure in support of achieving stakeholder value. |
| Dependencies between architectural elements | Low-dependency architectures are a critical enabler for scaling-up agile development.¹ |
| Dependencies between stories | High-value stories may require the implementation of lower-value stories as precursors.² |

¹ Mary and Tom Poppendieck – “Leading Lean Software Development”
² Mark Denne, Jane Cleland-Huand – “Software by Numbers”
Propagation cost

• “Density” of the DSM
  – Proposed by McCormack et al. in 2006
  – Several limitations as a tool to measure T.D.

• Improved PC:
  – Boolean to continuous value (=dependency “strength”)
  – Changes not uniformly spread throughout the code
  – Less sensitive to size of code

McCormack et al. 2006
So Technical debt...

• ... it’s messy

• Cannot isolate or tokenize
  – Lots of dependencies

• Cannot assess easily
  – Cost and value dependent on future evolution

• Polymorphic
  – Good & bad, costly and beneficial, harmful and innocuous
Practical steps

From tactical (and simple) to more strategic (and sophisticated)
• Tactical
  – Short-term actions – limited scope
  – Actual means: tools, process steps, immediate plan

• Strategic
  – Long-term plan – wider scope
  – Process, management, education
  – Drive some of the tactical actions above
Practical steps (1) - Awareness

• Organize a lunch-and-learn with your team to introduce the concept of technical debt. Illustrate it with examples from your own projects, if possible.

• Create a category “TechDebt” in your issue tracking system, distinct from defects, or new features. Point at the specific artifacts involved.

• Standardize on one single form of “Fix me” or “Fix me later” comment in the source code to mark places that should be revised and improved later. They will be easier to spot with a tool.
Practical steps (2) - Identification

• Acquire and deploy in your development environment a static code analyser to detect code-level “code smells”. (Do not panic in front of the large number of positive warnings).
• After some “triage” feed them in the issue tracking system, in the tech debt category
• At each development cycle (iteration), reduce some of the technical debt by explicitly bringing some tech debt items into your iteration or sprint backlog.
Practical steps (3) - Evaluation

- For identified tech debt items, give not only estimates of the cost to “reimburse” them or refactor them (in staff effort), but also estimate of the cost to not reimburse them: how much it drags the progress now. At least describe qualitatively the impact on productivity or quality. This can be assisted by tools from your development environment, to look at code churn, and effort spent.

- Prioritize technical debt items to fix or refactor, by doing them first in the parts of your code that are the most actively modified, leaving aside or for later the parts that are never touched.
Practical Steps (4) Architectural debt

• Refine in your issue tracker the TechDebt category into 2 subcategories: simple, localized, code-level debt, and wide ranging, structural or architectural debt.

• Acquire and deploy a tool that will give you hints about structural issues in your code: dependency analysis

• Organize small 1-hour brainstorming sessions around the question: “What design decision did we make in the past that we regret now because it is costing us much?” or “If we had to do it again, what should have we done?”
  – This is not a blame game, or a whining session; just identify high level structural issues, the key design decisions from the past that have turned to technical debt today.
Practical steps (5) – Process improvements

• For your major kinds of technical debt, identify the root cause – schedule pressure, process or lack of process, people availability or turn over, knowledge or lack of knowledge, tool or lack of tool, change of strategy or objectives – and plan specific actions to address these root causes, or mitigate their effect.

• Develop an approach for systematic regression testing, so that fixing technical debt items does not run you in the risk of breaking the code.
  – Counter the “It is not really broken, so I won’t fix it.”

• If you are actively managing risks, consider bringing some major tech debt items in your list of risks.
References

References


Other sources (Talks/slides)

- McConnell, S. (October 2011) Managing technical debt. (Webinar)
Other sources

• Slides on Sonar, from Olivier Gaudin, CEO of Sonarqube

• Slides on SQALE from Jean-Louis Letouzey, Inspearit

• Slides on DSM from Ipek Ozkaya and Robert Nord, SEI
Conceptual model of Technical debt