Technical Debt
from metaphor to theory and practice

Philippe Kruchten
Helsinki, August 21st 2012

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
Outline

• What is technical debt? Several viewpoints.
• The technical debt landscape
• Structural or architectural debt
• Research on technical debt
• “Managing” technical debt
• Summary, useful pointers

Acknowledgements

• Research on TD partly funded by the
  Software Engineering Institute | Carnegie Mellon
  – Ipek Ozkaya, Rod Nord, Nanette Brown
  – They have also contributed to building this presentation over the last 2 years.

• UBC master students Erin Lim Kam-Yan and Marco Gonzalez-Rojas …
  – … with some industry partners
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

Technical Debt (M. Fowler)

<table>
<thead>
<tr>
<th>Reckless</th>
<th>Prudent</th>
</tr>
</thead>
<tbody>
<tr>
<td>“We don’t have time for design”</td>
<td>“We must ship now and deal with consequences”</td>
</tr>
<tr>
<td>Deliberate</td>
<td>Inadvertent</td>
</tr>
<tr>
<td>“What’s Layering?”</td>
<td>“Now we know how we should have done it”</td>
</tr>
</tbody>
</table>

McConnell 2007

Fowler 2009, 2010
First more capabilities
Then, more infrastructure
underestimated re-architecting costs
neglected cost of delay to market
need to monitor technical debt to gain insight into life-cycle efficiency
First more infrastructure
Then, more capabilities

Example

Technical Debt (Chris Sterling)

• Technical Debt: issues found in the code that will affect future development but not those dealing with feature completeness.

Or

• Technical Debt is the decay of component and intercomponent behaviour when the application functionality meets a minimum standard of satisfaction for the customer.
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.

Source: McConnell, 2011

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.

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”

Example: TD is the sum of...

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

**Totals**
- Work 1,202 person days
- Cost $577,000
Causes of Technical Debt

<table>
<thead>
<tr>
<th>TECHNOLOGY</th>
<th>PROCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Technology limitations</td>
<td>• Little consideration of code maintenance</td>
</tr>
<tr>
<td>• Legacy code</td>
<td>• Unclear requirements</td>
</tr>
<tr>
<td>• COTS</td>
<td>• Cutting back on process (code reviews)</td>
</tr>
<tr>
<td>• Changes in technology</td>
<td>• Little or no history of design decisions</td>
</tr>
<tr>
<td>• Project maturity</td>
<td>• Not knowing or adopting best practices</td>
</tr>
<tr>
<td>PEOPLE</td>
<td></td>
</tr>
<tr>
<td>• Postpone work until needed</td>
<td></td>
</tr>
<tr>
<td>• Making bad assumptions</td>
<td></td>
</tr>
<tr>
<td>• Inexperience</td>
<td></td>
</tr>
<tr>
<td>• Poor leadership/team dynamics</td>
<td></td>
</tr>
<tr>
<td>• No push-back against customers</td>
<td></td>
</tr>
<tr>
<td>• &quot;Superstars&quot; – egos get in the way</td>
<td></td>
</tr>
<tr>
<td>• Little knowledge transfer</td>
<td></td>
</tr>
<tr>
<td>• Know-how to safely change code</td>
<td></td>
</tr>
<tr>
<td>• Subcontractors</td>
<td></td>
</tr>
</tbody>
</table>

| PRODUCT                            |                                              |
|• Schedule and budget constraints    |                                              |
|• Poor communication between        |                                              |
| developers and management          |                                              |
|• Changing priorities (market       |                                              |
| information)                       |                                              |
|• Lack of vision, plan, strategy    |                                              |
|• Unclear goals, objectives and      |                                              |
| priorities                          |                                              |
|• Trying to make every customer     |                                              |
| happy                              |                                              |
|• Consequences of decisions not clear|                                              |

Israel Gat, 2010
Copyright © KESL 2012

Lim et al., 2012
Technical debt landscape

- Visible
  - New features
  - Additional functionality

- Mostly invisible
  - Architectural debt
  - Structural debt
  - Test debt
  - Coding style violations
  - Code complexity

- Code smells
  - Low internal quality
  - Code smells
  - Code complexity

- Defects
  - Low external quality

Evolution issues: evolvability

Quality issues: maintainability
Value of Software Architecture

A little détour

Value and cost

• Architecture has no (or little) externally visible “customer value”
• Iteration planning (backlog) is driven by “customer value”
• Ergo: architectural activities are often not given attention

• BUFD & YAGNI & Refactor!
Value and cost

- Cost of development is not identical to value
- Trying to assess value and cost in monetary terms is hard and often leads to vain arguments
- Use “points” for cost and “utils” for value
- Use simple technique(s) to evaluation cost in points and value in utils.

What’s in your backlog?

<table>
<thead>
<tr>
<th>Visible</th>
<th>Invisible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Value</td>
<td>New features, Added functionality</td>
</tr>
<tr>
<td>Negative Value</td>
<td>Architectural, Structural features, Technical Debt</td>
</tr>
</tbody>
</table>
TD: negative value, invisible

Visible
- New features
- Added functionality

Invisible
- Architectural, Structural features
- Technical Debt

Positive Value
- New features
- Added functionality

Negative Value
- Defects

Copyright © KESL 2012

Technical Debt (1)

12

$15

12

$16

12

$18

a

$5

b

$3

$20

$19

$18

Copyright © KESL 2012

Copyright © 2012 by Philippe Kruchten
Technical Debt

• Defect = Visible feature with negative value

• Technical debt = Invisible feature with negative value

• Cost .... of fixing

• Value .... of repaying technical debt, interests loss of productivity, etc.

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
Deferring implementation: Value decreases

But technical debt increases over time
Advances

Areas of further investigation

Research on TD

- Characterize objectively and quantitatively the amount of technical debt in a given system
- Taxonomy of technical debt => Better detection
- Causes of technical debt => Improved prevention
- Project management strategies to control and to cope with technical debt

- Tools and methods to deal with code smells, etc.
- Application of Real Options, Dependency Structure Matrix, or other value-based technique
• Code level debt (McConnell type 1)
• Structural debt (McConnell type 2)

---

**Code level**

• A.K.A., Code smells
• Much research, though fragmented
• Many tools to do static code analysis
• Example: Code replication (clones)
• Approach: detect + refactoring
Tech Debt = Maintainability?

• Example:
  – SIG (Software Improvement Group), Amsterdam

Debt at the Architectural Level

• Harder to detect with tools
• Less researched

• A few paths to explore:
  – Dependency structure matrices
  – Business Theories:
    • Real Option
    • Net present value
Real Options Theory

• Often mentioned, but rarely put in application in software

TD and Real Options

\[ P_1: \begin{cases} S_0 \to -2M & \quad p=0.5 \\ p=0.5 & \quad \text{Market loves it} \\ p=0.5 & \quad \text{Market hates it} \end{cases} \]

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

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

NPV \( (P_2) = -1M + 0.5 \times 3M + 0.5 \times 1M = 1M \)

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 x 2.5M + 0.33 x 1M = 1M

More realistically:
Debt + interest
High chances of success

TD and Real Options (4)

Not debt really, but options with different values...
Do we want to invest in architecture, in test, etc...

Source: K. Sullivan, 2010
Options Theory

• Often mentioned, but rarely put in application in software
• Not even scratched the surface
• Pay-off not obvious, though…
  – Too much guesswork involved to trust results,
  – Lot of work involved

Debt at the Architectural level

• Design Structure Matrix (DSM)
  – a.k.a, Dependency Structure Matrix
• Domain Mapping Matrix (DMM)

• Tools to create and manipulate DSMs and DMMs
Dependency Structure Matrix

A | B | C
---|---|---
A | Strength of B’s dependency on A | Strength of C’s dependency on B
B | Strength of A’s dependency on B | |
C | | |

Dependencies for MS-Lite

- US 01
- US 02
- US 03
- US 04
- US 05
- US 06
- US 07
- US 08
- US 09
- ACT 14
- ACT 15
- ACT 16
- ACT 17
- ACT 18
- ACT 19
- ACT 20
- ACT 21
Dependence Structure Matrix

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
Exploring other variations

• Size of components
  – Add some weighting factor related to the size of the component A and B, where A depends on B

• Nothing very useful so far; need more experimentation and validation on large real systems

Example of PC: Evolution of Ant

Technical debt reduction

Copyright © KESL 2012
DSM

• Value of DSM not fully explored yet
  – Concept of propagation cost
  – Concept of density
  – Need to integrate values and costs

• Tools to produce or manipulate DSM
  – SonarJ
  – Lattix

• “What if” scenarios

“Tackling” 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

![Chart showing velocity and 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

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.
TD: a few suggestions

- Inform
- Identify debt; name it
- Classify debt: code quality, or structural
- Assign value and cost (immediate and future)
- Make it visible (put in backlog)
- Prioritize with other backlog elements

Remember

- Technical debt is not a defect
- Technical debt is not necessarily a bad thing

<table>
<thead>
<tr>
<th>Visible</th>
<th>Invisible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Value</td>
<td>Hidden, architectural feature</td>
</tr>
<tr>
<td>Negative Value</td>
<td>Technical Debt</td>
</tr>
</tbody>
</table>

Visible Feature
Visible defect
Also...

• A suitable system architecture is not likely to spontaneously emerge out of weekly refactorings

• How much architecture do you need or have?

• Some novel projects need an
  – Architecture owner
    together with
  – Product owner, and ScrumMaster

Conclusion

• Technical debt is more a rhetorical category than a technical or ontological category.

• The concept resonates well with the development community, and sometimes also with management.

• It bridges the gap between business decision makers and technical implementers.

• It’s only a metaphor; do not push it too far.
Visible

New features

Addition functionality

Mostly invisible

Technological gap

architecture

Architectural debt

Structural debt

Low internal quality

Code smells

Code complexity

Low external quality

Test debt

Coding style violations

Coding style violations

Documentation debt

Evolution issues: evolvability

Quality issues: maintainability

Visible

Defects

Low external quality

Visible

Copyright © KESL 2012

Upcoming events

• Special issue of IEEE Software on Technical debt November 2012

• Possibly a 4th workshop on Technical Debt at ICSE 2013, in San Francisco
  – Or some other venue...
    • Saturn 2013
    • CompArch 2013

Copyright © KESL 2012

Copyright © 2012 by Philippe Kruchten

35
References

Other sources (Talks/slides)

- West, D. (January 2011), Balancing agility and technical debt, Forrester & Cast Software

Other pointers

http://techdebt.org

http://www.ontechnicaldebt.com/

@OnTechnicalDebt
Acknowledgements

• Research on TD partly funded by the Software Engineering Institute | Carnegie Mellon
  – Ipek Ozkaya, Rod Nord, Nanette Brown
  – They have also contributed to building this presentation over the last 2 years.

• UBC master students Erin Lim Kam-Yan and Marco Gonzalez-Rojas ...
  – … with some industry partners