

# Strategic management of Technical Debt

Philippe Kruchten

April 4<sup>th</sup>, 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.”



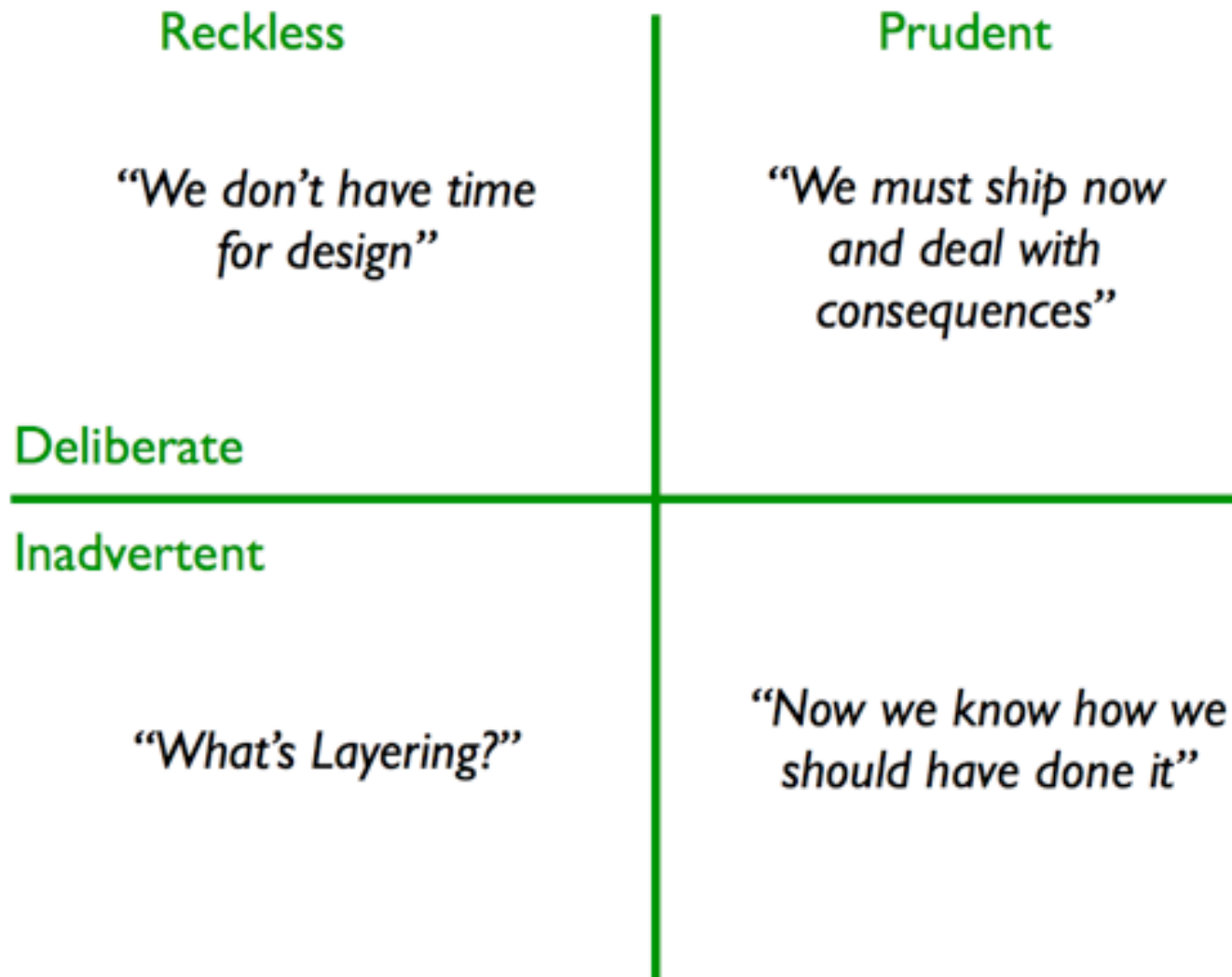
Cunningham, OOPSLA 1992

# 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



# Technical Debt (M. Fowler)



Fowler 2009, 2010

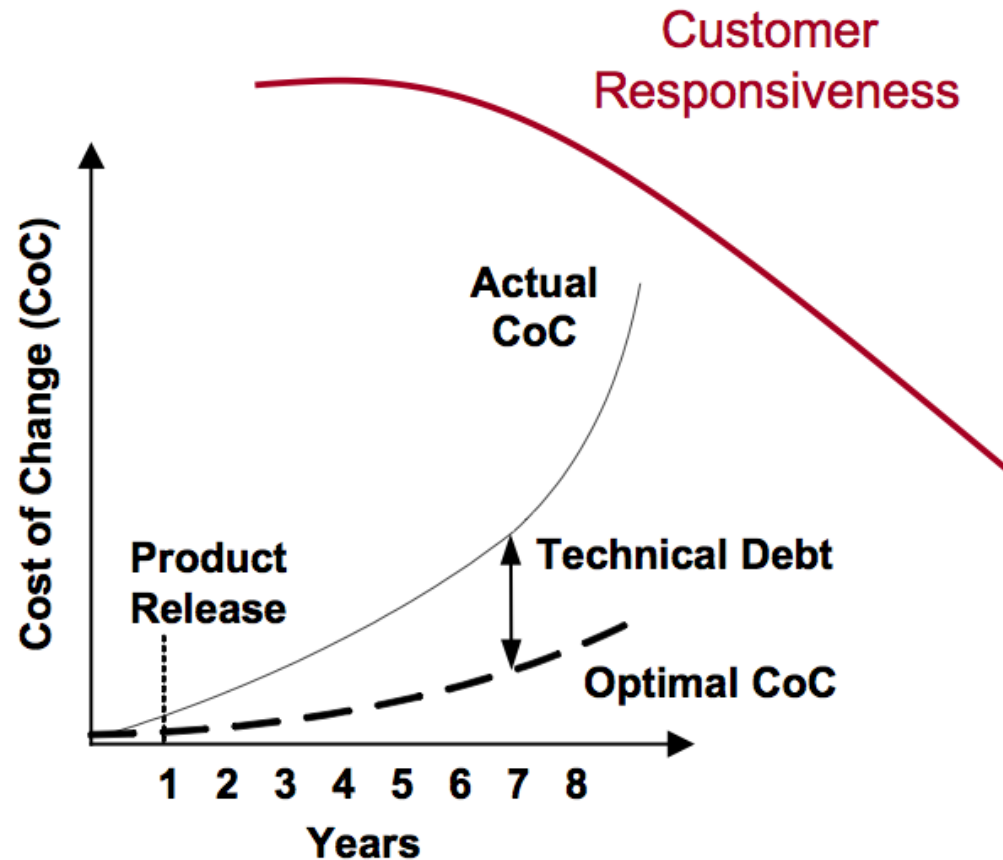
# Time is Money (I. Gat)

- 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

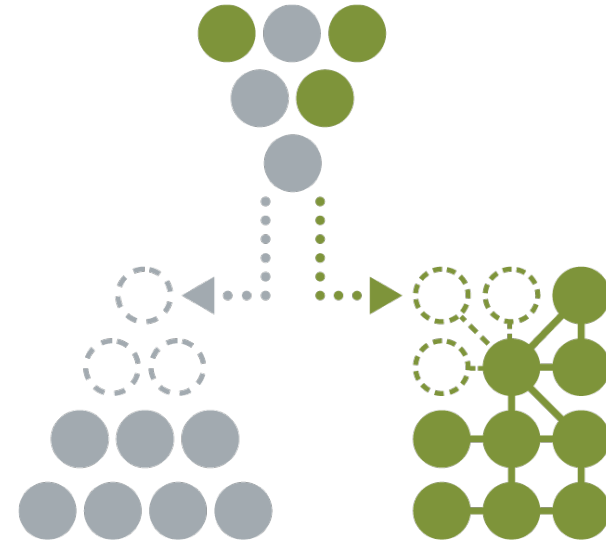


McConnell 2011

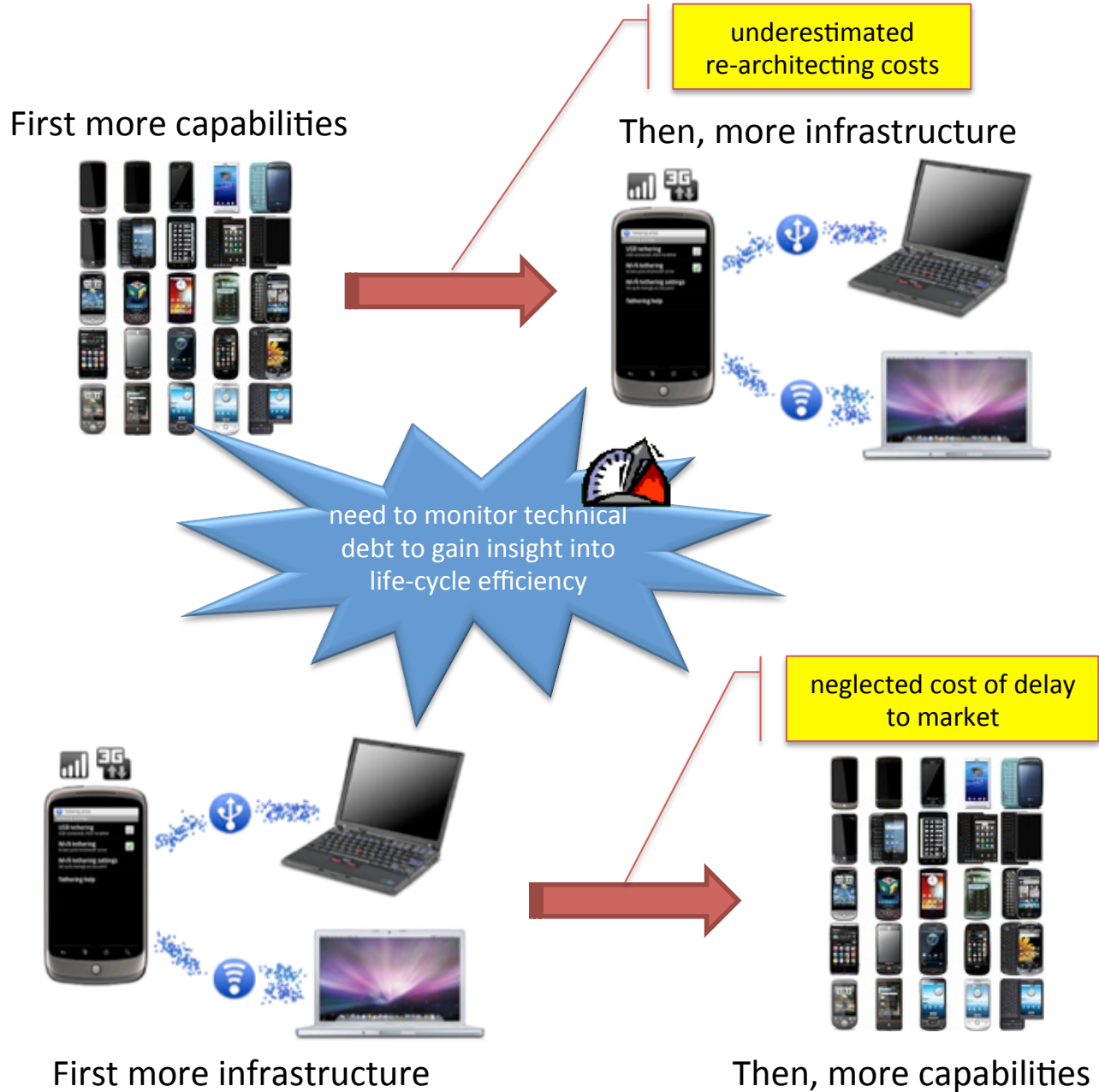
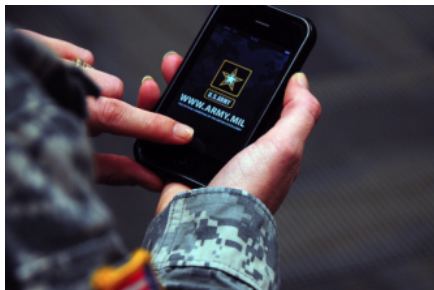
# Technical Debt Definition

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



# 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**

**CarnegieMellon**

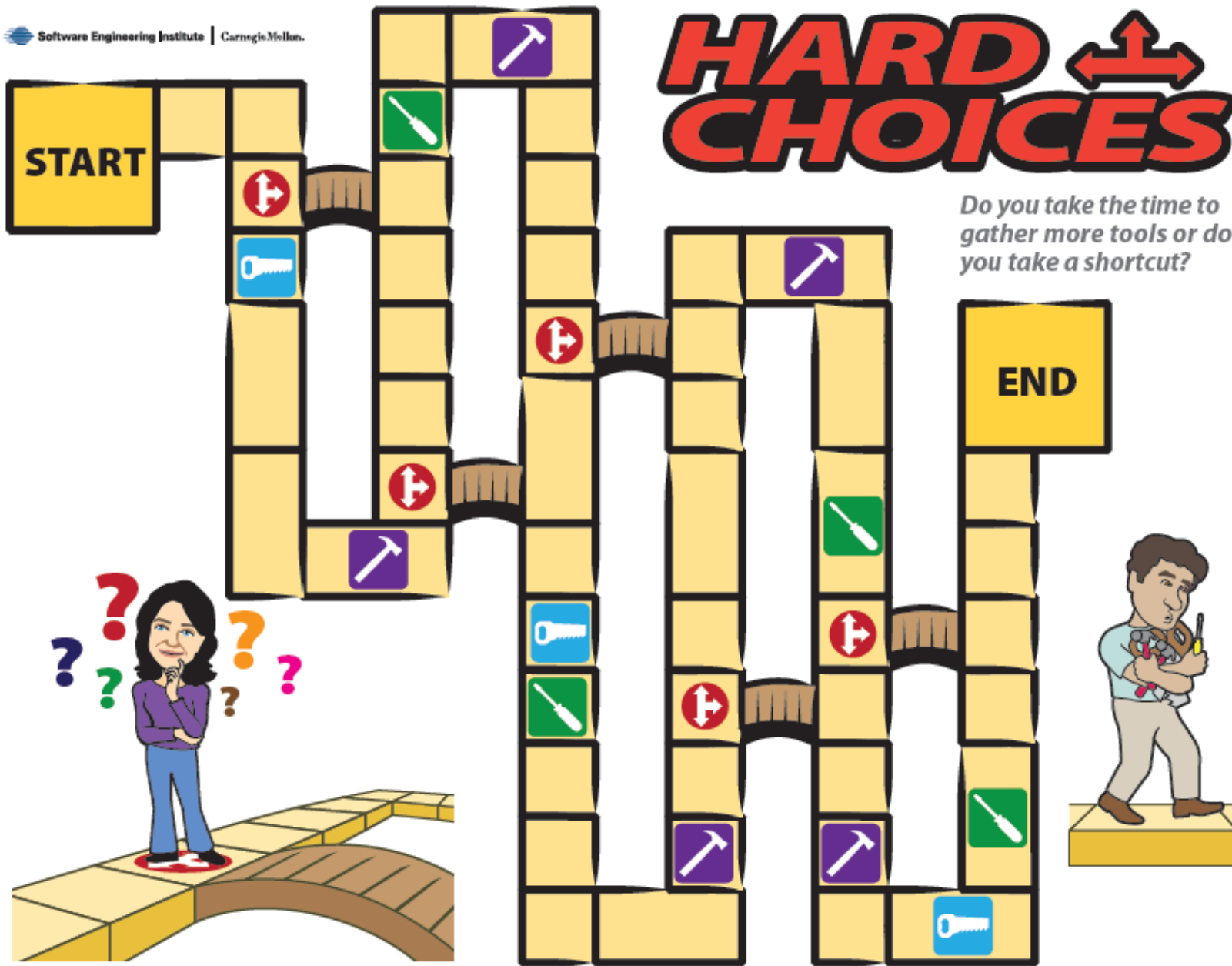


Hard Choices Strategy Game to Communicate Value of Architecture Thinking game  
downloadable from <http://www.sei.cmu.edu/architecture/tools/hardchoices/>.



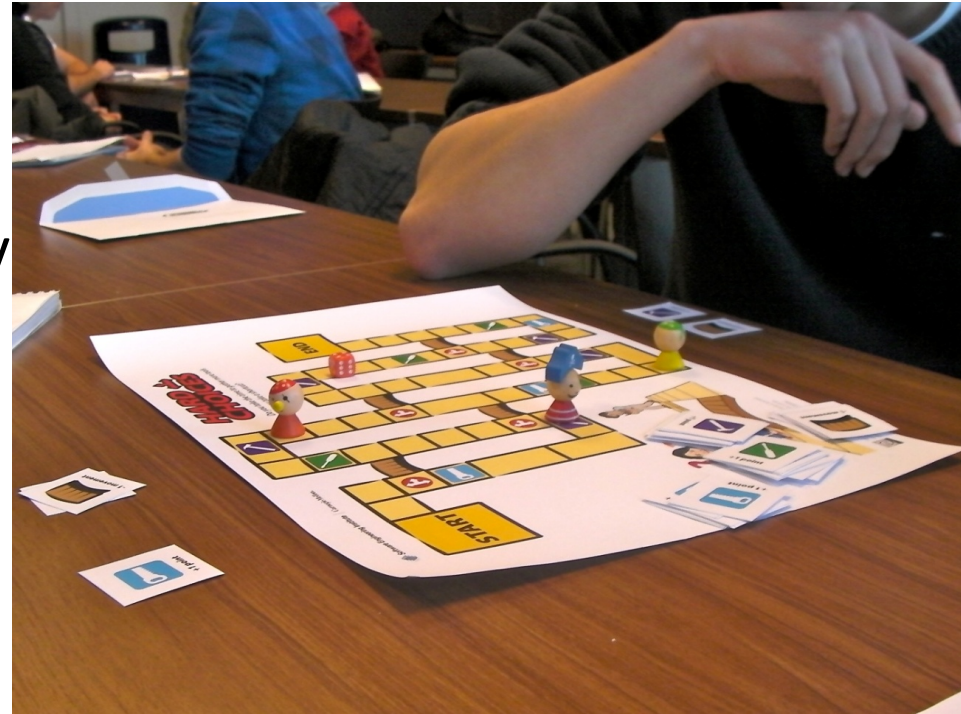
# HARD CHOICES

*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 not to take another. Instead you may play the card (returning it to the deck) and get a free turn.~~



# Debrief After the Game

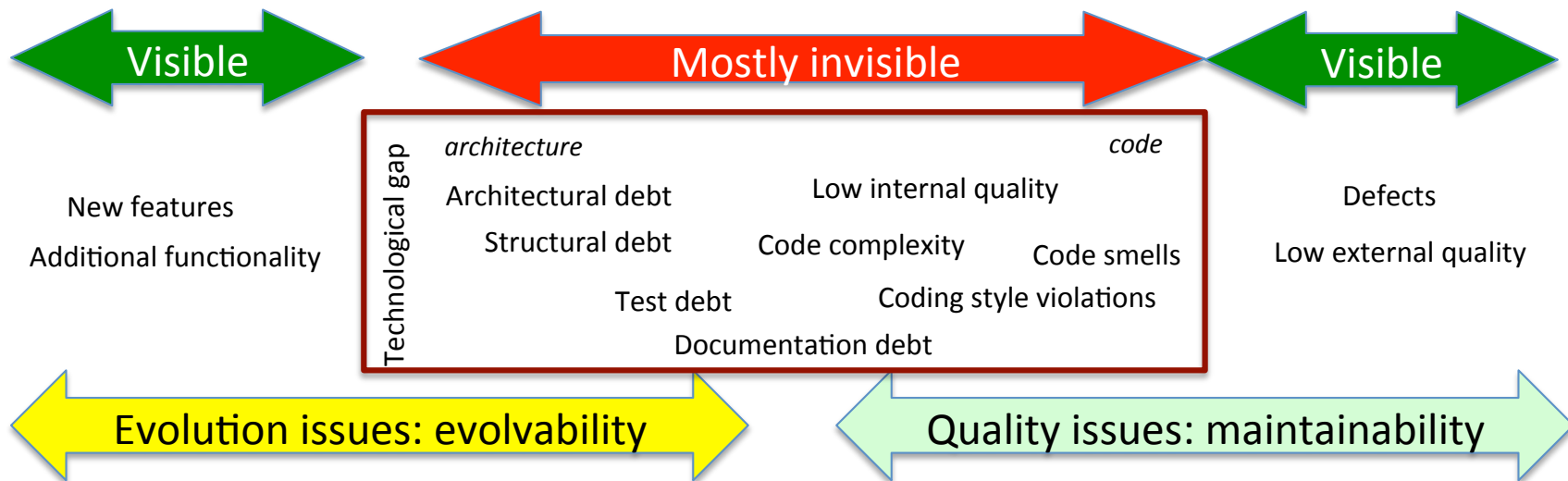


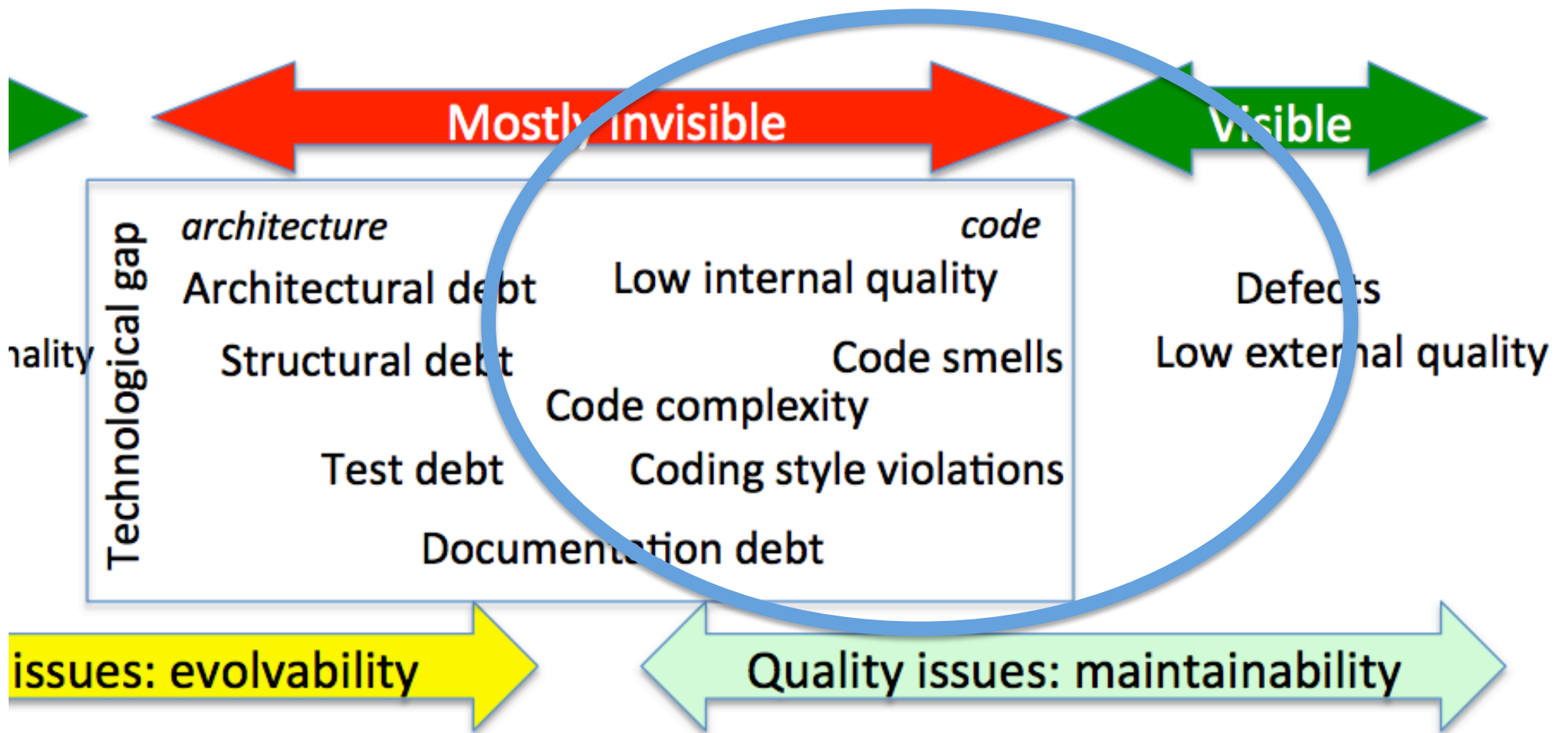
- What just happened?
  - How did you experience the game?
  - What strategies did you employ?
- So what?
  - How do your experiences in the game relate to the strategies you employ during software development in the face of uncertainty?
  - How does this relate to the choices you make—in investing effort 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?

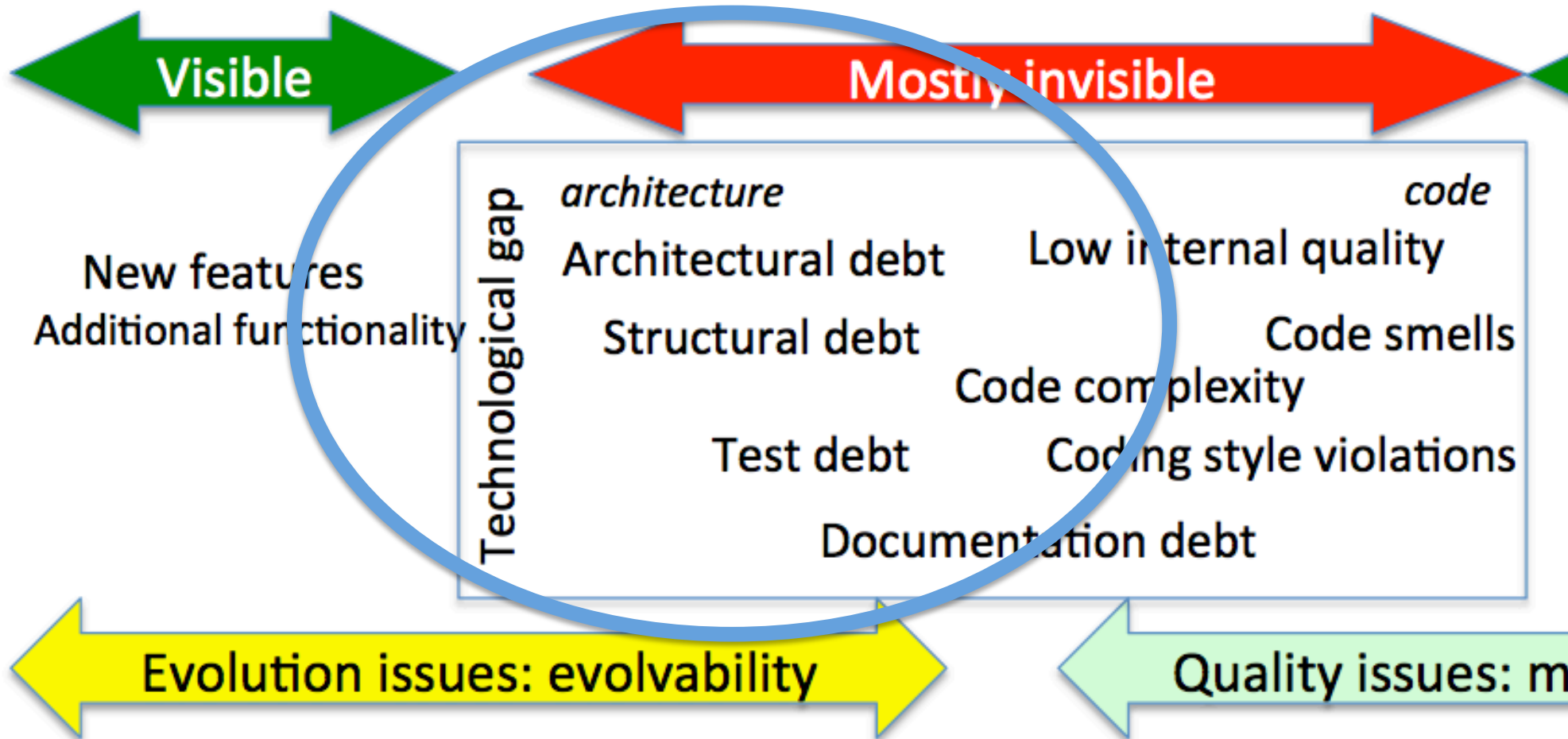
# Outline



- 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






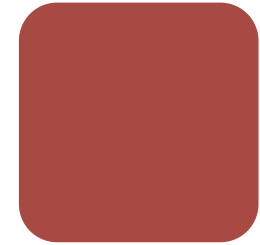


# TD: negative value, invisible

	Visible	Invisible
Positive Value	<b>New features Added functionality</b>	<b>Architectural, Structural features</b>
Negative Value	<b>Defects</b>	<b>Technical Debt</b>



# 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



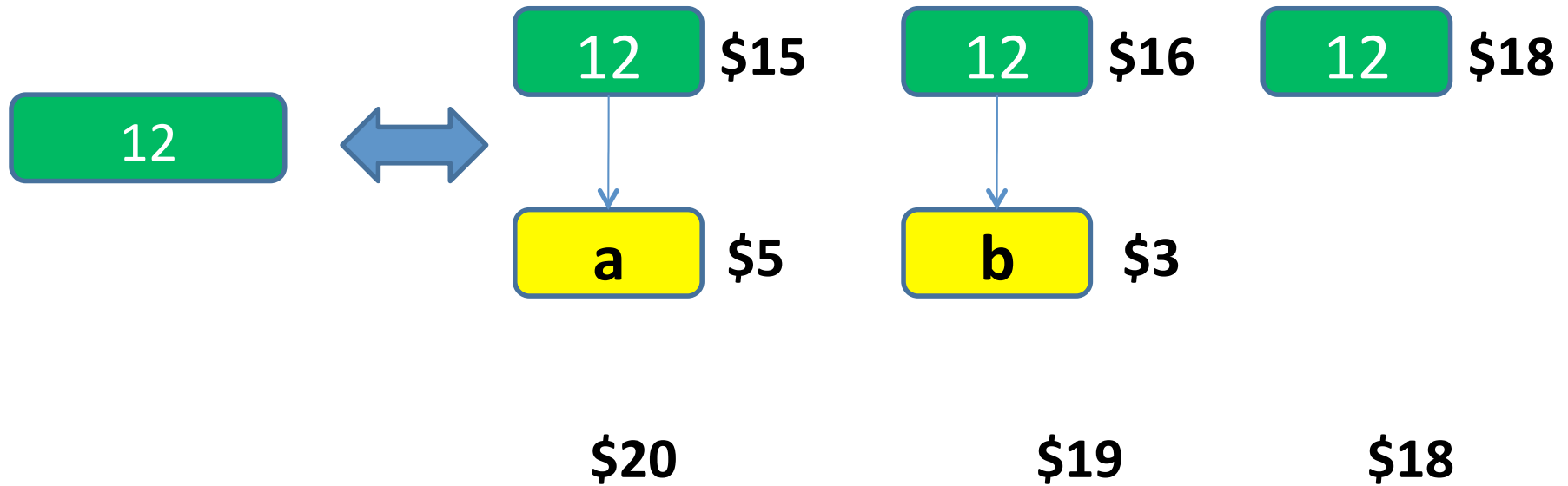
# Where the metaphor breaks

- Initial investment at  $T_0$  in an environment  $E_0$ .  
Now in  $T_2$ ,  $E$  has changed to  $E_2$ , 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

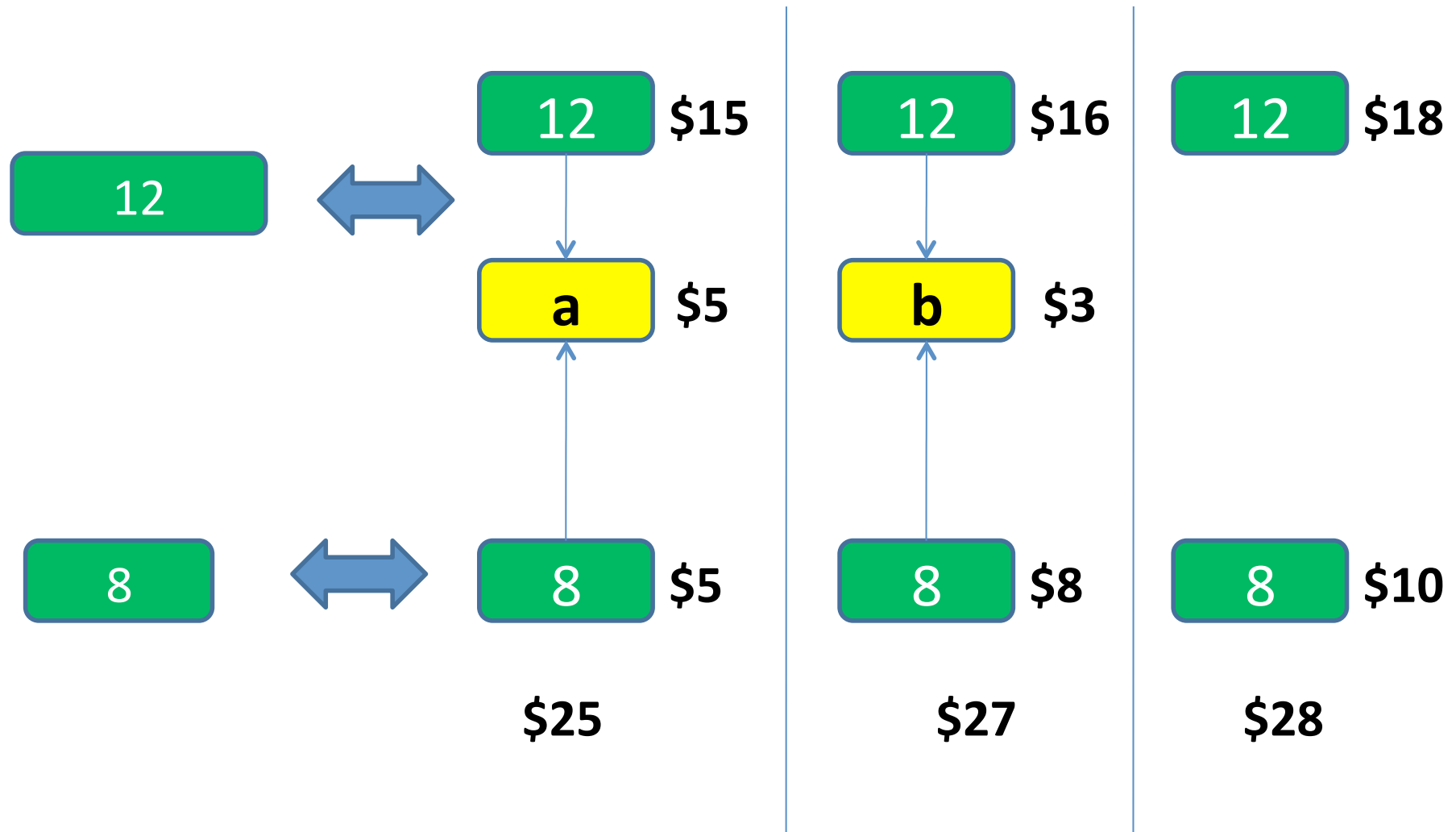
# Where the metaphor breaks...

- 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

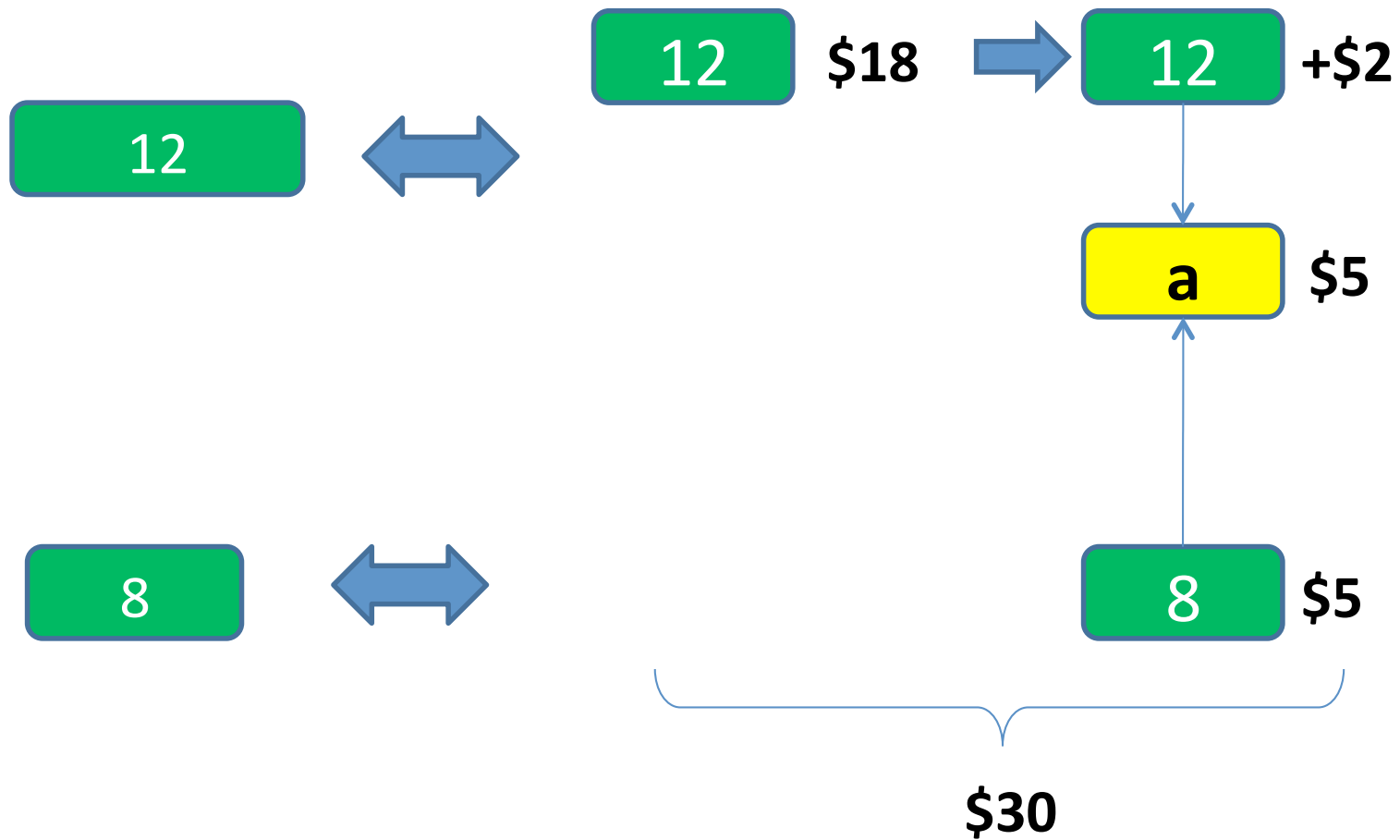
# Technical Debt (1)



# 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

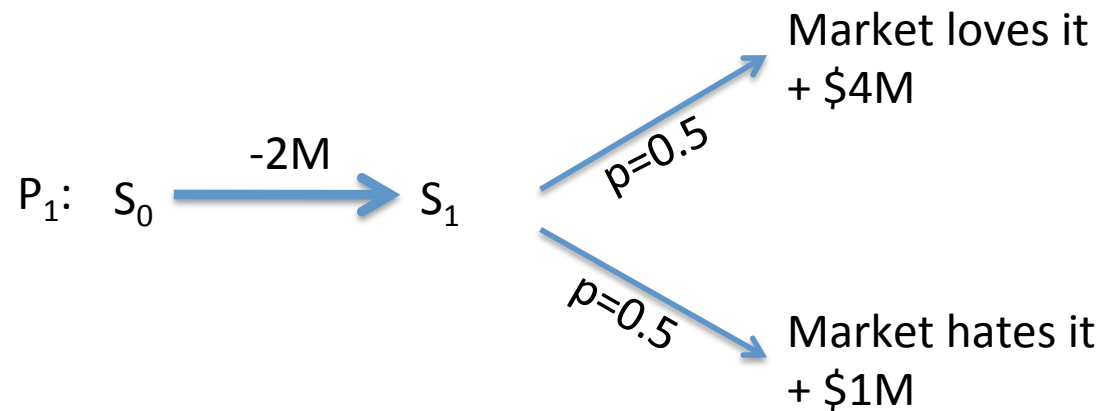
- If you are not incurring any interest, then it probably is not a debt



Technical debt  
as an investment ?



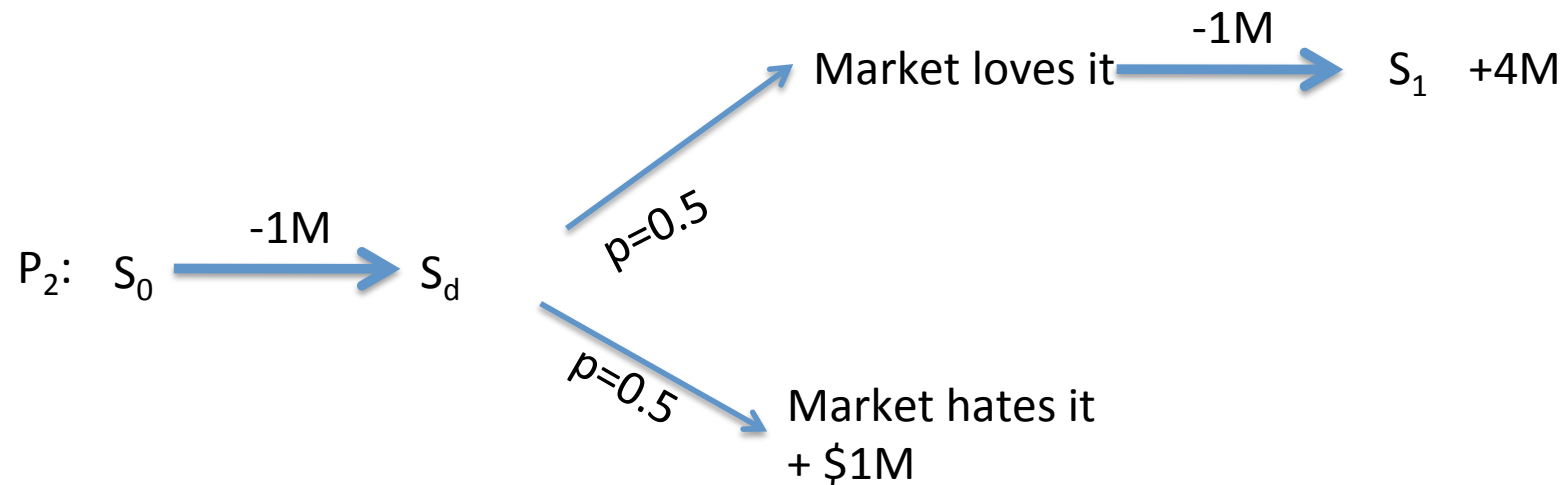
# TD and Real Options



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

Source: K. Sullivan, 2010  
at TD Workshop SEI 6/2-3

# TD and Real Options (2)

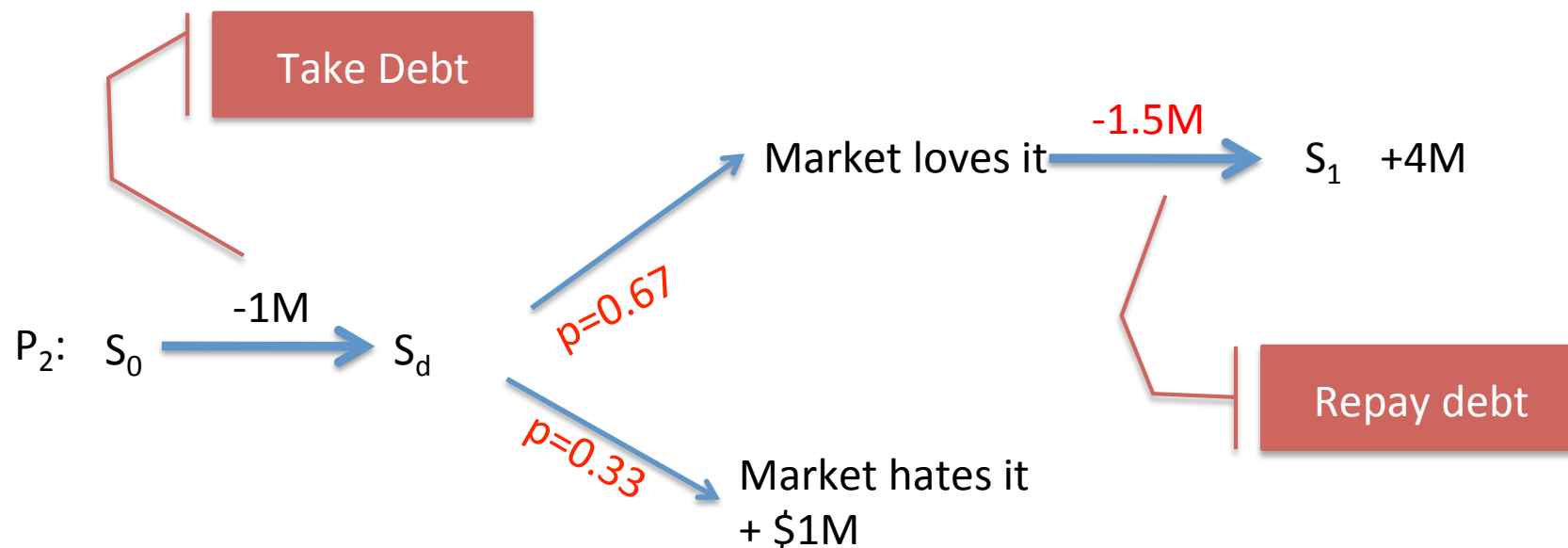


$$\text{NPV}(P_2) = -1M + 0.5 \times 3M + 0.5 \times 1M = 1M$$

Taking Technical Debt has increased system value.

Source: K. Sullivan, 2010

# TD and Real Options (3)



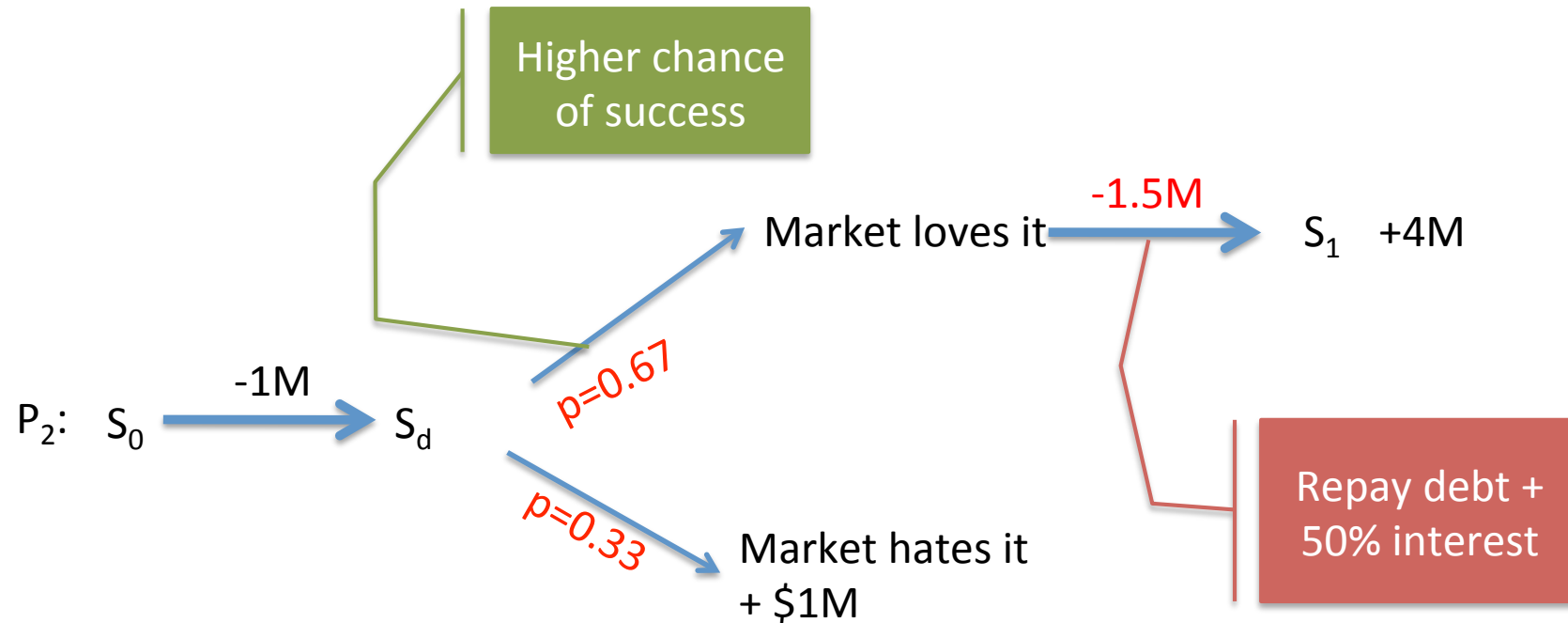
$$\text{NPV}(P_3) = -1M + 0.67 \times 2.5M + 0.33 \times 1M = 1M$$

More realistically:

Debt + interest

High chances of success

# TD and Real Options (3)



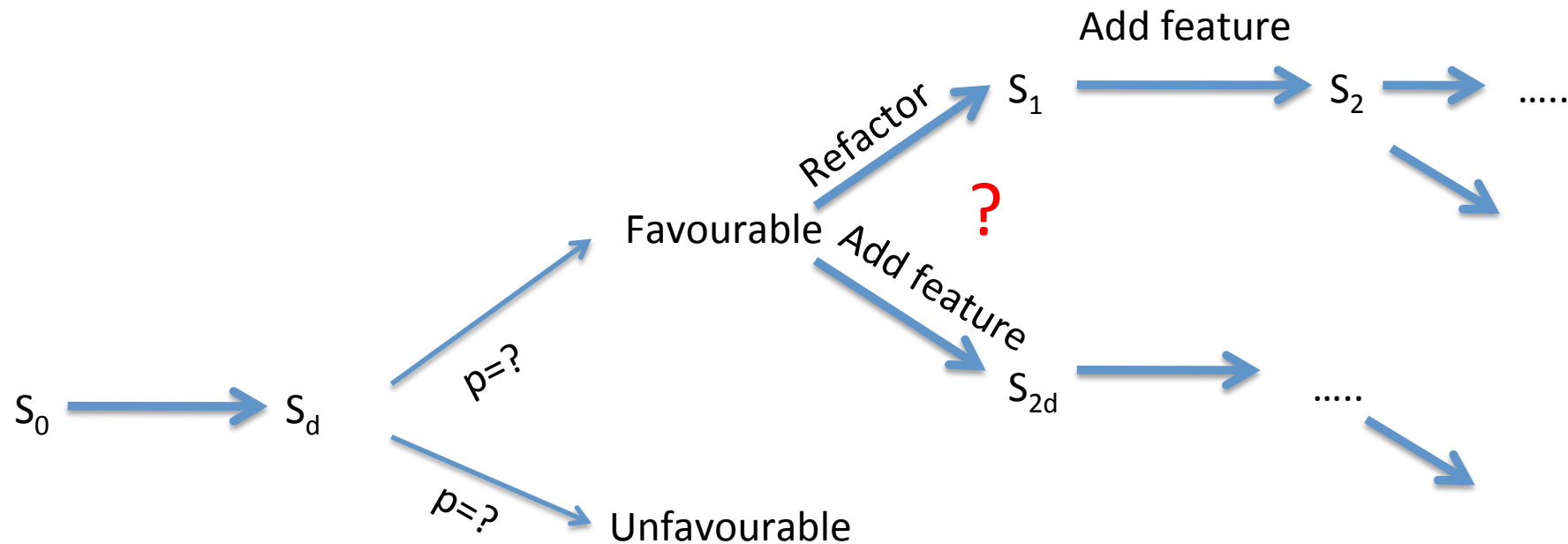
$$NPV (P_3) = -1M + 0.67 \times 2.5M + 0.33 \times 1M = 1M$$

More realistically:

Debt + interest

High chances of success

# 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

# 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:

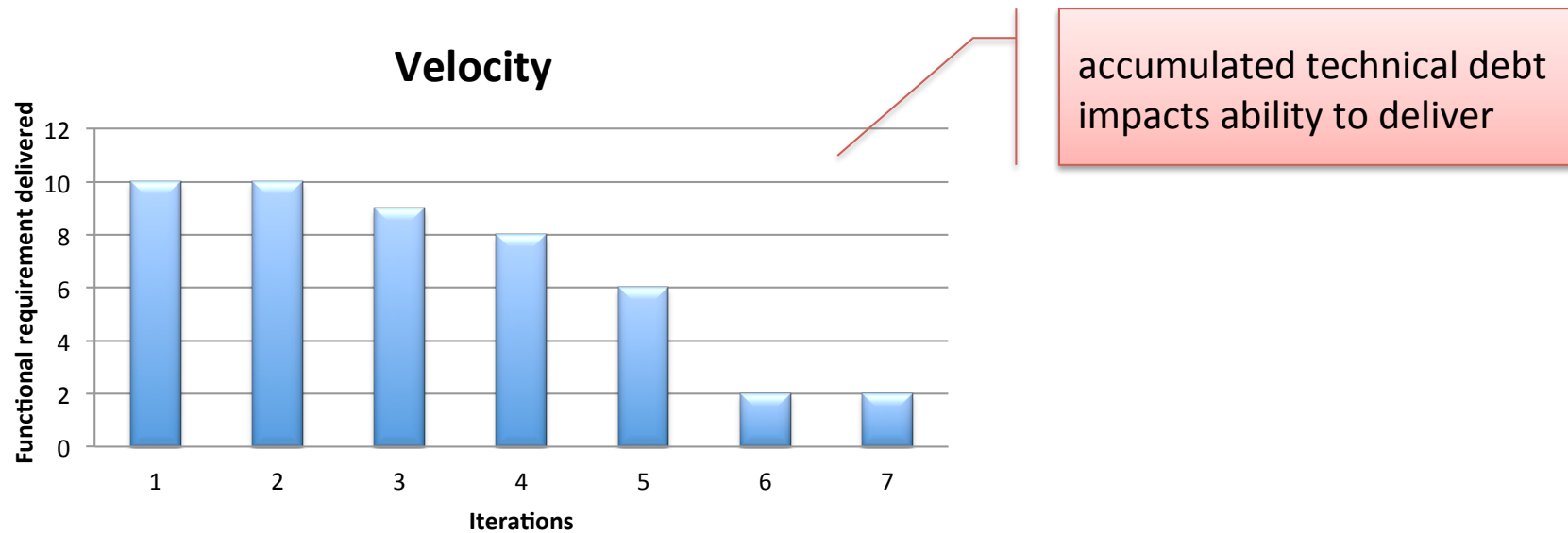
1. Ignorance is bliss
2. The elephant in the room
3. Big scary \$\$\$\$ numbers
4. Five star ranking
5. Constant reduction
6. We're agile, so we are immune!





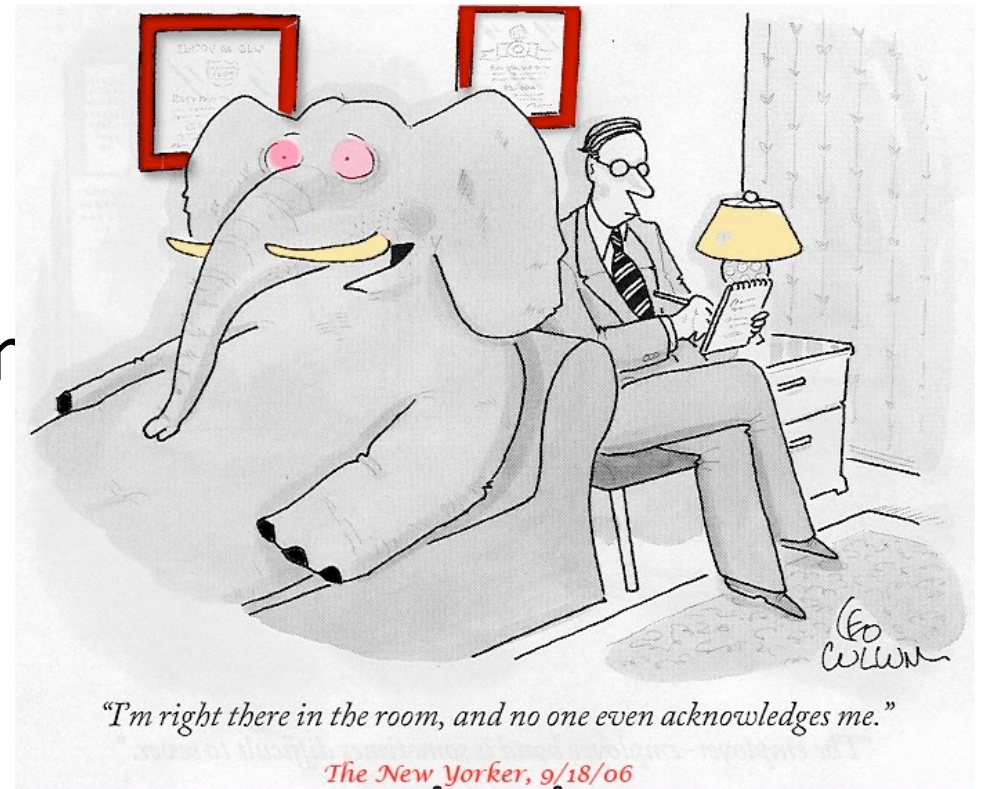
# Ignorance is bliss

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



# 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

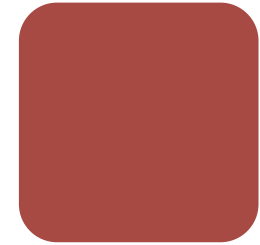
# Static analysis + Consulting

- 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

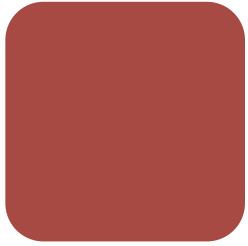


- Fits the metaphor, indeed.
- Looks very objective... but...
- Subjective in:
  - What is counted
  - What tool to use
  - Cost to fix

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”??



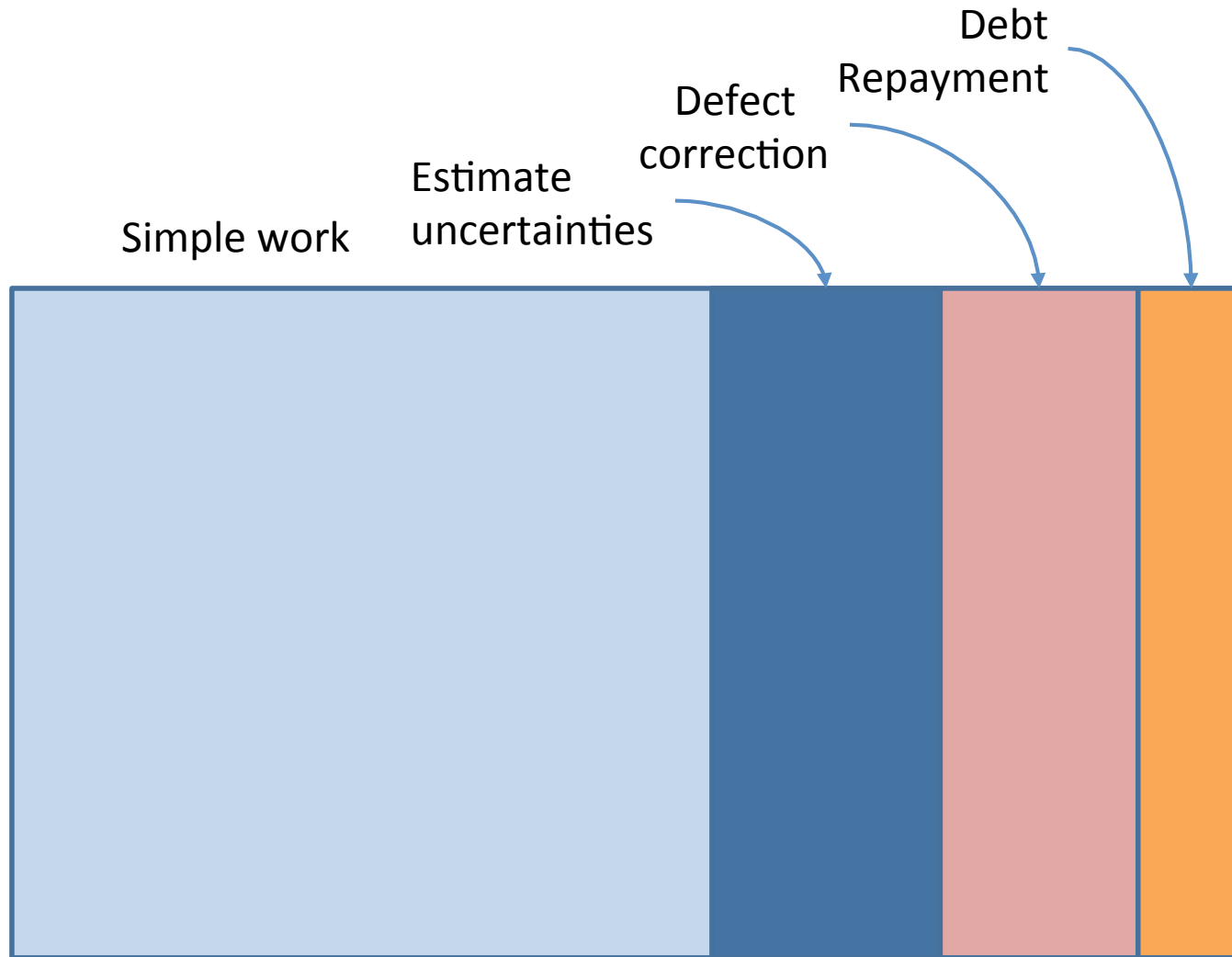
# Five star ranking

- 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

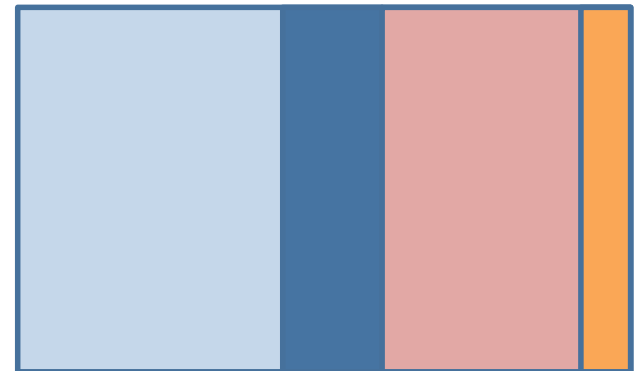
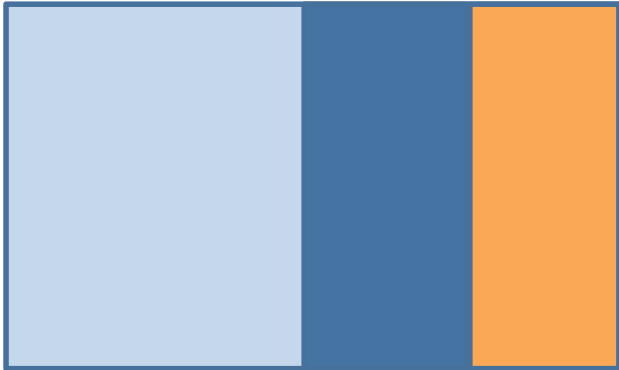
- Make technical debt a visible item on the backlog
- Make it visible outside of the software dev. organization
- Incorporate debt reduction as a regular activity
- Use buffer in longer term planning for yet unidentified technical debt
- Lie (?)

# 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*

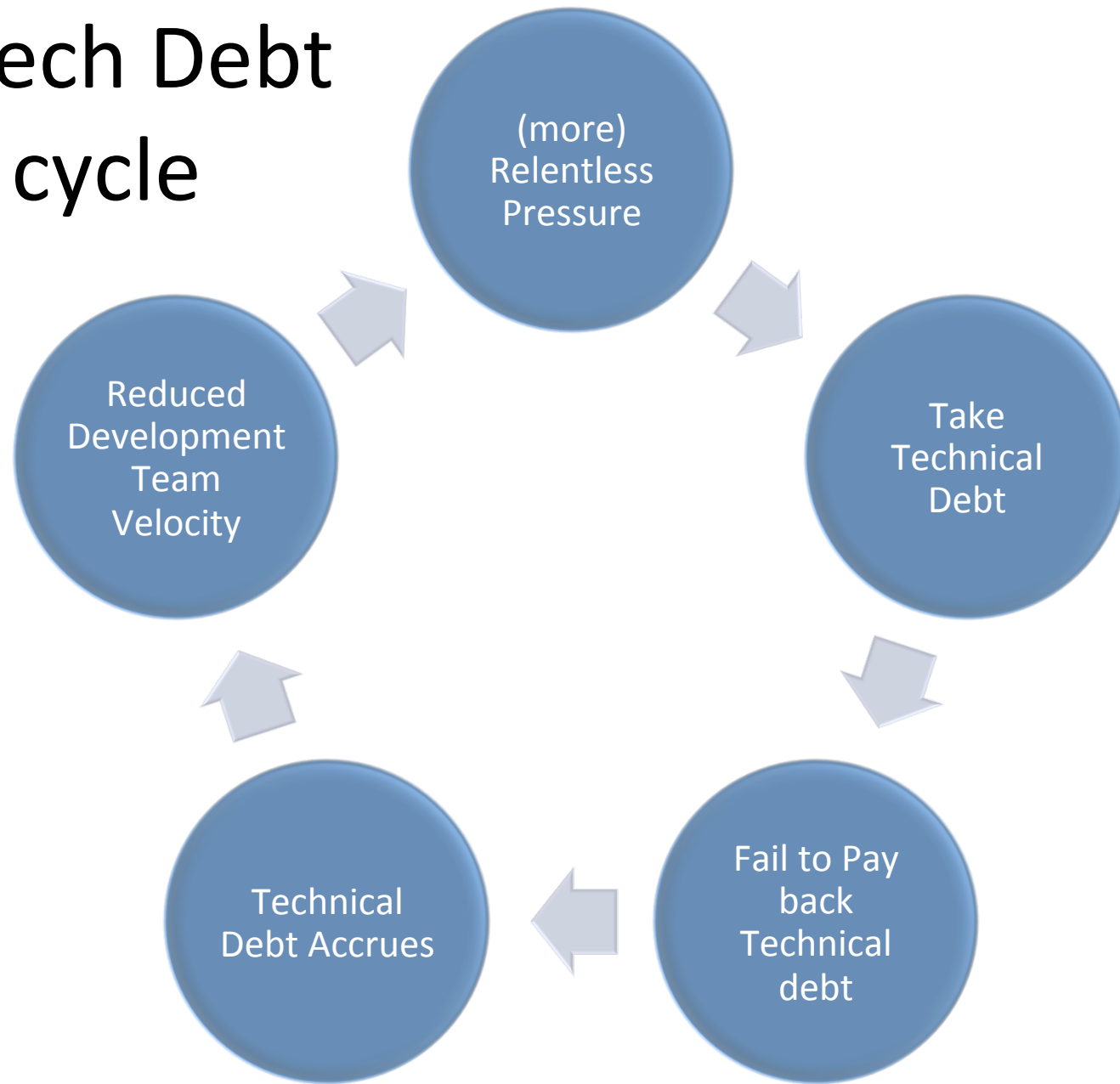


# Hitting the wall

- 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

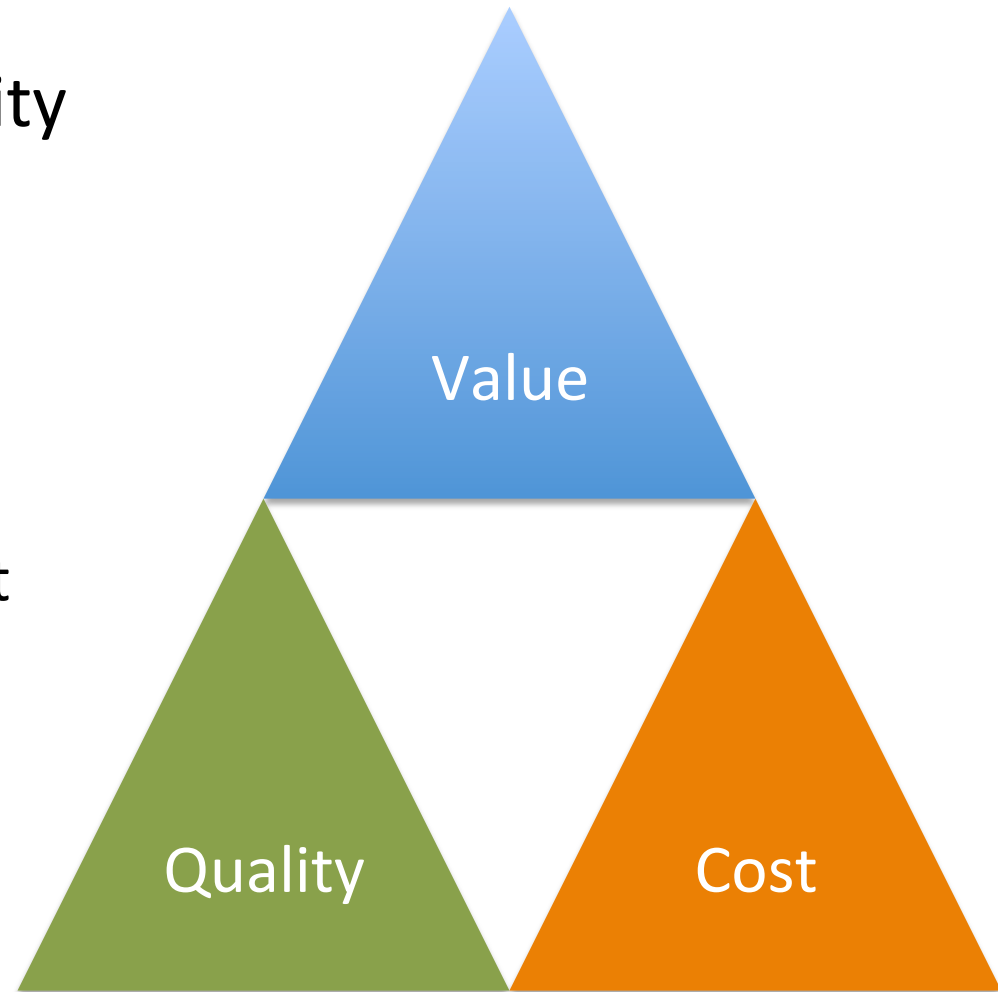


**Israel Gat, 2010**

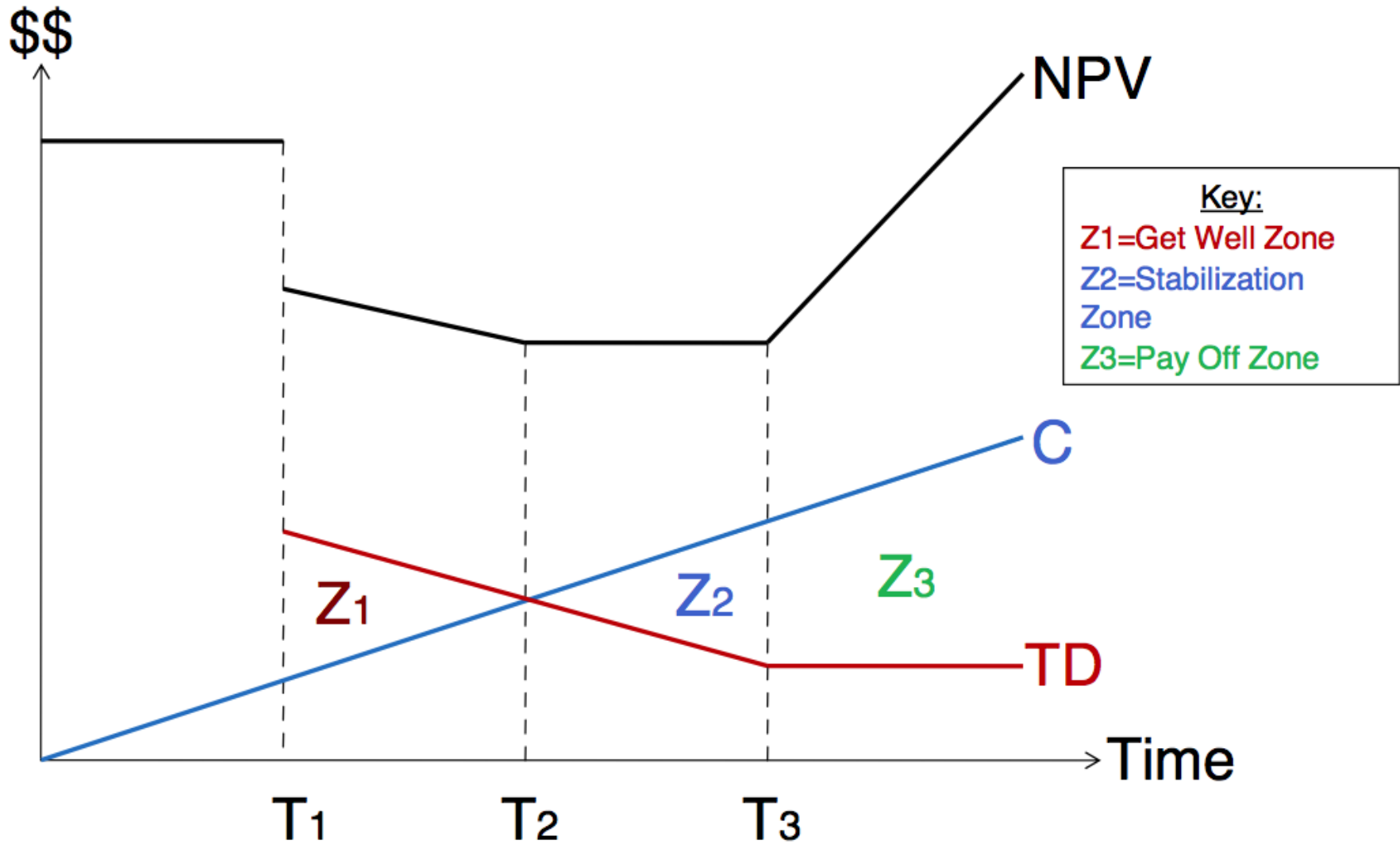
<http://theagileexecutive.com/2010/09/20/how-to-break-the-vicious-cycle-of-technical-debt/>

# Value, Quality, Constraints

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



# Evolution over time

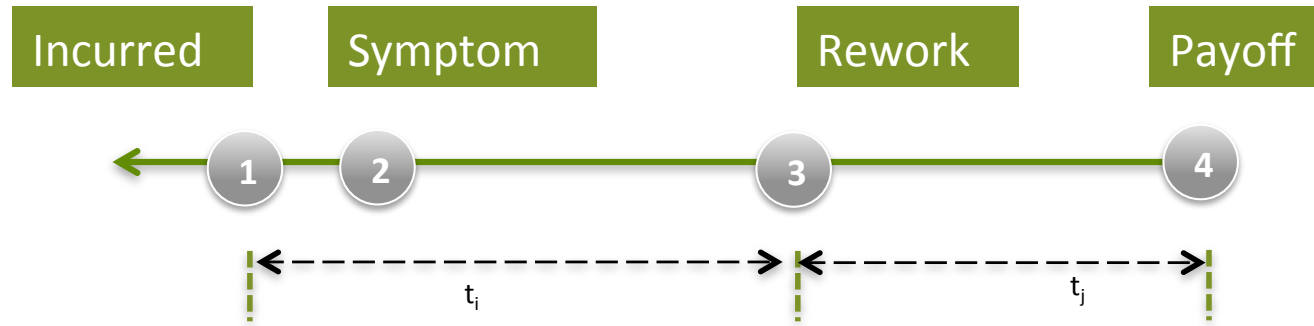


Gat & Heintz, Cutter, 2010





# Timeline



## *What is the debt?*

Technical debt item description  
Risk analysis, Development state analysis

Intentional and strategic

5 →

## *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

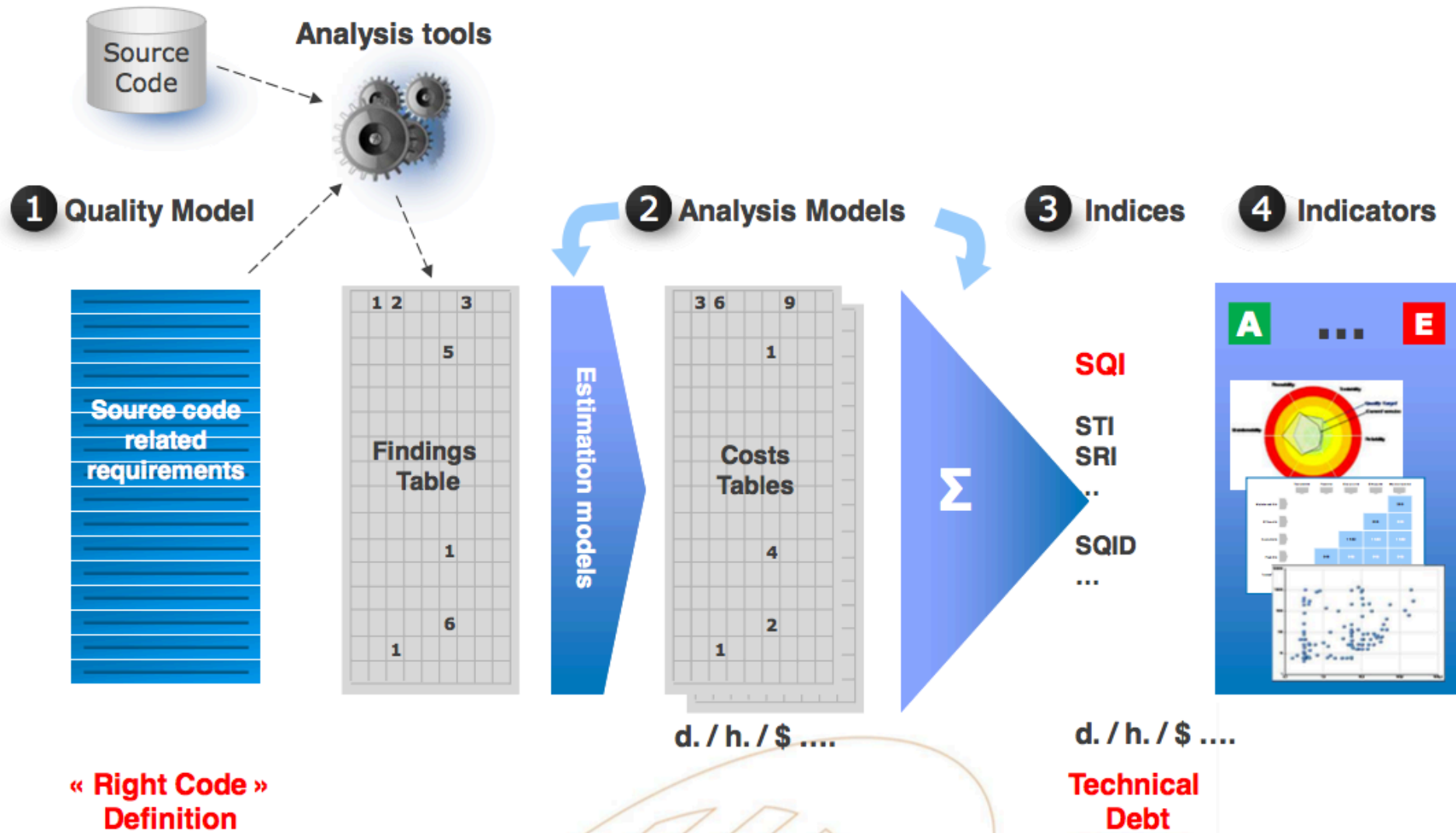
- Vendors include
  - CAST
  - Inspearit
  - SonarSource (Sonarqube)
  - Thoughtworks
  - Software Improvement Group (SIG)
  - Lattix
  - Hello2morrow
  - Tocéa (Scertify)
  - Xdepend
  - Klocwork
  - JetBrains

- Real Option theory
- Dependency Structure Matrix
  - Propagation cost
- Sonarqube
- SQALE
- Scertify

# SQALE

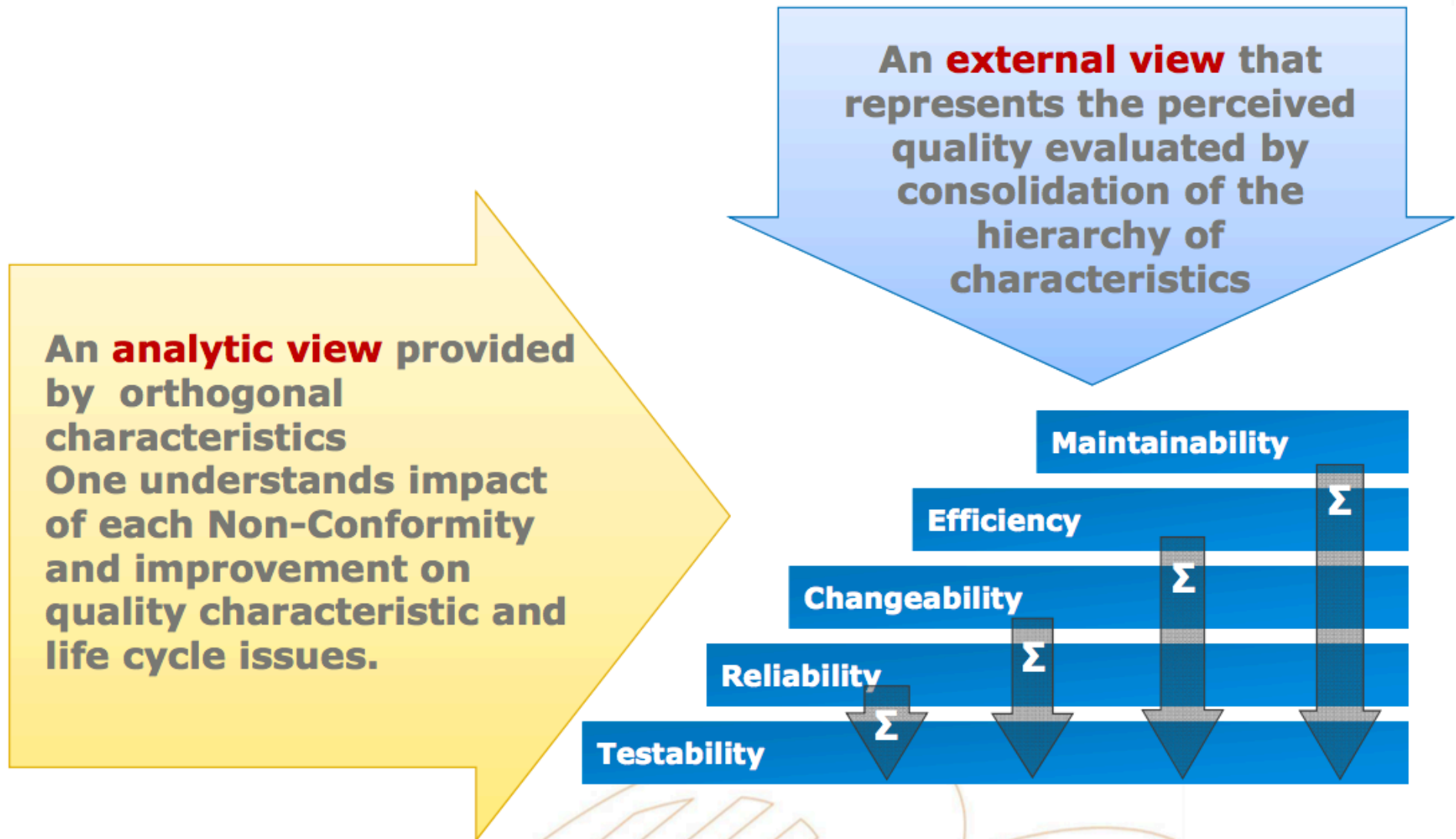
- SQALE = Software Quality Assessment based on Lifecycle Expectation
- Jean-Louis Letouzey and Thierry Coq
- Inspearit
  - (previously known as Det Norske Veritas France)

# SQALE

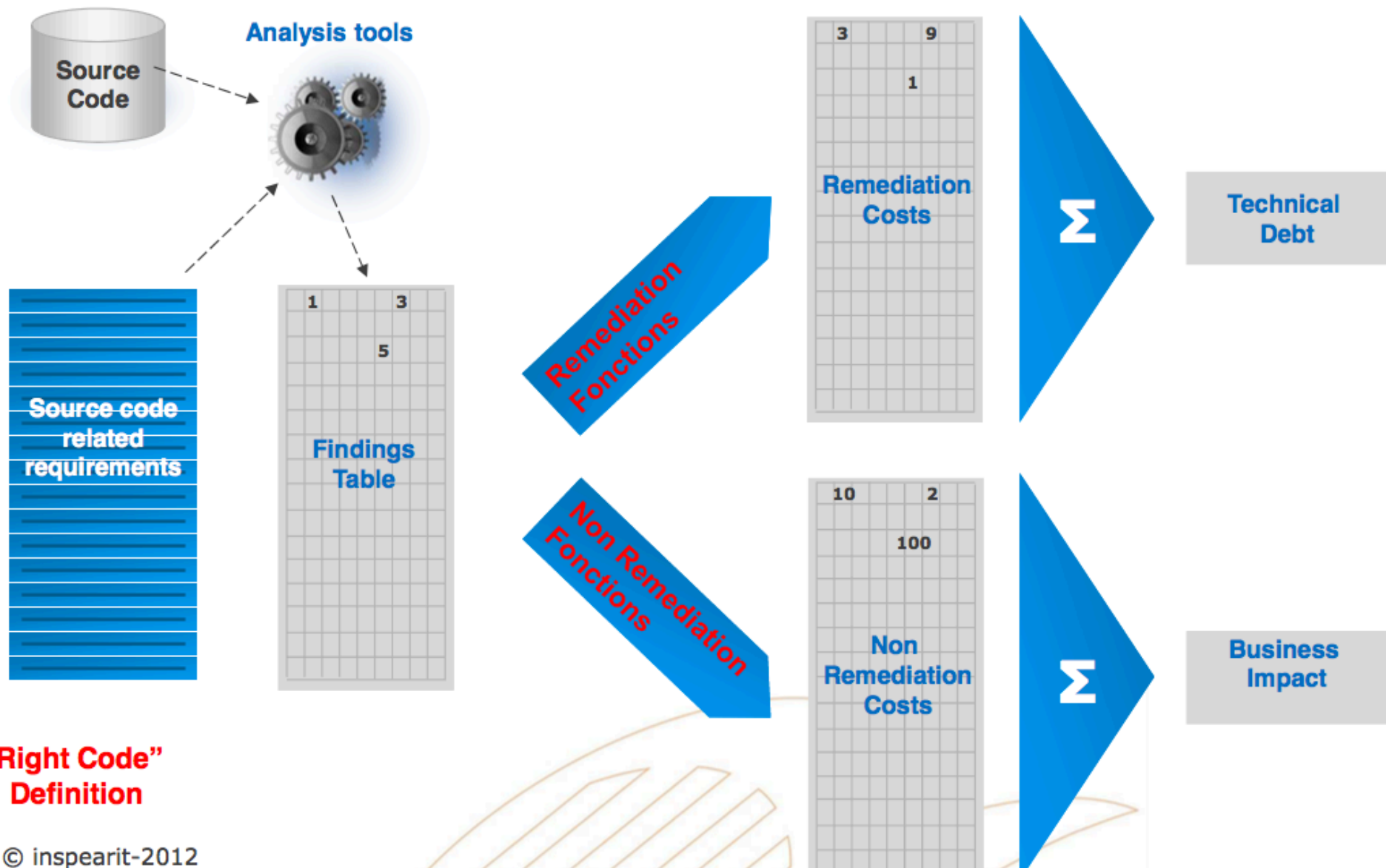




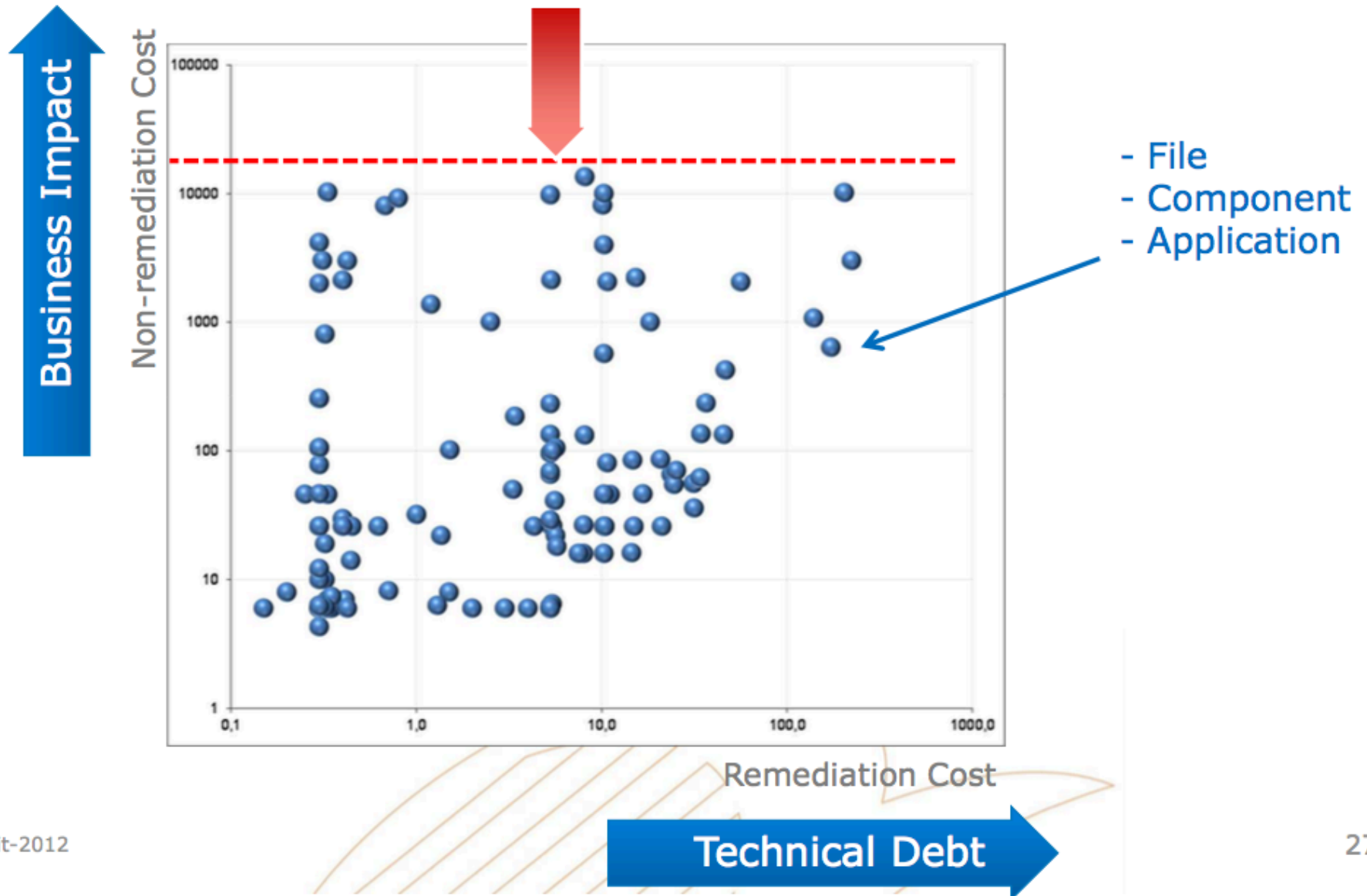
# Quality model



# Costs

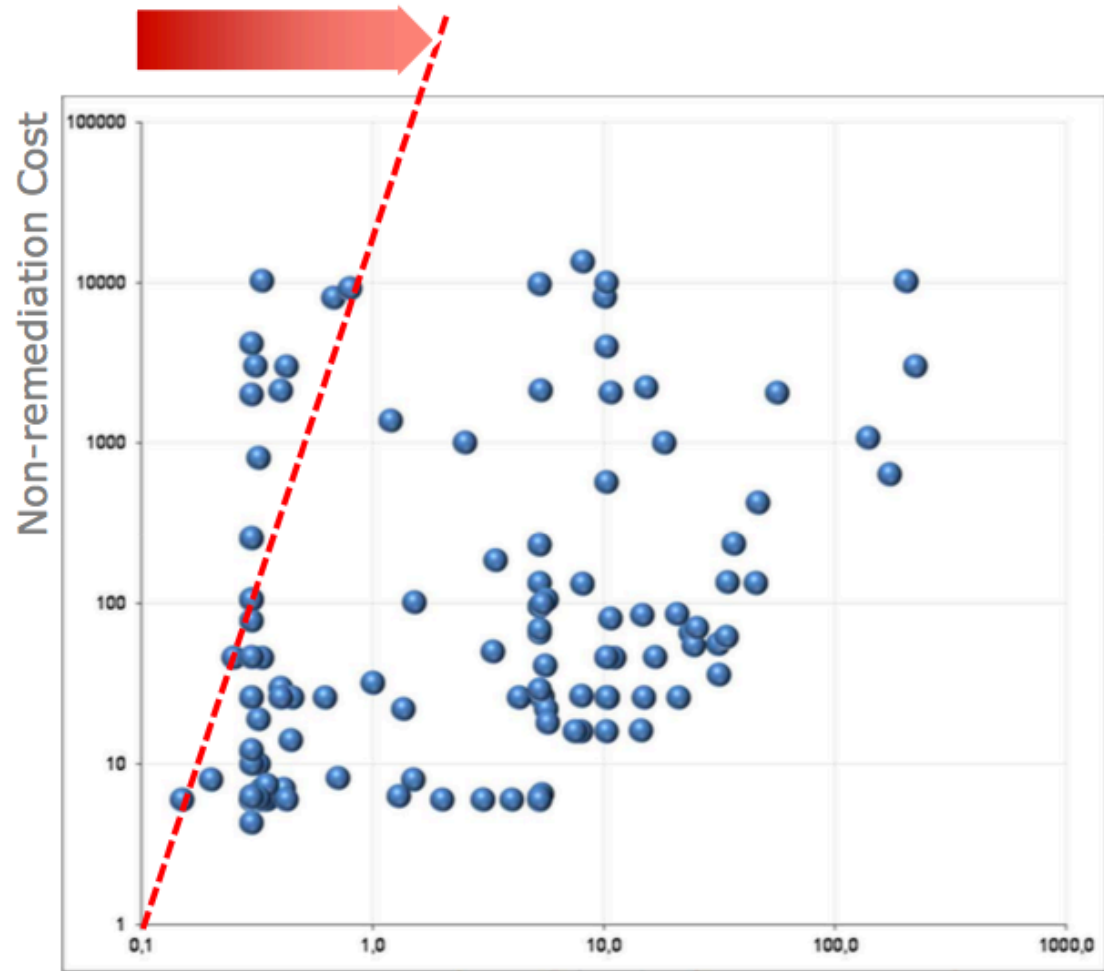


# Reduce business impact



# Maximize RoI

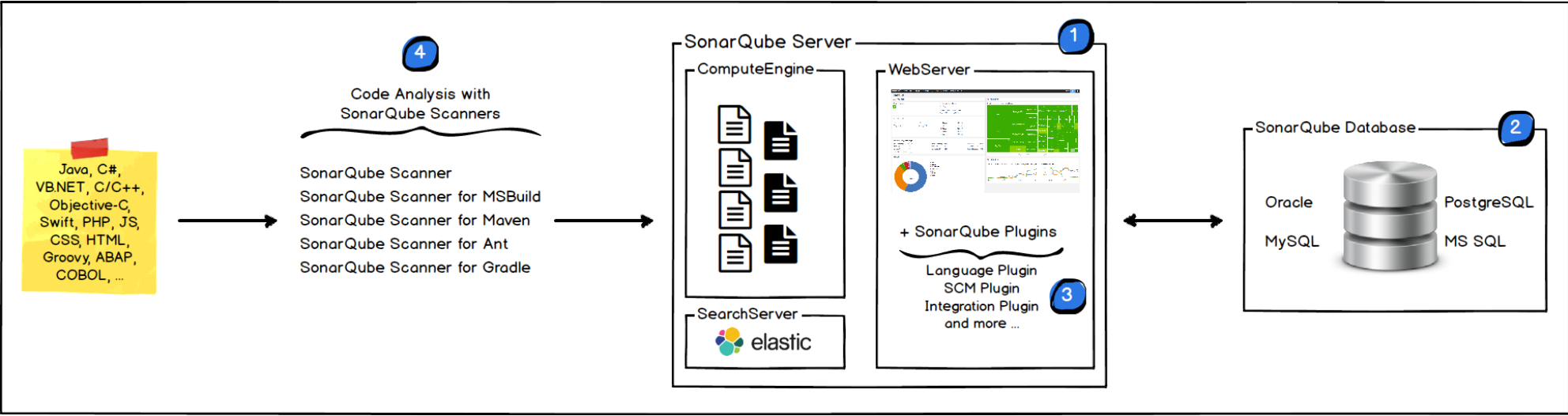
Business Impact



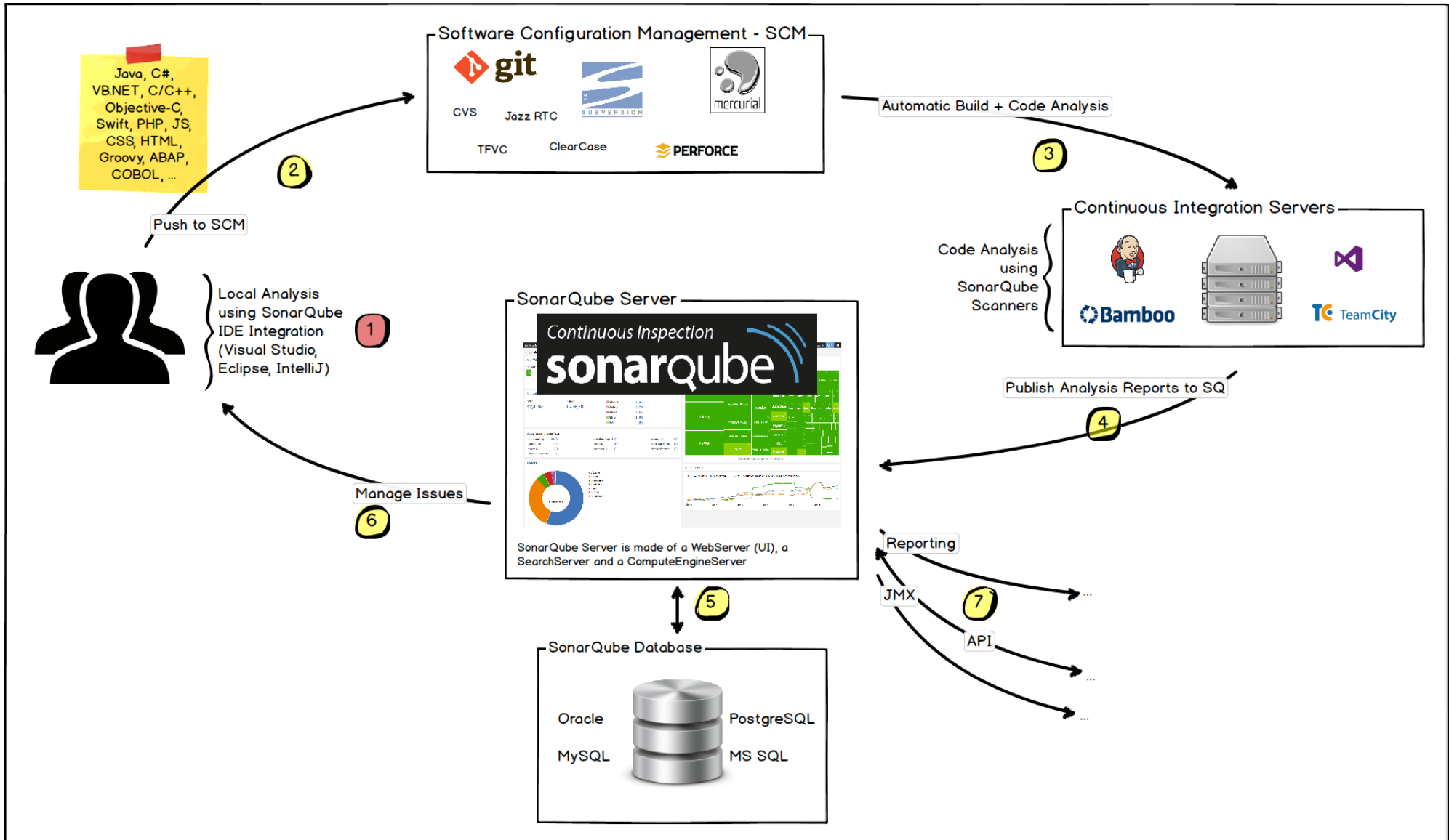
Remediation Cost

Technical Debt

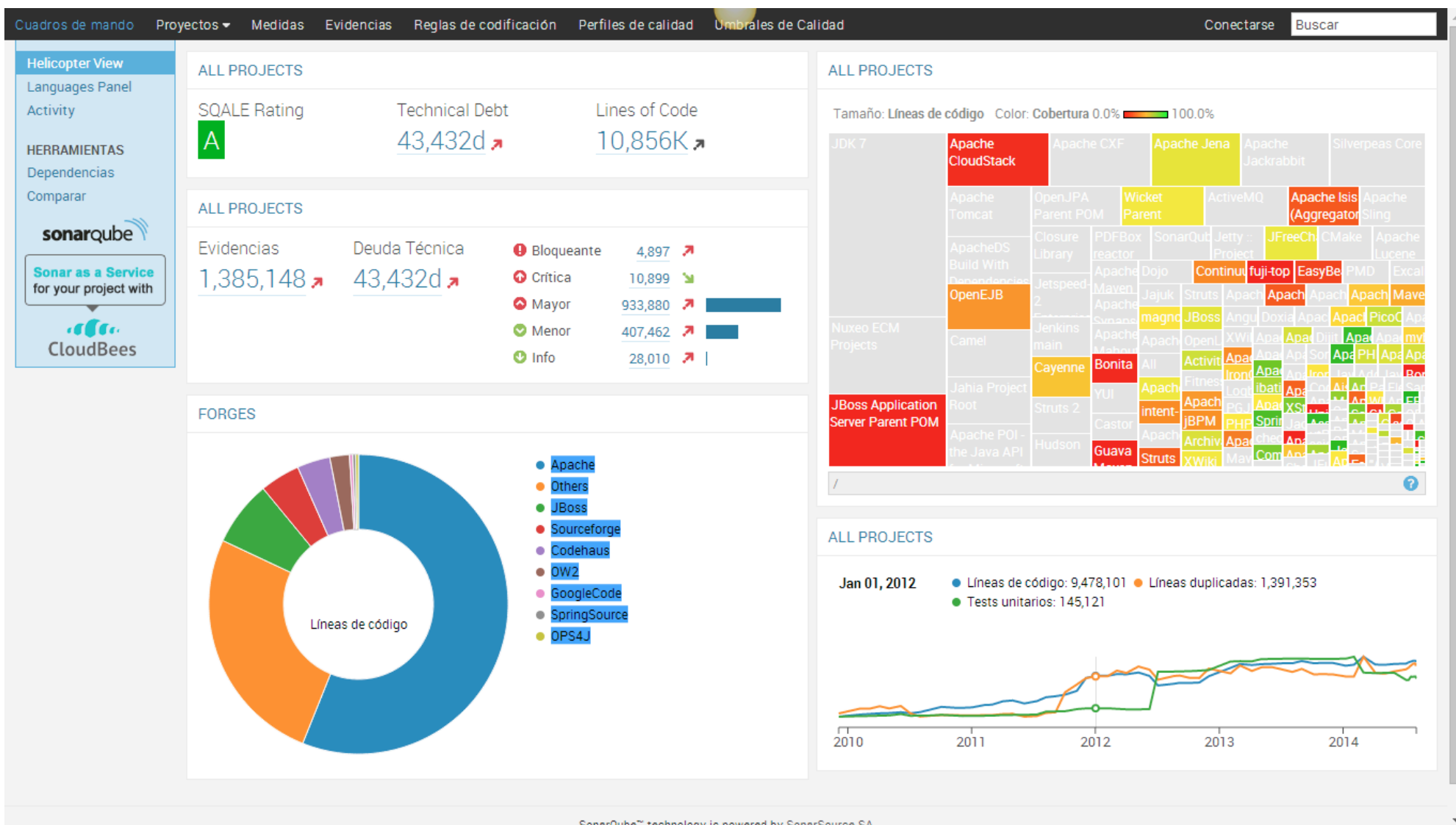
# SonarQube



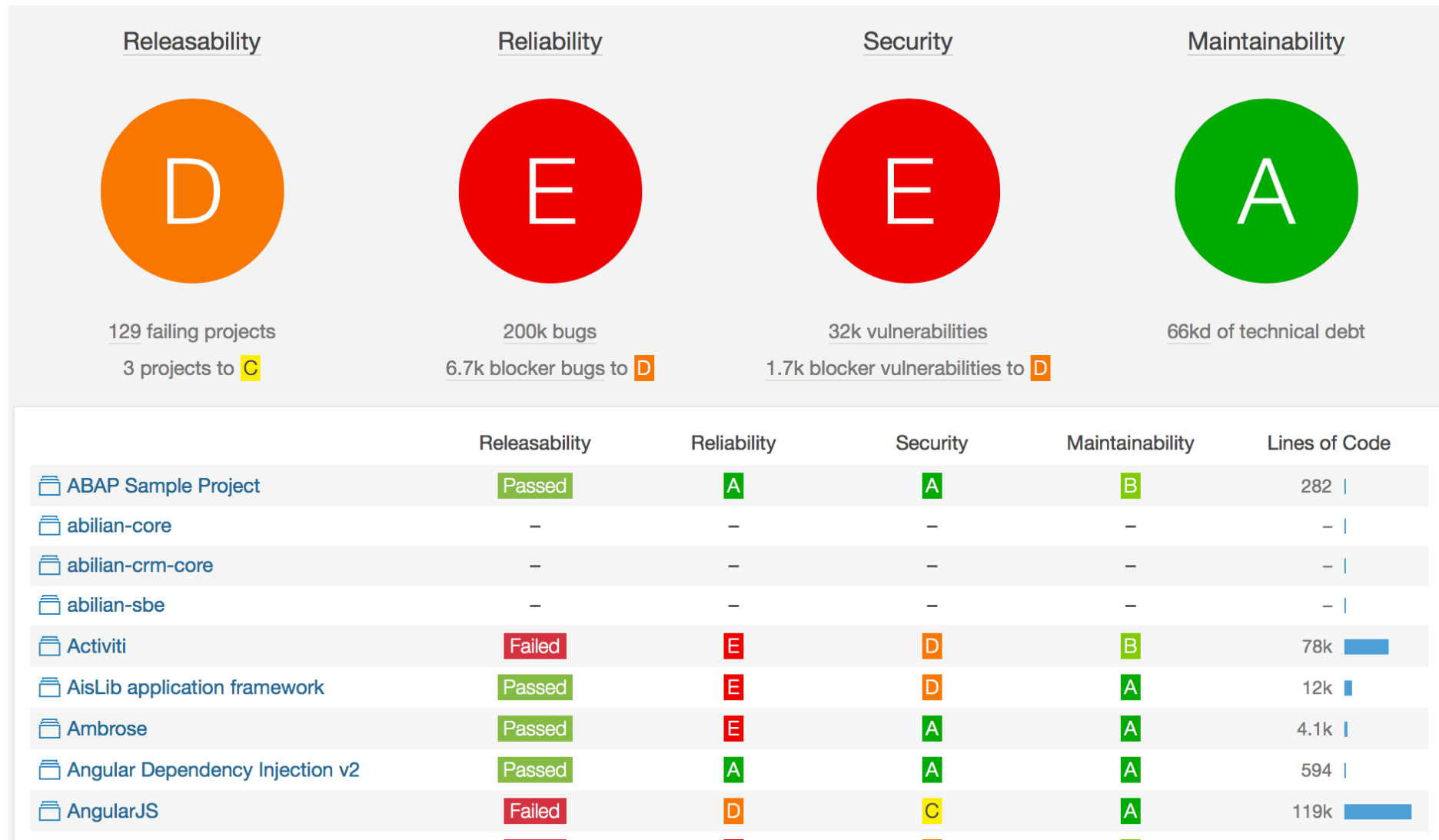
# SonarQube



# SonarQube and SQALE



# SQALE-like dashboard with SonarQube





# Structural level

- Dependency analysis
  - Propagation cost
- Interview the designers
  - TD and its causes are in their heads

# Dependency Structure Matrix

---

	A	B	C
A		Strength of B's dependency on A	
B	Strength of A's dependency on B		Strength of C's dependency on B
C			

The diagram illustrates a Dependency Structure Matrix (DSM) for three variables: A, B, and C. The matrix is a 3x3 grid with a light pink background. The columns are labeled A, B, and C at the top. The rows are labeled A, B, and C on the left. The diagonal cells (A-A, B-B, C-C) are empty. The off-diagonal cells contain text describing dependencies: 'Strength of B's dependency on A' in the B-A cell, 'Strength of A's dependency on B' in the A-B cell, and 'Strength of C's dependency on B' in the C-B cell. Blue arrows point from the diagonal cells to the off-diagonal cells: from A-A to B-A, from B-B to A-B, and from B-B to C-B.

# Dependencies for MS-Lite

		Alarm Notificat	L&R Engine	Alarm Engine	Alarm Manage	Logon	Client Updates	Rule Processo	Adapter Mana	User Session	Data Access	Cache	FSS Adapter	Data Persisten	Publish-Subsc
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
US 01	1						X			X					X
US 02	2						X			X					X
US 03	3			X				X		X					
US 04	4	X		X	X			X		X					X
US 05	5				X						X				
US 06	6		X					X		X					X
US 07	7		X	X										X	
US 08	8					X					X				
US 09	9								X				X		
ACT 14	23								X						
ACT 15	24								X				X		X
ACT 16	25								X				X		X
ACT 17	26						X			X		X			X
ACT 18	27	X					X								
ACT 19	28	X													
ACT 20	29					X					X				
ACT 21	30									X					



# Dependencies in Release Planning

## Dependencies between stories & supporting architectural elements

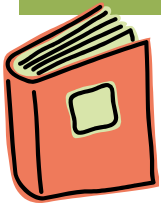
Understanding the dependencies between stories and architectural elements enables staged implementation of technical infrastructure in support of achieving stakeholder value.

## Dependencies between architectural elements

Low-dependency architectures are a critical enabler for scaling-up agile development.<sup>1</sup>

## Dependencies between stories

High-value stories may require the implementation of lower-value stories as precursors.<sup>2</sup>



<sup>1</sup> Mary and Tom Poppendieck – “Leading Lean Software Development”

<sup>2</sup> 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

# 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

- Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., et al. (2010). *Managing Technical Debt in Software-Intensive Systems*. Paper presented at the Future of software engineering research (FoSER) workshop, part of Foundations of Software Engineering (FSE 2010) conference.
- Brown, N., Nord, R., Ozkaya, I., Kruchten, P., & Lim, E. (2011). Hard Choice: A game for balancing strategy for agility. Paper presented at the 24th IEEE CS Conference on Software Engineering Education and Training (CSEE&T 2011), Honolulu, HI, USA.
- Cunningham, W. (1992). *The WyCash Portfolio Management System*. Paper presented at the OOPSLA'92 conference, ACM. Retrieved from <http://c2.com/doc/oopsla92.html>
- Curtis, B., Sappidi, J., & Szyrkarski, A. (2012). Estimating the Principal of an Application's Technical Debt. *IEEE Software*, 29(6).
- Denne, M., & Cleland-Huang, J. (2004). *Software by Numbers: Low-Risk, High-Return Development*, Prentice Hall.
- Denne, M., & Cleland-Huang, J. (2004). The Incremental Funding Method: Data-Driven Software Development, *IEEE Software*, 21(3), 39-47.
- Fowler, M. (2009), *Technical debt quadrant*, Blog post at: <http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html>
- Gat, I. (ed.). (2010). *How to settle your technical debt--a manager's guide*. Arlington Mass: Cutter Consortium.
- Kruchten, Ph. (2010) Contextualizing Agile Software Development," Paper presented at the EuroSPI 2010 conference in Grenoble, Sept.1-3, 2010



# References

- Kruchten, P., Nord, R., & Ozkaya, I. (2012). Technical debt: from metaphor to theory and practice. *IEEE Software*, 29(6).
- Kruchten, P., Nord, R., Ozkaya, I., & Visser, J. (2012). Technical Debt in Software Development: from Metaphor to Theory--Report on the Third International Workshop on Managing Technical Debt, held at ICSE 2012 *ACM SIGSOFT Software Engineering Notes*, 37(5).
- Li, Z., Madhavji, N., Murtaza, S., Gittens, M., Miranskyy, A., Godwin, D., & Cialini, E. (2011). Characteristics of multiple-component defects and architectural hotspots: a large system case study. *Empirical Software Engineering*, 16(5), 667-702. doi: 10.1007/s10664-011-9155-y
- Lim, E. (2012). *Technical Debt: What Software Practitioners Have to Say*. (Master's thesis), University of British Columbia, Vancouver, Canada.
- Lim, E., Taksande, N., & Seaman, C. B. (2012). A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *IEEE Software*, 29(6).
- MacCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7), 1015-1030.
- Nord, R., Ozkaya, I., Kruchten, P., & Gonzalez, M. (2012). In search of a metric for managing architectural technical debt. Paper presented at the *Working IEEE/IFIP Conference on Software Architecture (WICSA 2012)*, Helsinki, Finland.
- McConnell, S. (2007) *Notes on Technical Debt*, Blog post at: <http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx>
- Special issue of *Cutter IT Journal* on Technical Debt, edited by I. Gat (October 2010) *Cutter IT Journal*, 23 (10).
- Sterling, C. (2010) *Managing Software Debt*, Addison-Wesley.





# References (cont.)

- R. O. Spinola, N. Zazworka, A. Vetrò, C. B. Seaman, and F. Shull, "Investigating Technical Debt Folklore: Shedding Some Light on Technical Debt Opinion," in Proceedings of the 4th Workshop on Managing Technical Debt, at ICSE 2013, P. Kruchten, I. Ozkaya, and R. Nord, Eds., IEEE, 2013.
- K. Schmid, "On the Limits of the Technical Debt Metaphor," in Proceedings of the 4th Workshop on Managing Technical Debt, at ICSE 2013, P. Kruchten, I. Ozkaya, and R. Nord, Eds., IEEE, 2013, pp. 63-66.
- K. Schmid, "A Formal Approach to Technical Debt Decision Making," in Proceedings of the Conference on Quality of Software Architecture QoSA'2013, Vancouver, 2013, ACM.
- Avgeriou, P., Kruchten, P., Ozkaya, I., & Seaman, C. (eds) "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)". Dagstuhl Reports (Vol. 6, issue 4 pp. 110-138). Dagstuhl, Germany: Schloss Dagstuhl--Leibniz-Zentrum für Informatik.

# Other sources (Talks/slides)

- Gat, I., Heintz, J. (Aug. 19, 2010) *Webinar: Reining in Technical Debt*, Cutter Consortium.
- McConnell, S. (October 2011) *Managing technical debt*. (Webinar)
- Kniberg, H. (2008) *Technical debt-How not to ignore it*, at Agile 2008 conference.
- Kruchten, P. (2009) *What colour is your backlog?* Agile Vancouver Conference. <http://philippe.kruchten.com/talks>
- Sterling, C. (2009) <http://www.slideshare.net/csterwa/managing-software-debt-pnsgc-2009>
- Short, G. (2009) <http://www.slideshare.net/garyshort/technical-debt-2985889>
- West, D. (January 2011), *Balancing agility and technical debt*, Forrester & Cast Software



# 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

