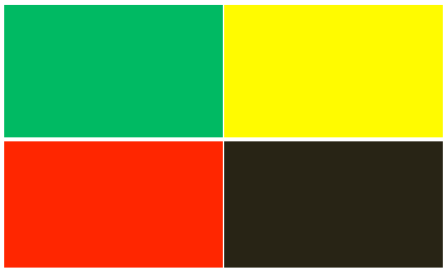


What colours is your backlog?

Philippe Kruchten

1

Philippe Kruchten, Ph.D., P.Eng.

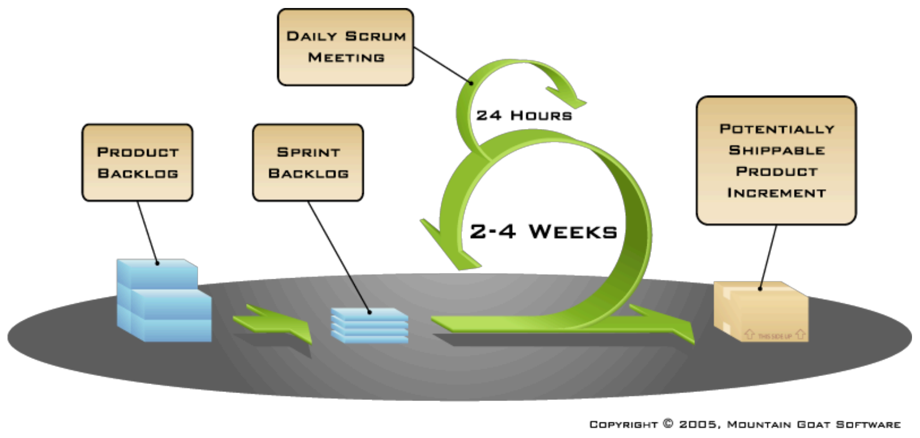
Professor of Software Engineering
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, BC Canada
pbk@ece.ubc.ca

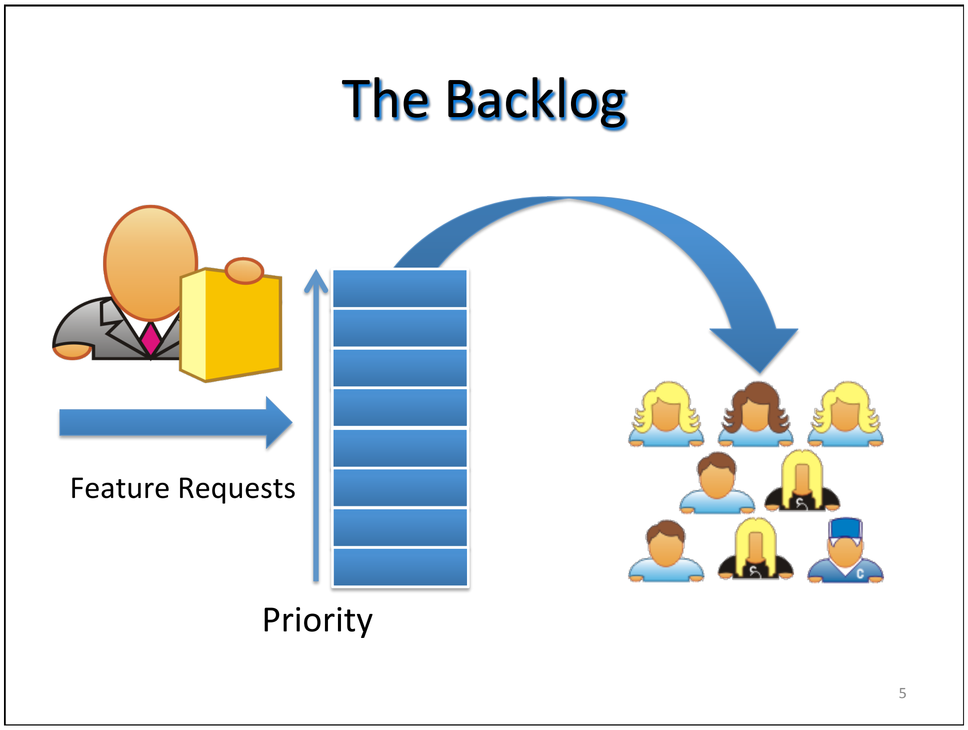
2

CPEN 321 -Oct 31st - Outline

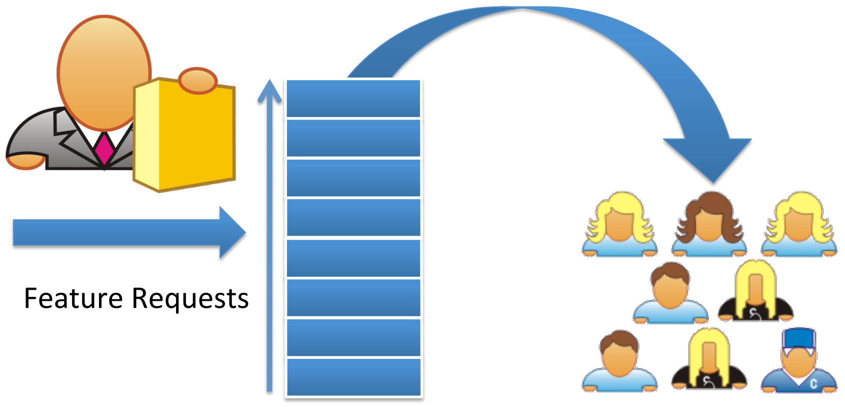
- Scrum workflow
- Backlog
- Managing a complete backlog
- Technical debt
- Managing technical debt

Scrum

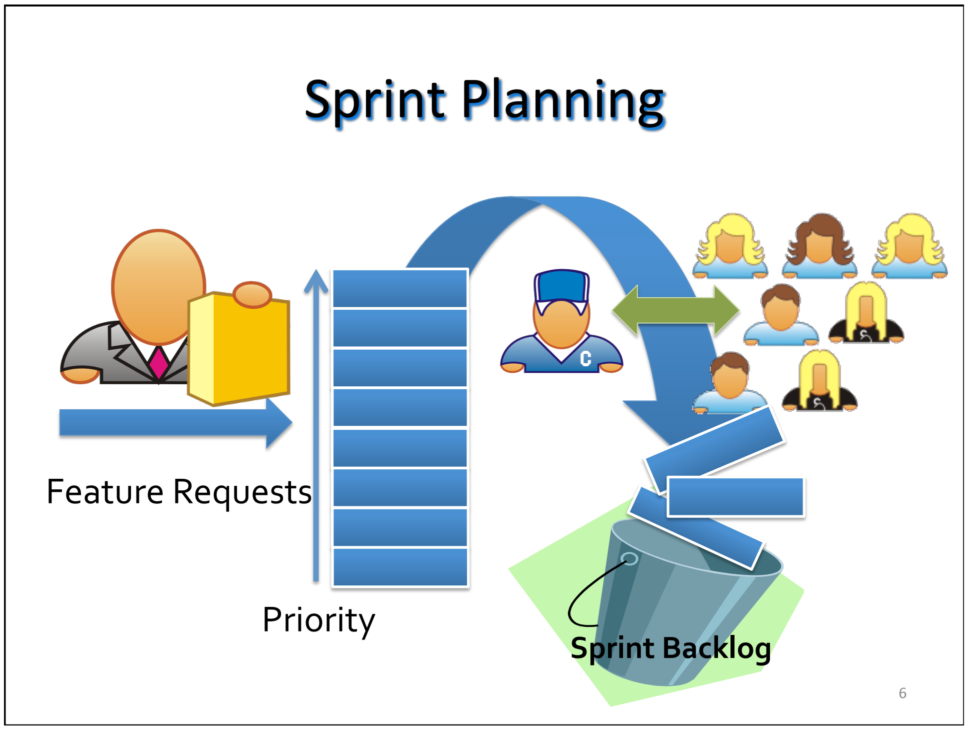

Daily Scrum Meeting
24 Hours
Product Backlog
Sprint Backlog
2-4 Weeks
Potentially Shippable Product Increment
Copyright © 2005, Mountain Goat Software



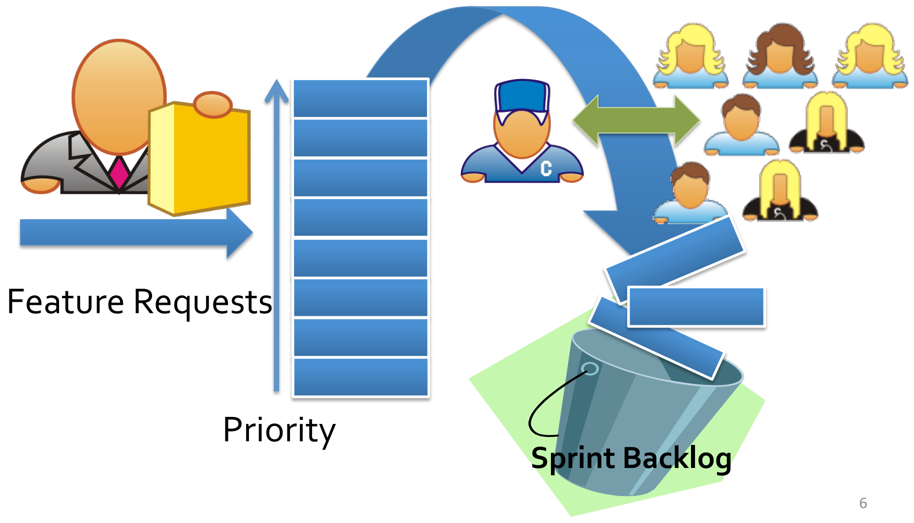
The Backlog



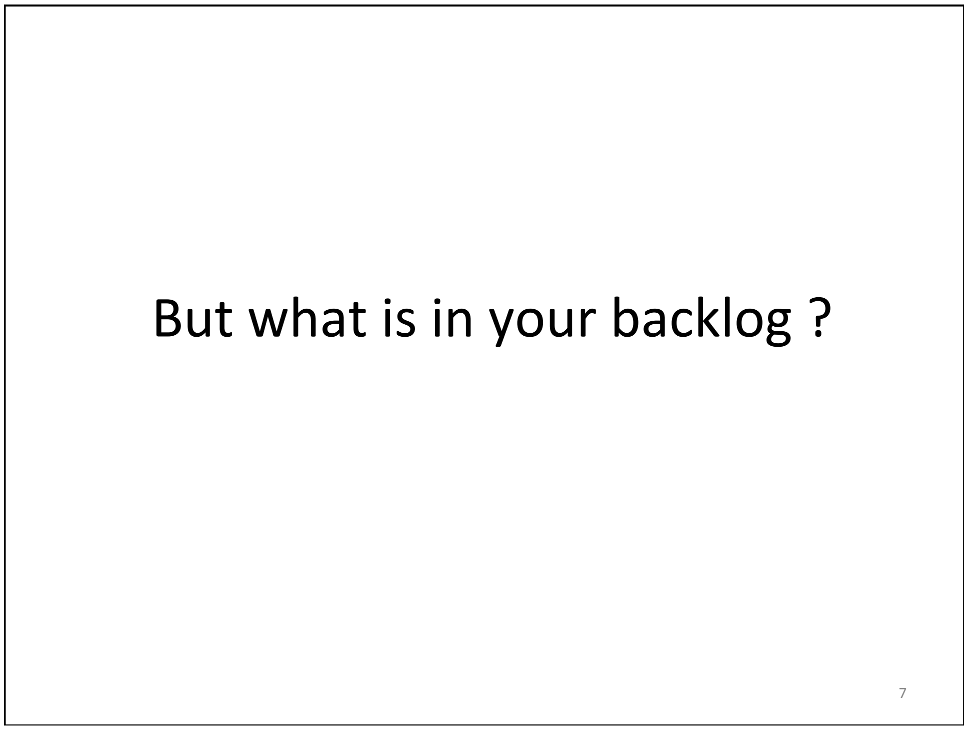
5



Sprint Planning

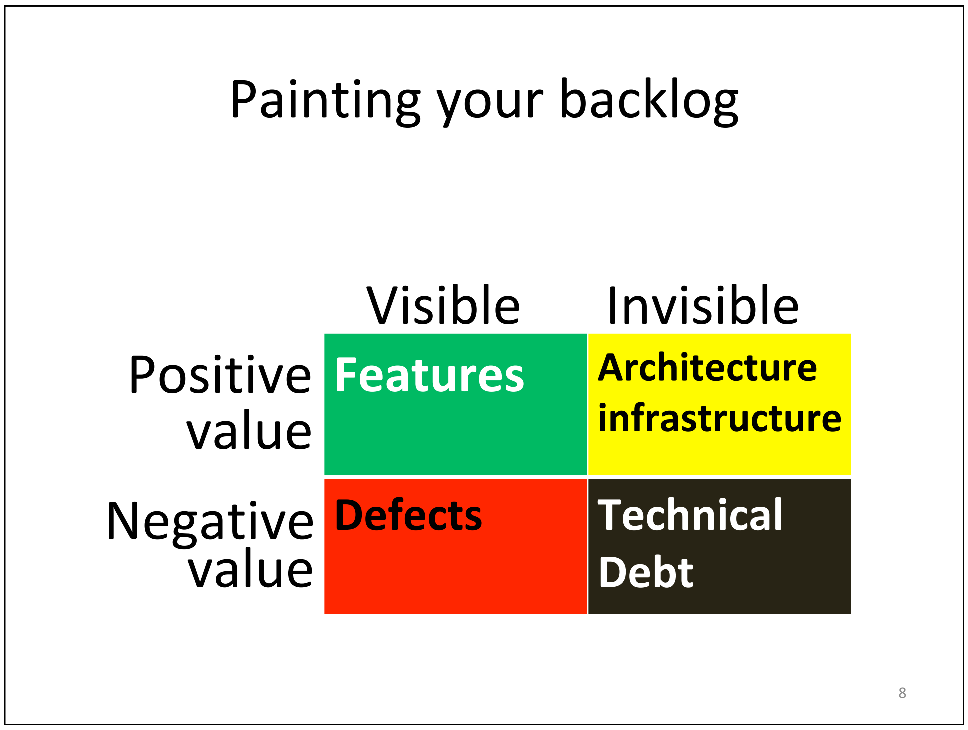


6



But what is in your backlog ?

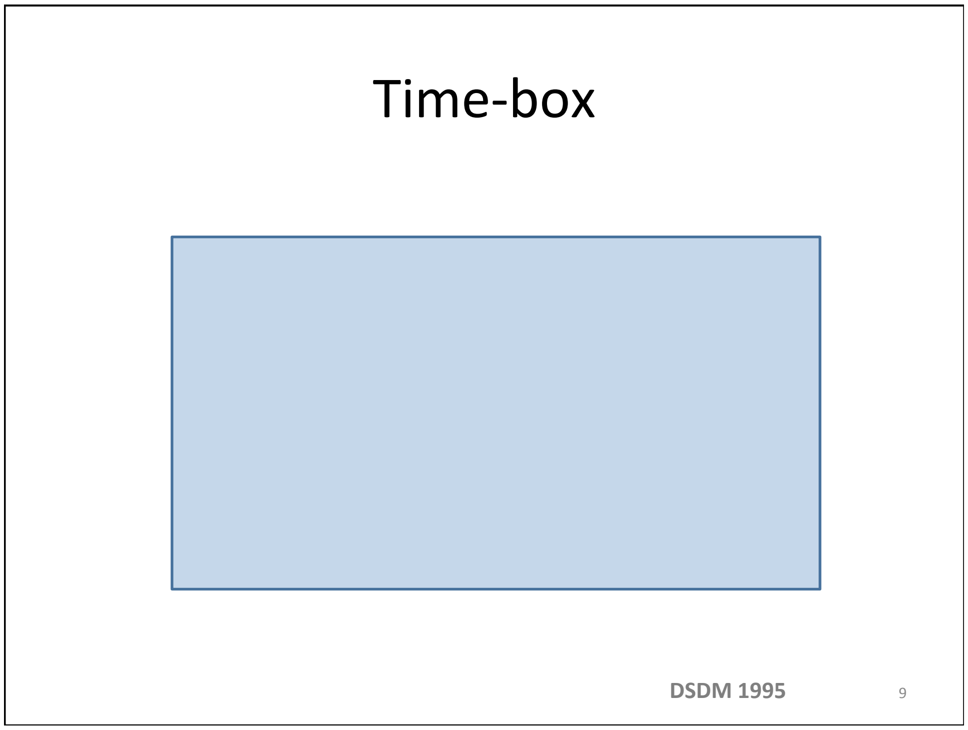
7



Painting your backlog

	Visible	Invisible
Positive value	**Features**	**Architecture infrastructure**
Negative value	**Defects**	**Technical Debt**

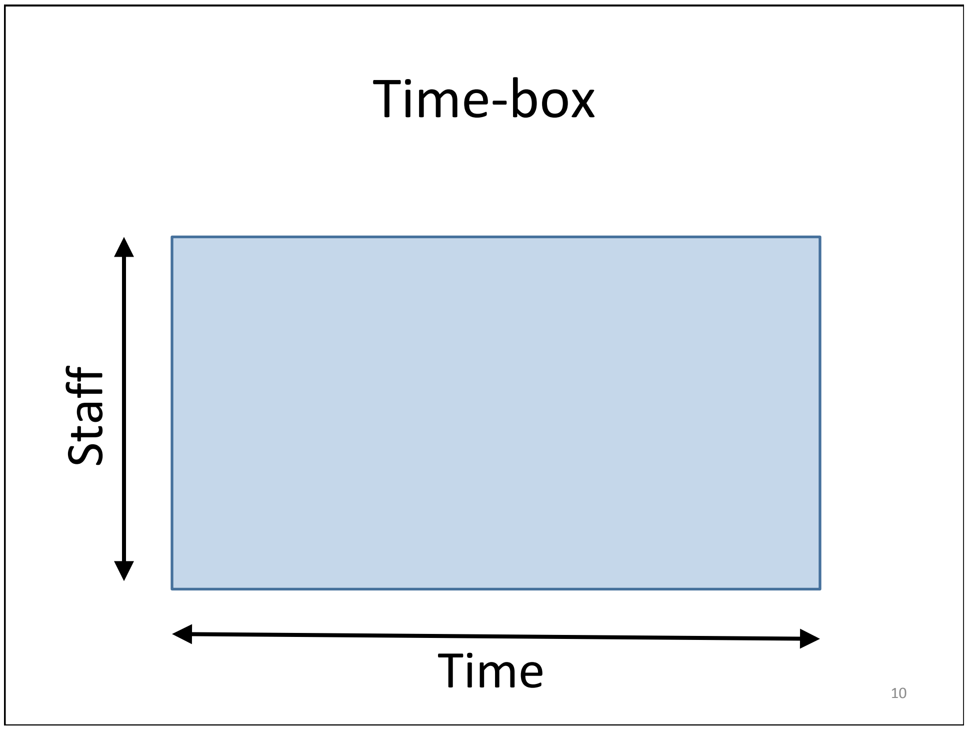
8



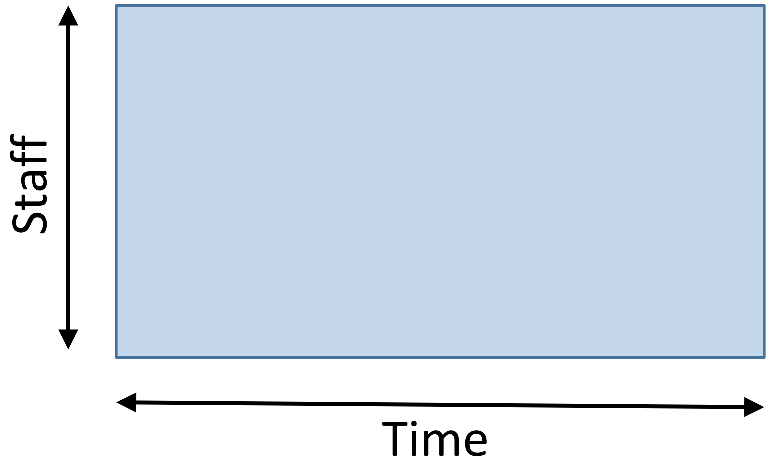
Time-box

DSDM 1995

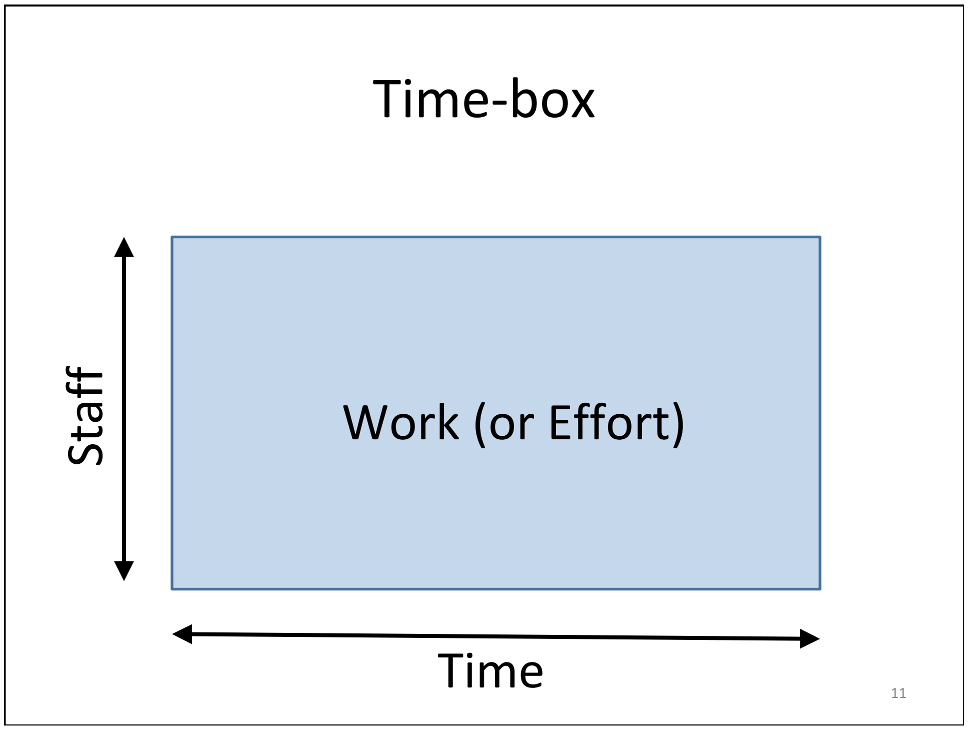
9



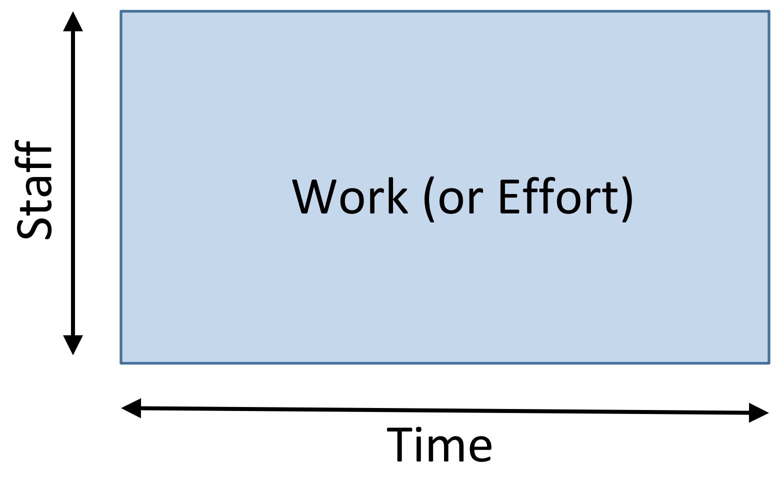
Time-box



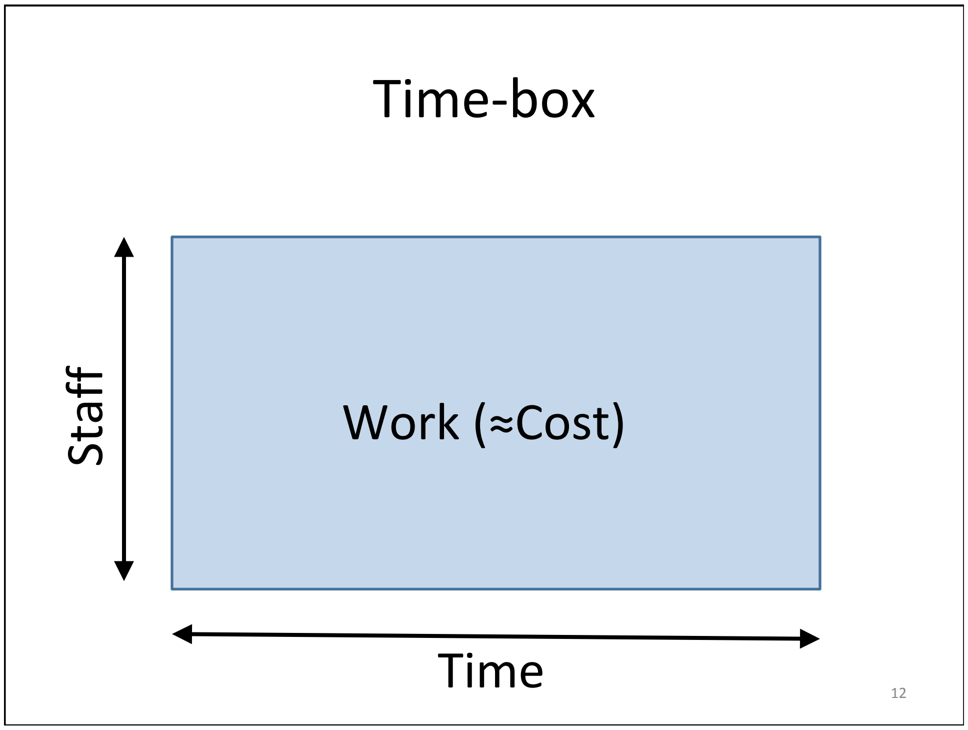
10



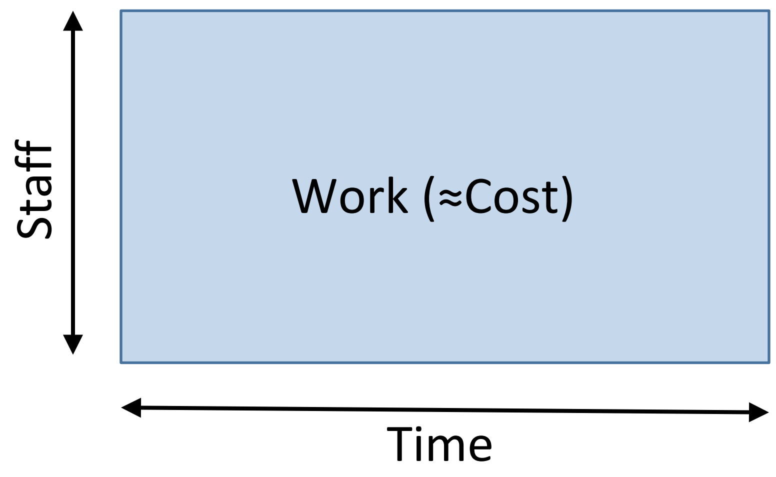
Time-box



11

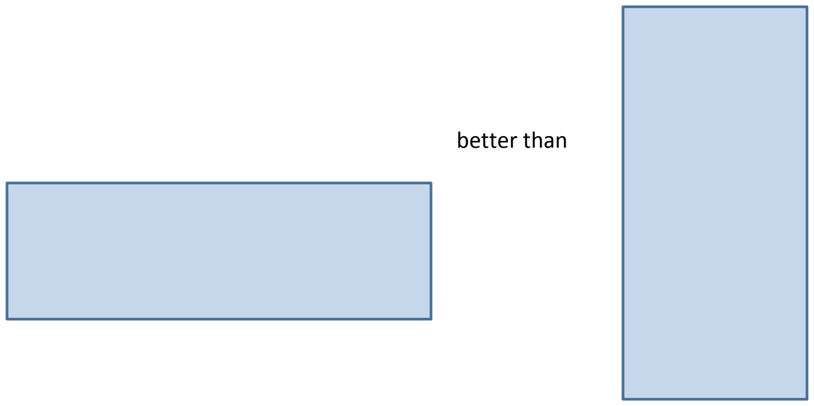


Time-box



12

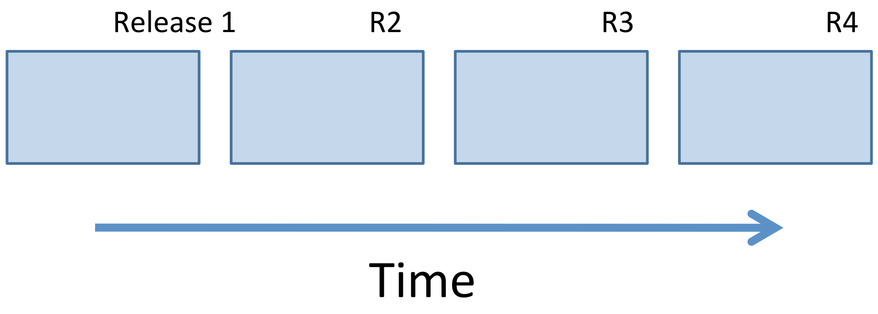
Time-box



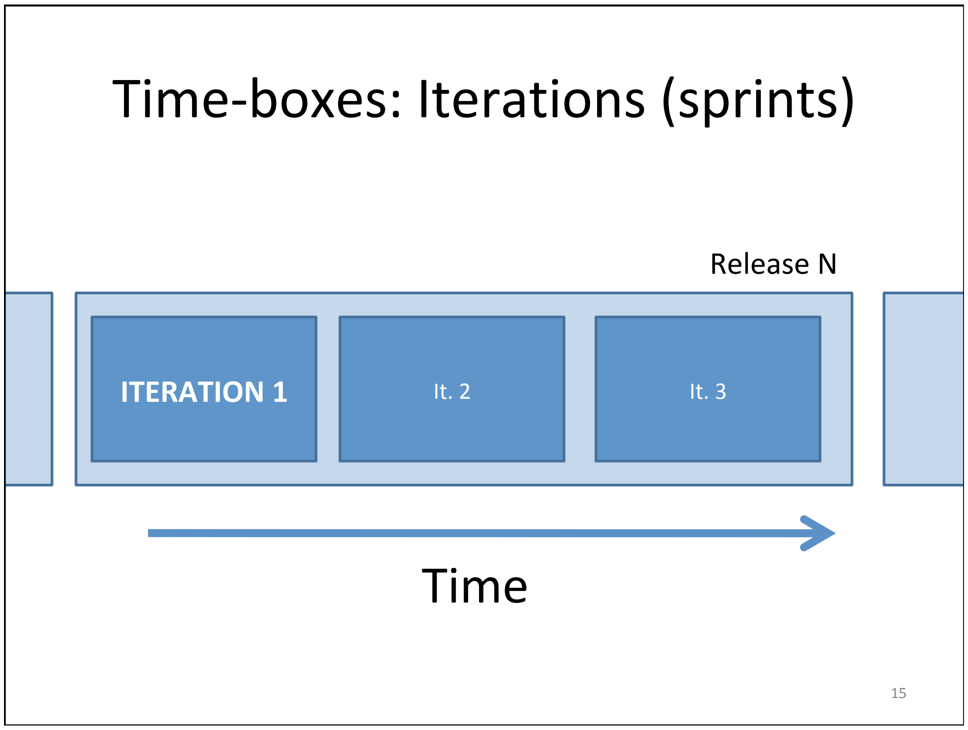
Brooks, Mythical Man-Month, 1975
Boehm, COCOMO, 1981

13

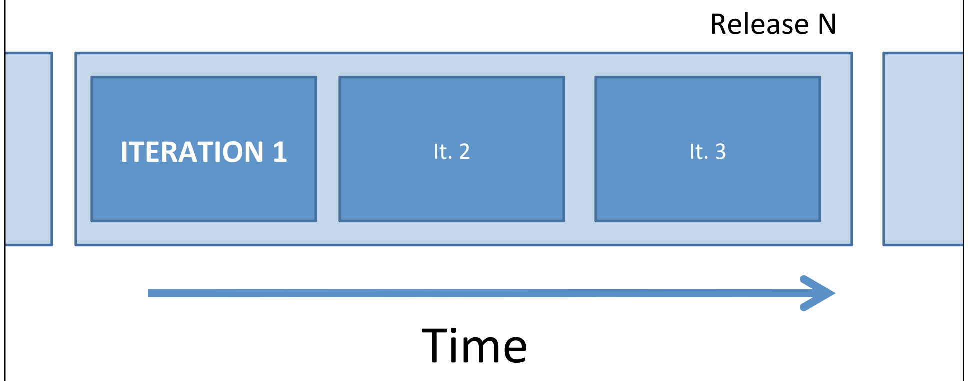
Time-boxes: Releases



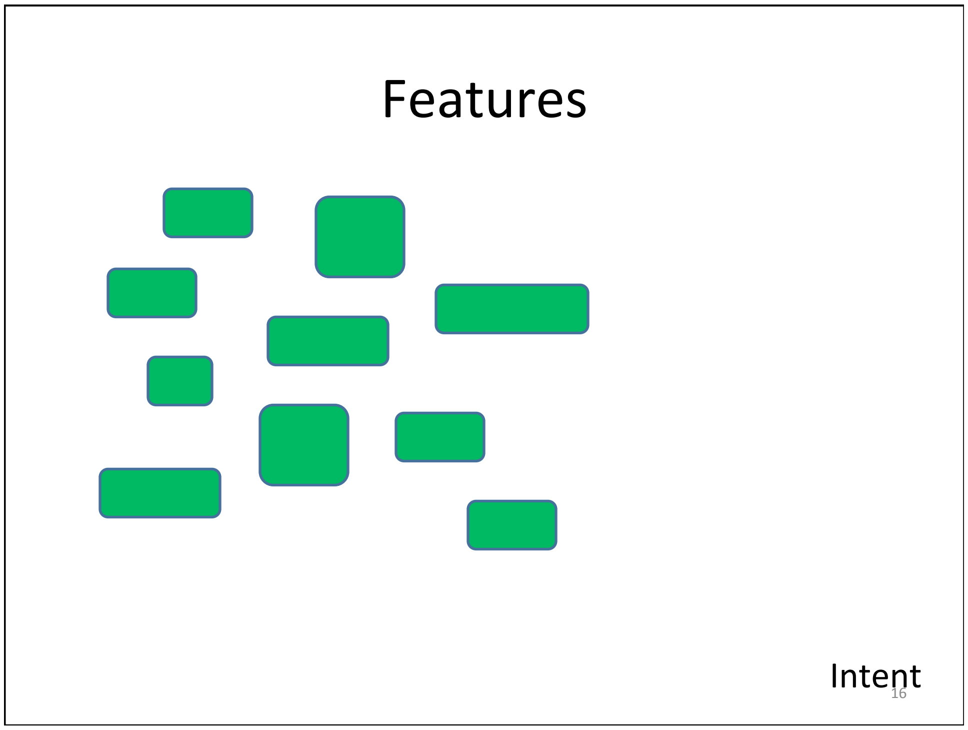
14



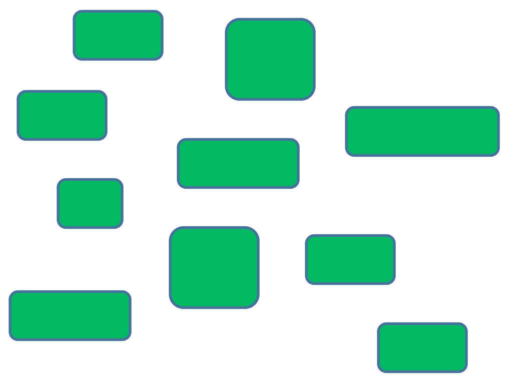
Time-boxes: Iterations (sprints)



15

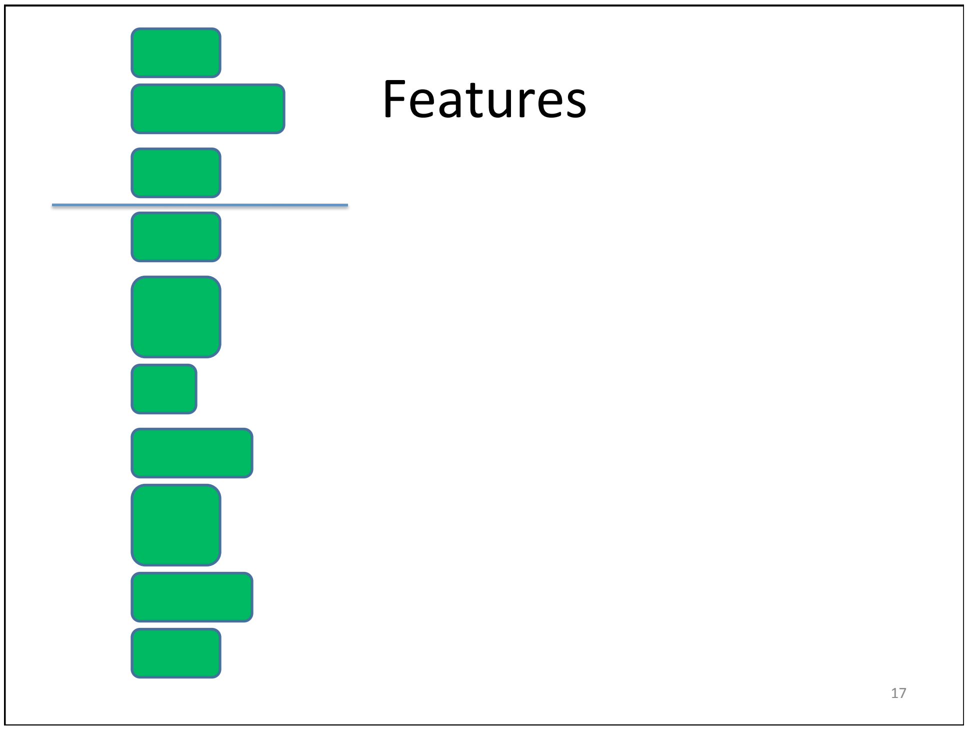


Features



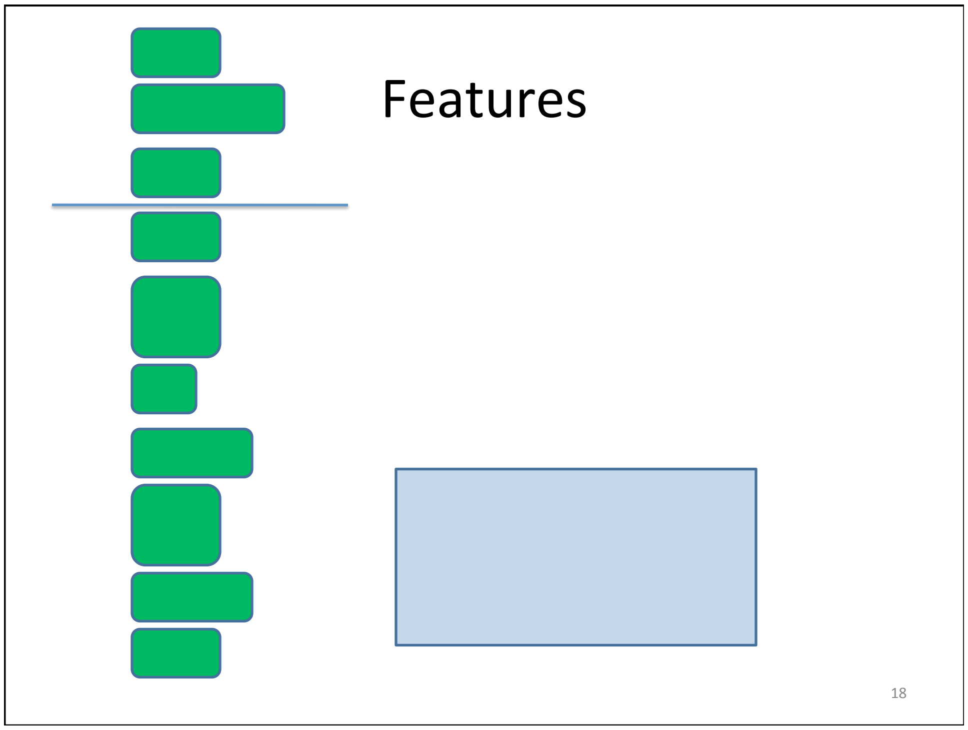
Intent

16



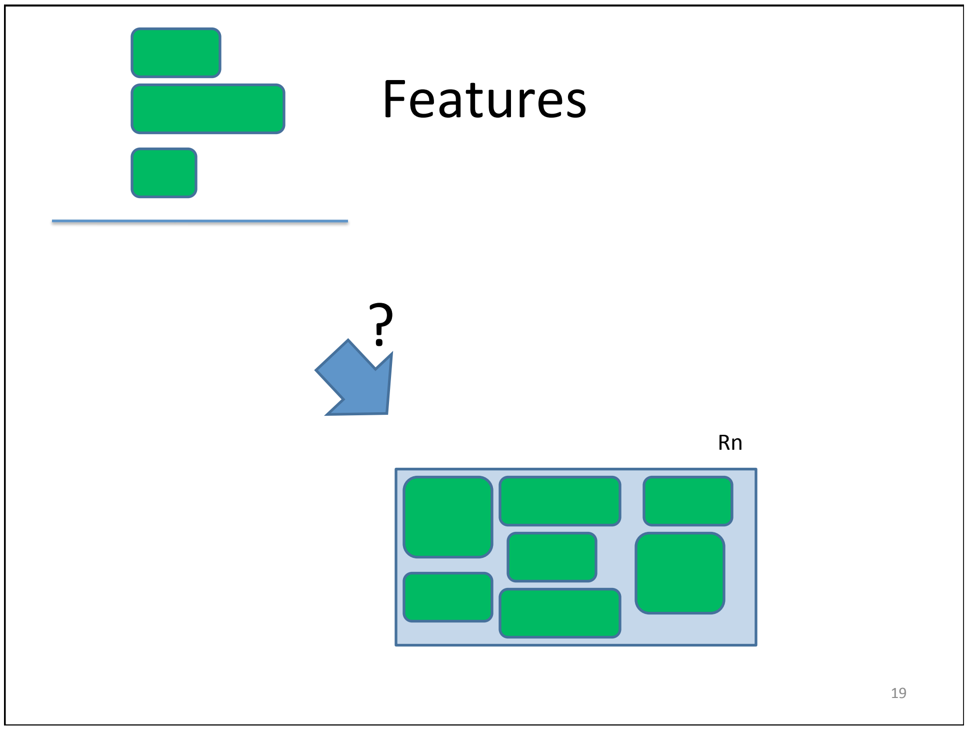
Features

17

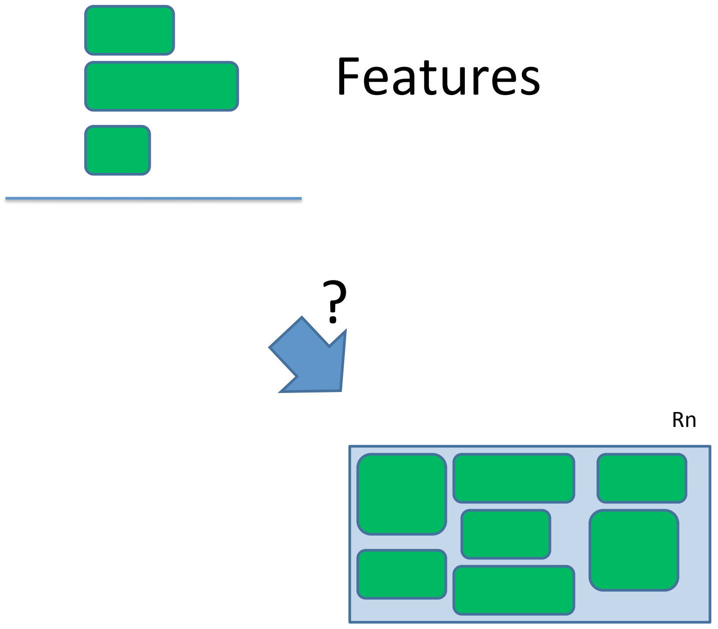


Features

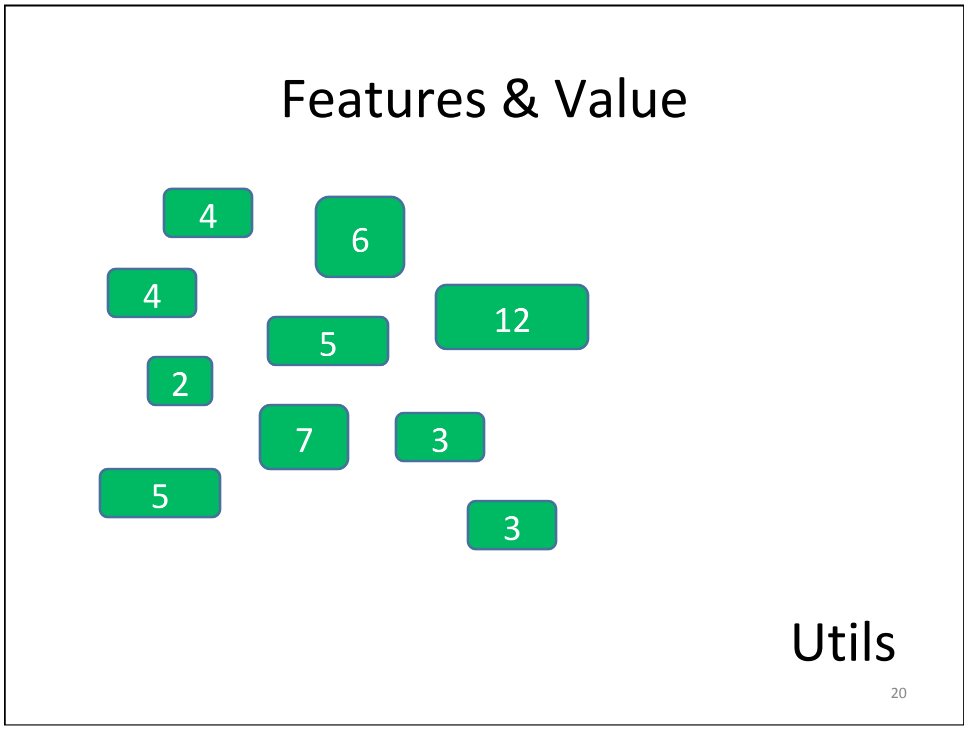
18



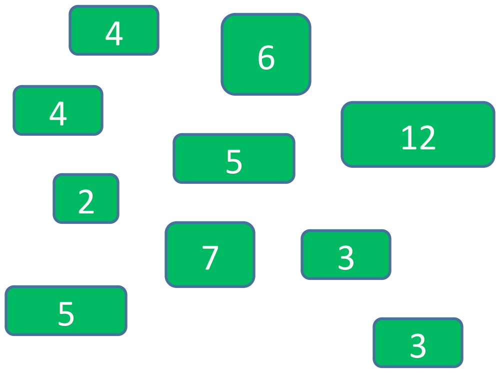
Features



19



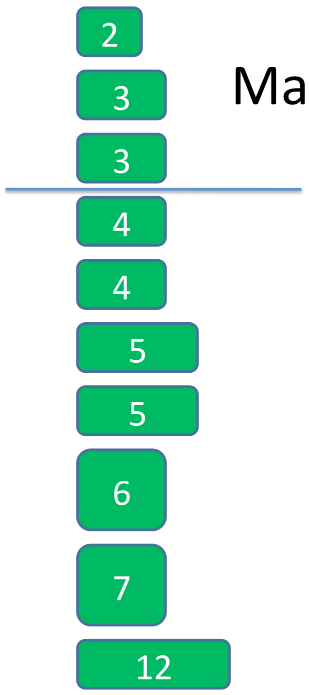
Features & Value



Utils

20

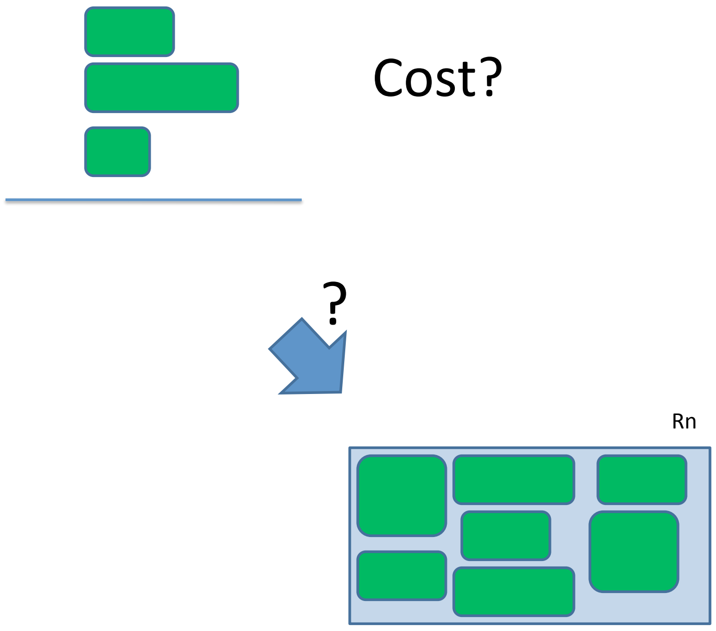
Maximizing value



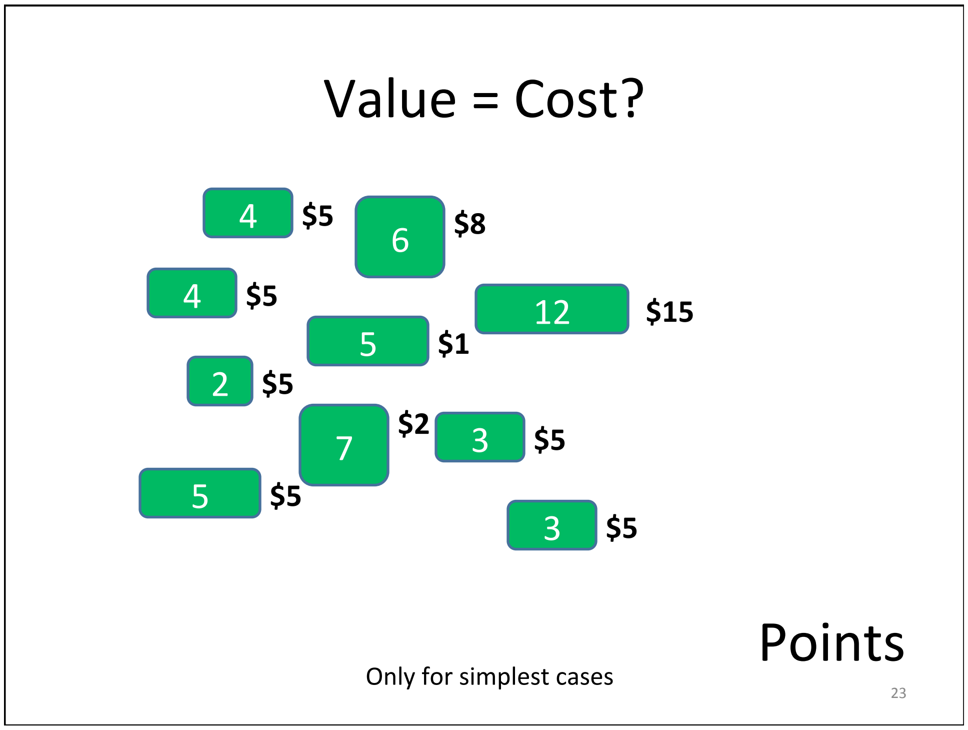
Highest value first
Ignore time

21

Cost?



22

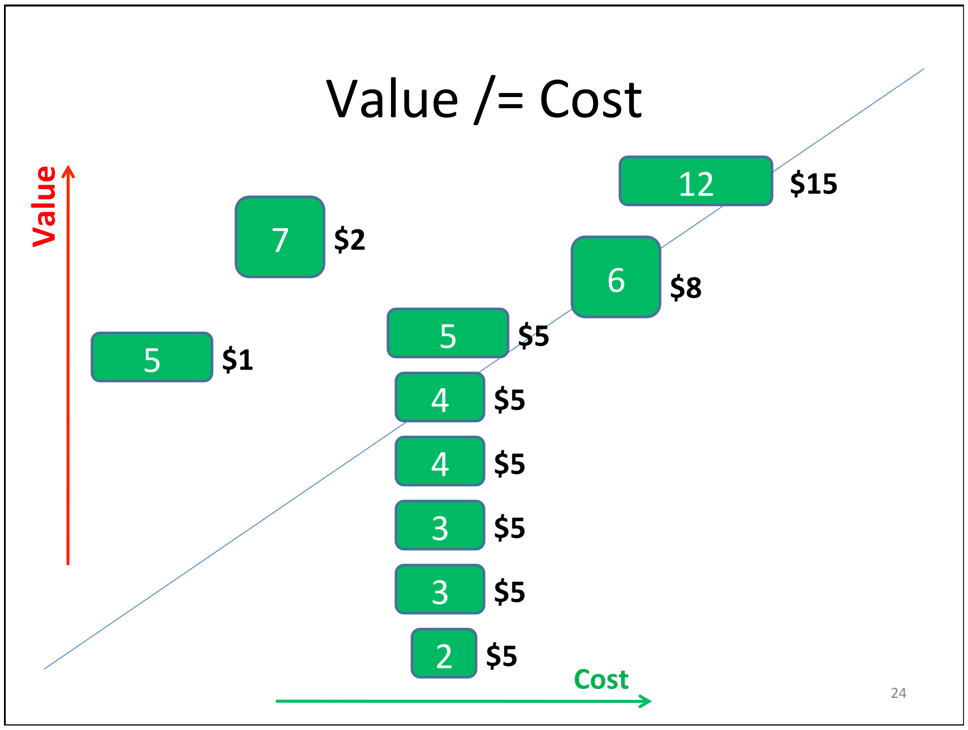


Value = Cost?

Points

Only for simplest cases

23



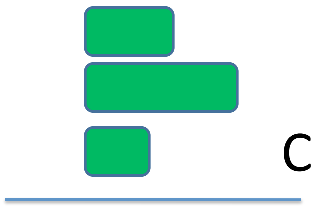
Value /= Cost

24

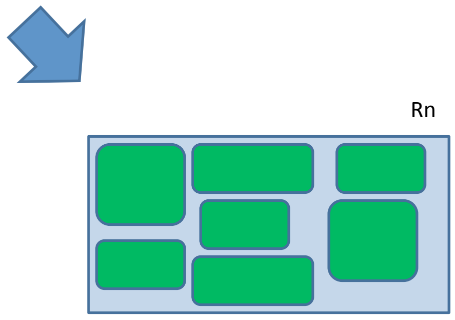
Value and Cost

- Value: to the business (the users, the customers, the public, etc.)
- Cost: to design, develop, manufacture, deploy, maintain
- Simple system, stable architecture, many small features:
  - Statistically value aligns to cost
- Large, complex, novel systems ?

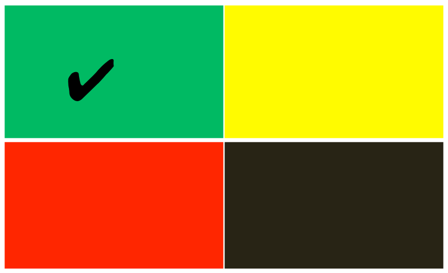
25



Cost will impose the limit



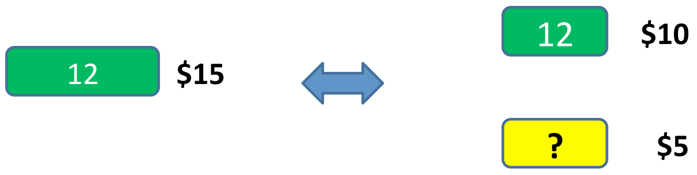
28

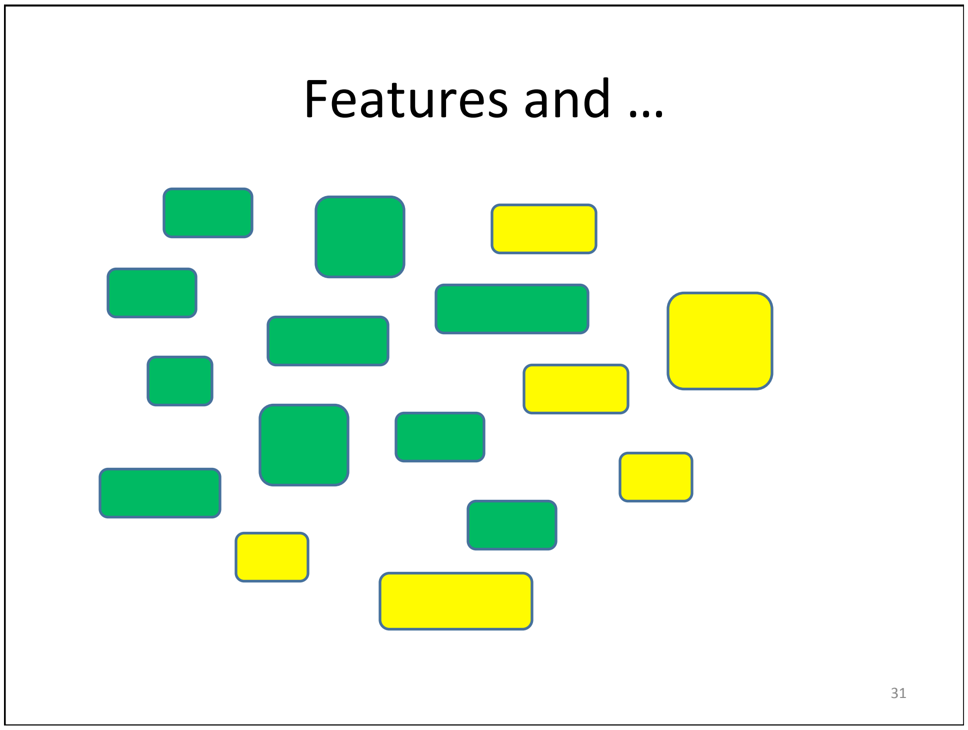


What colour is your backlog?

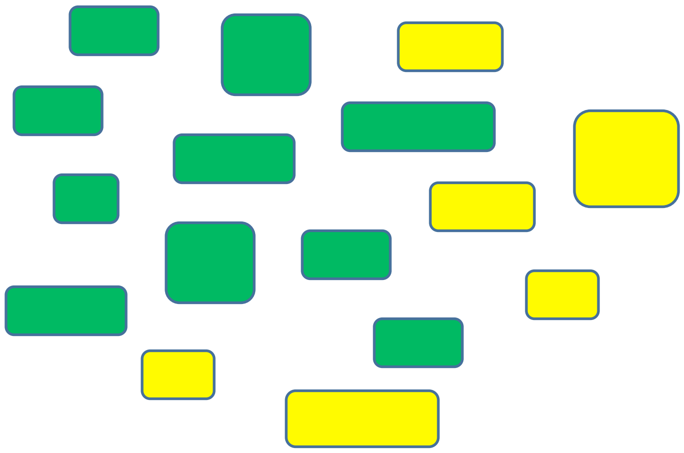
(so far)

Invisible Features

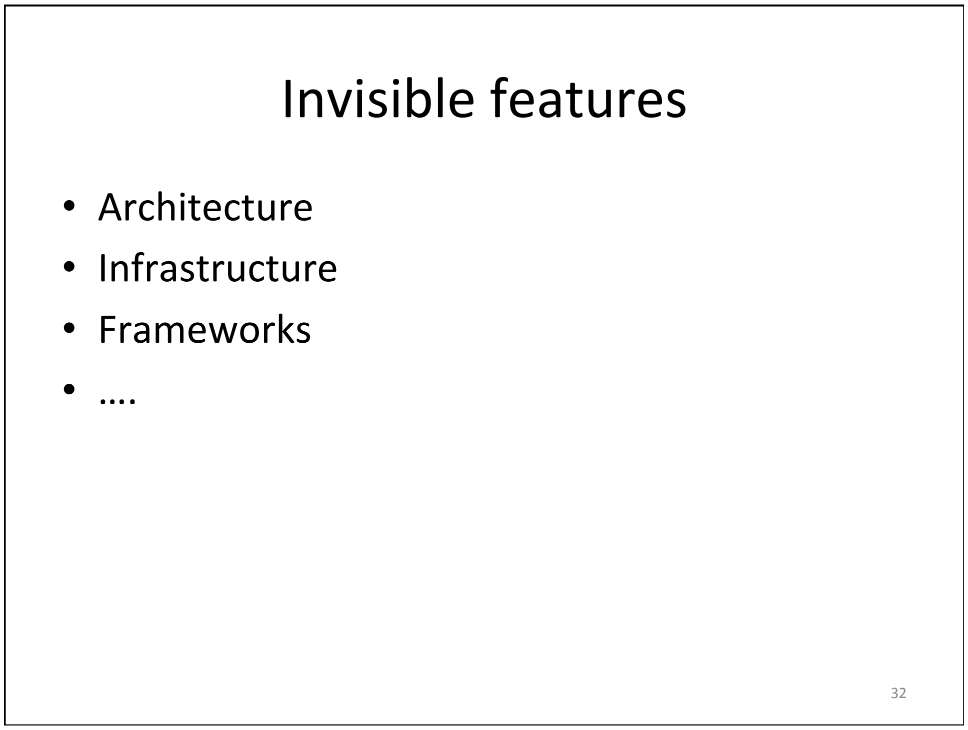




Features and …



31

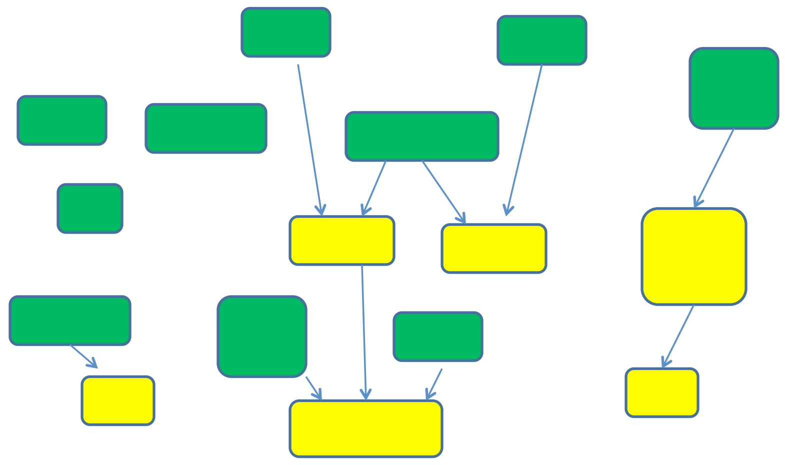


Invisible features

- Architecture
- Infrastructure
- Frameworks
- ….

32

Dependencies

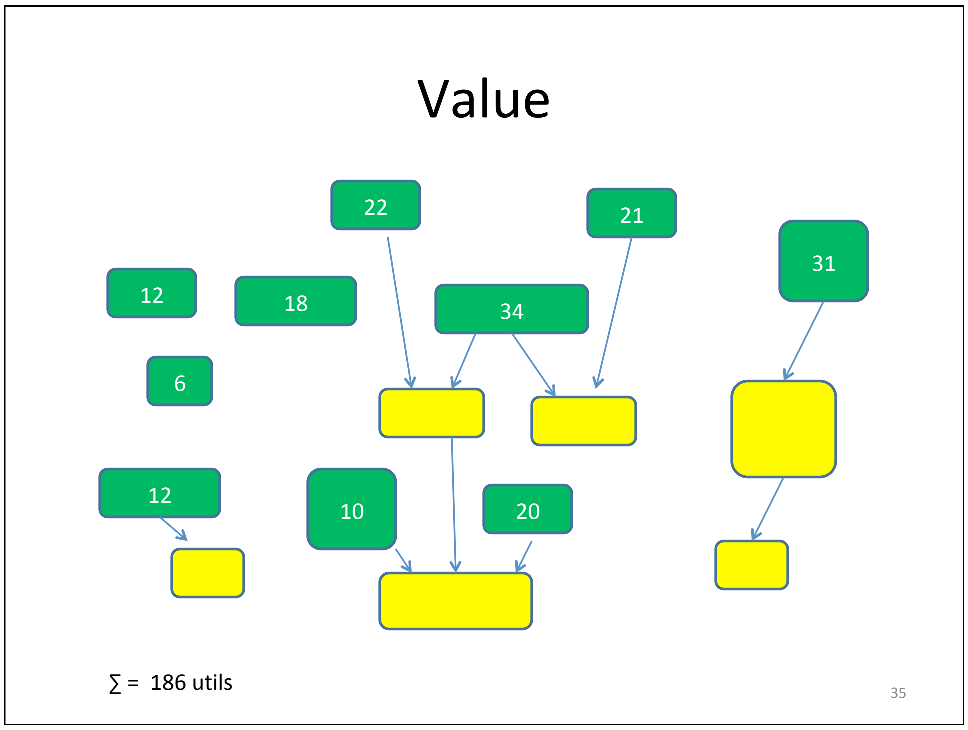


33

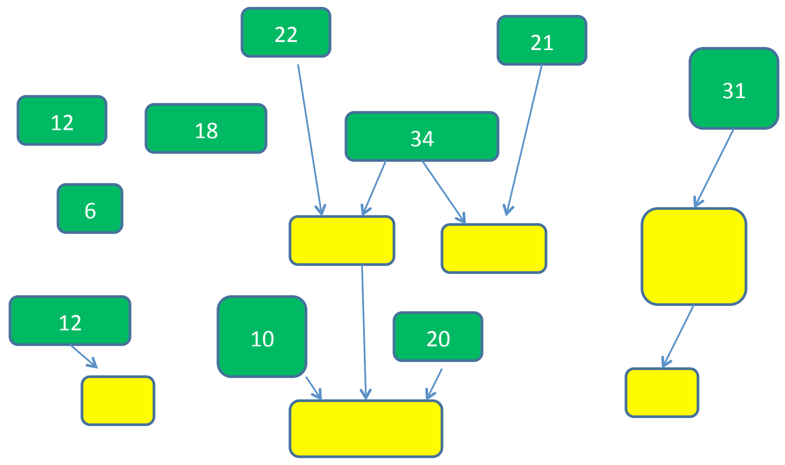
Architecture: Value and Cost

- Architecture has no (or little) externally visible "customer value"
- Iteration planning (backlog) is driven solely (?) by "customer value"
- YAGNI, BUFD, Metaphor,…
- "Last responsible moment!"
- We'll refactor it later!
- *Ergo:* architectural activities are not given proper attention
- *Ergo:* large technical debts

34

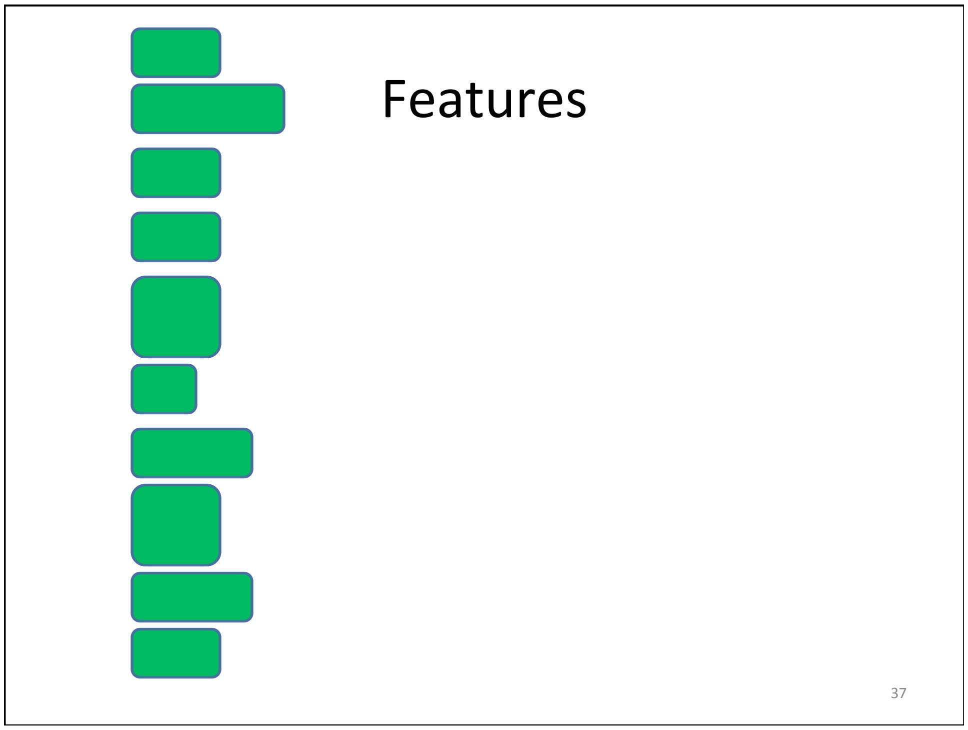


Value

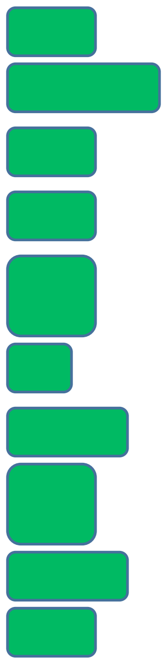


∑ = 186 utils

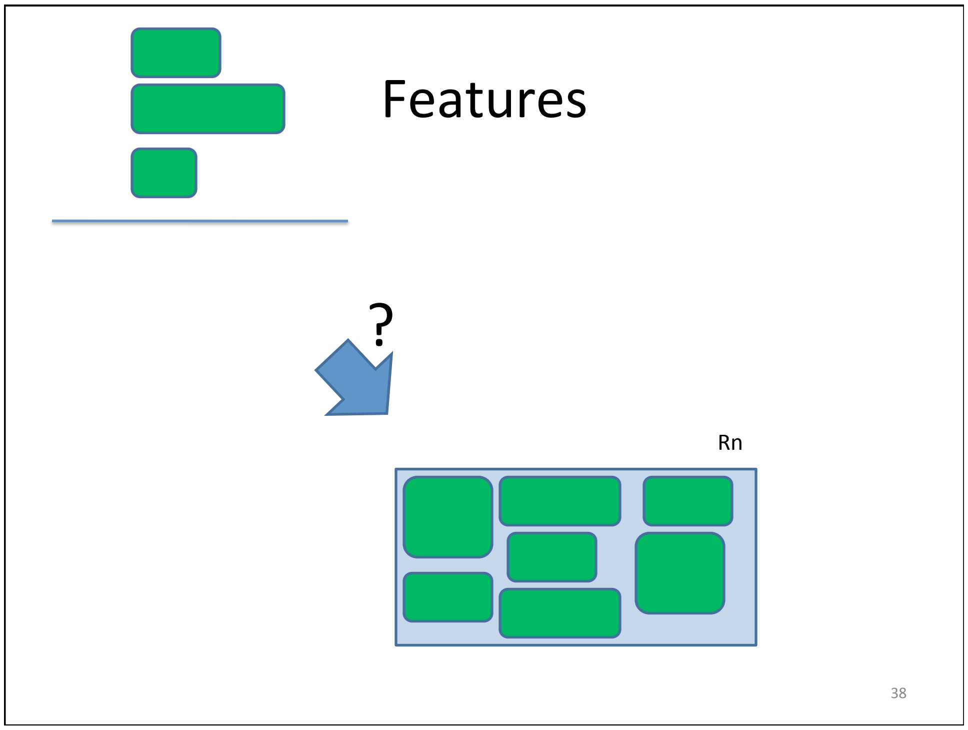
35



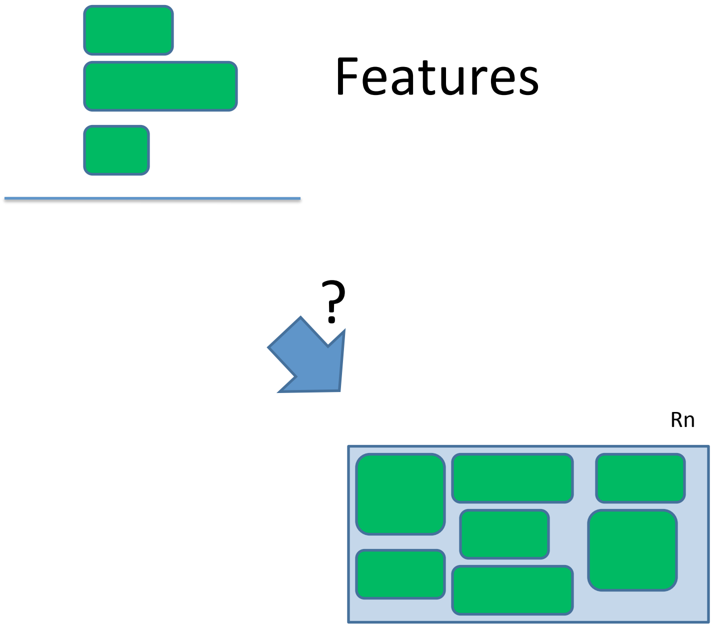
Features



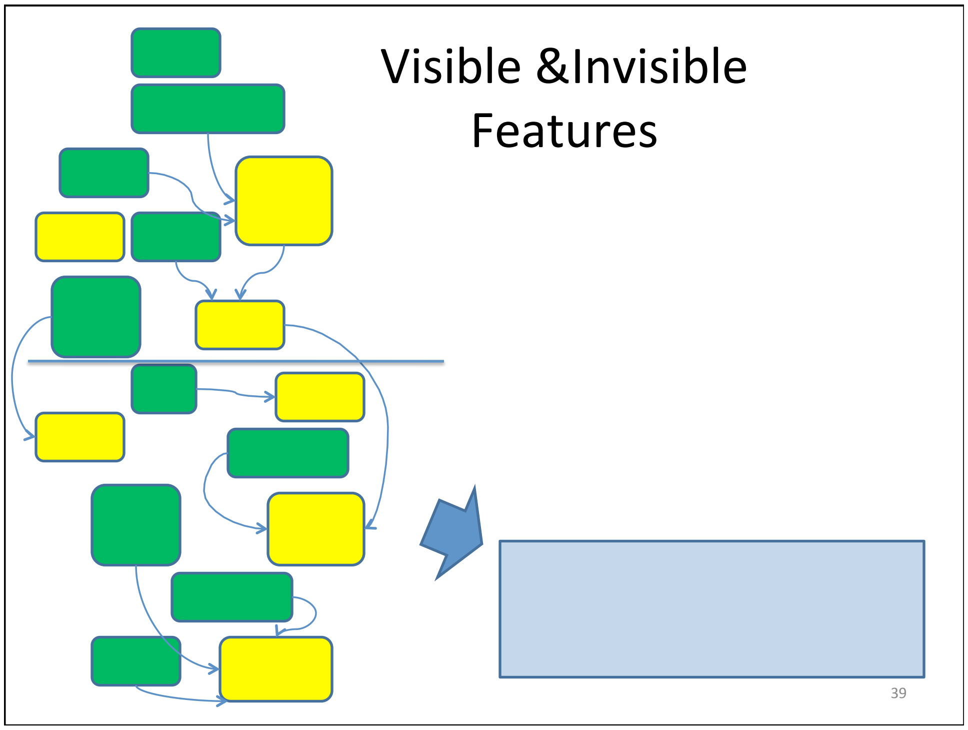
37



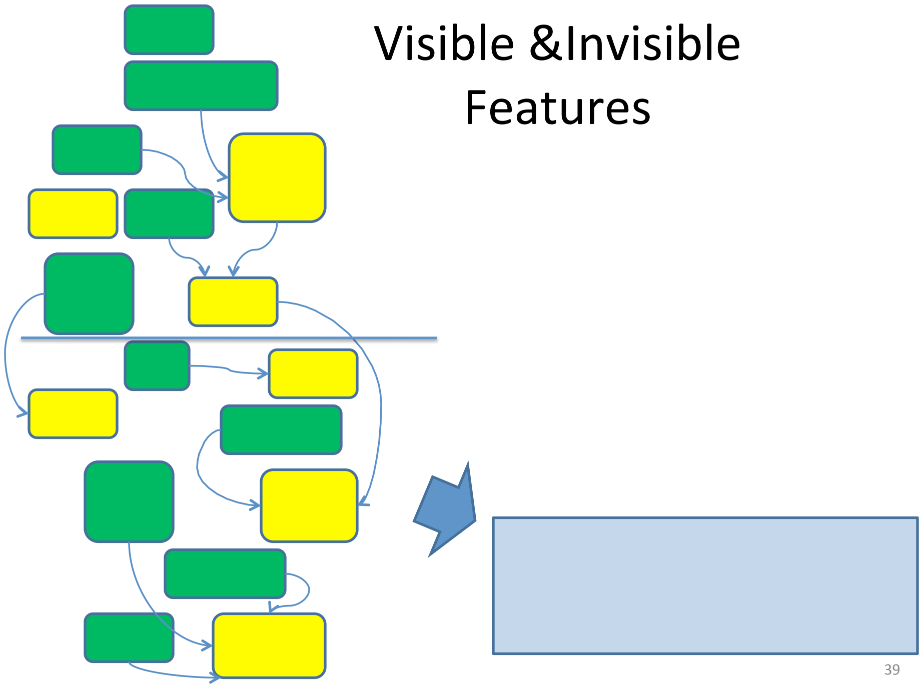
Features



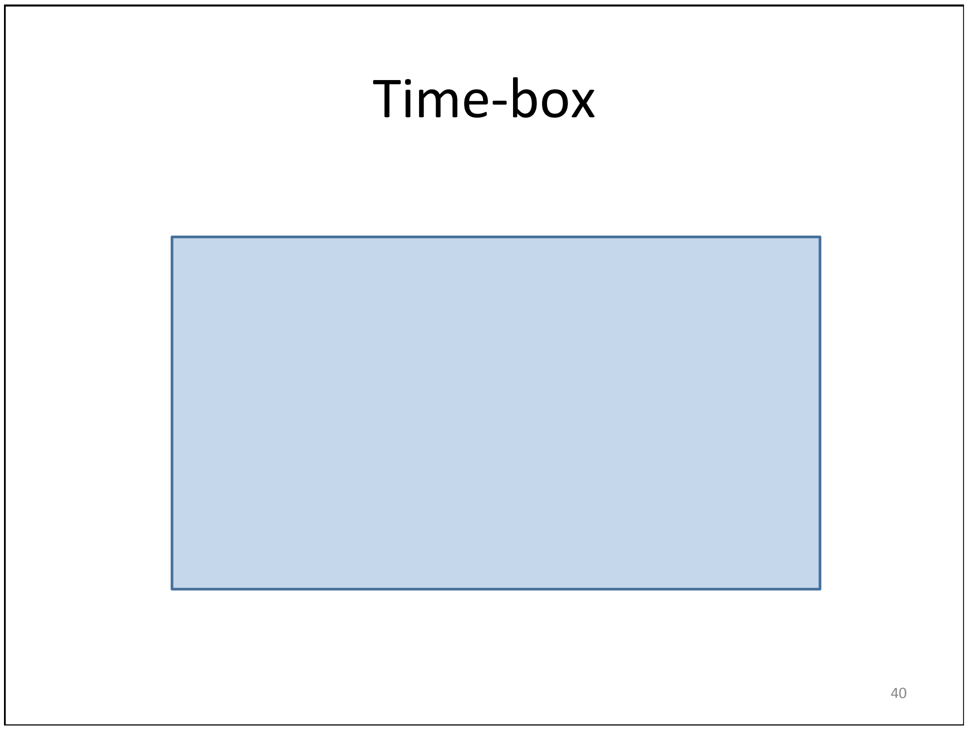
38



Visible &Invisible Features

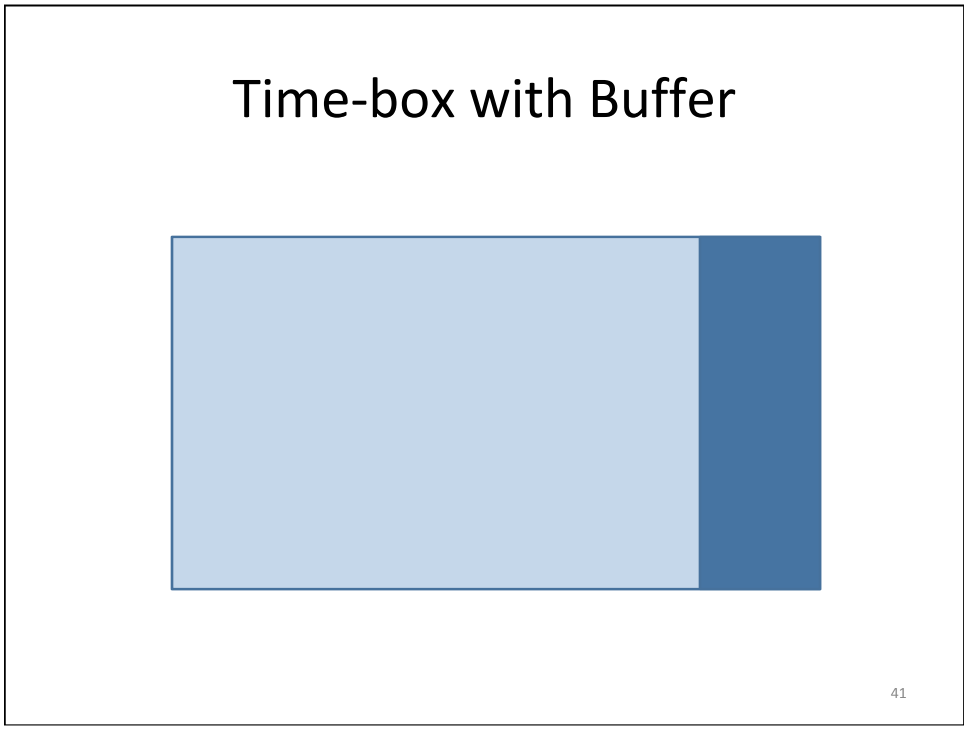


39

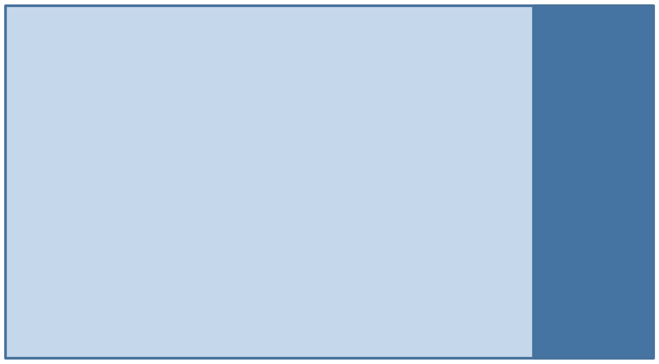


Time-box

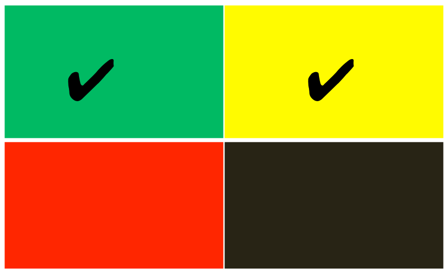
40



Time-box with Buffer



41



What colour is your backlog?

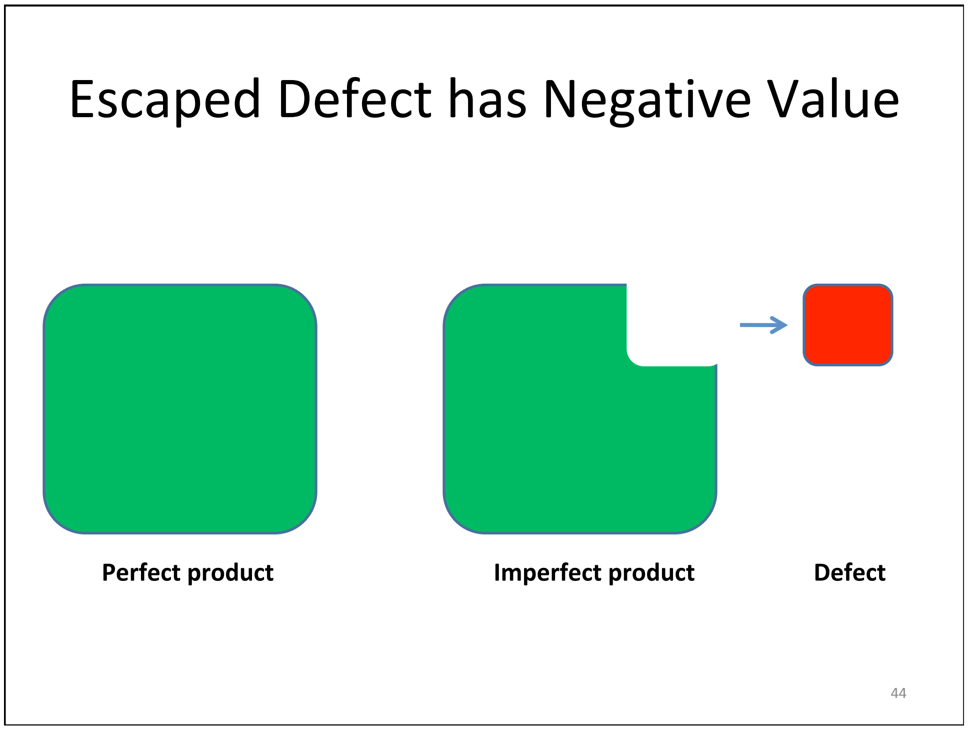
(so far)

42

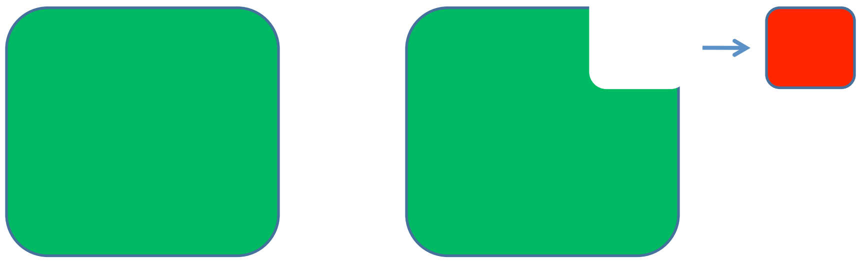
Defects

- Defect = Feature with negative value
- Fixing a defect has a positive cost (= work)
- Time/place of discovery
  - Inside development (in-house, in process)
  - Outside development in a released product (escaped defects)

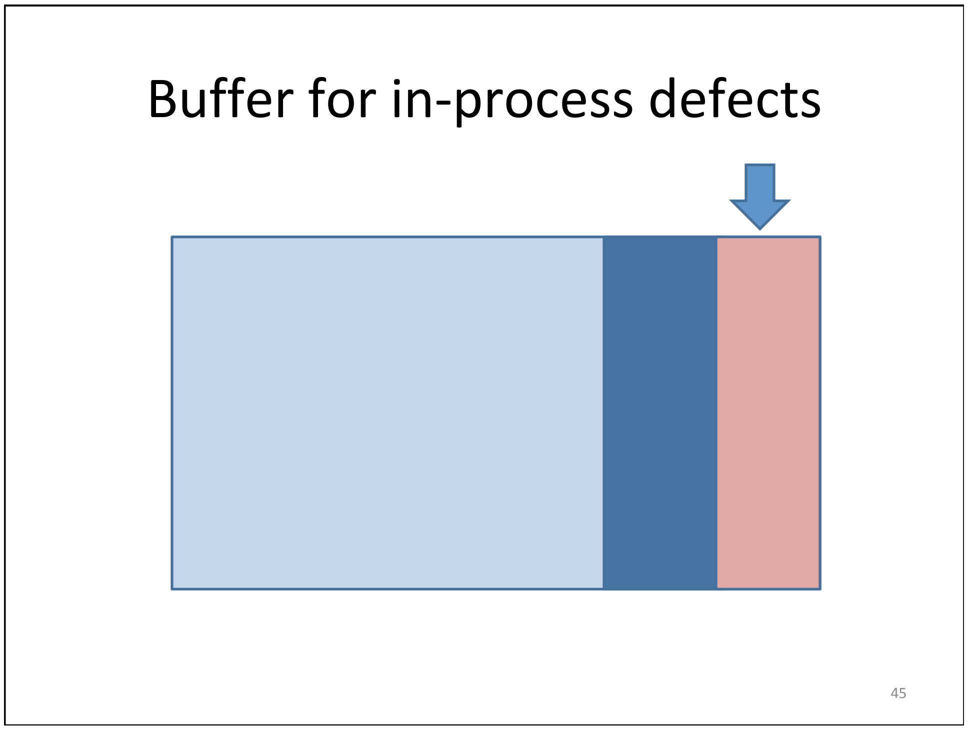
43



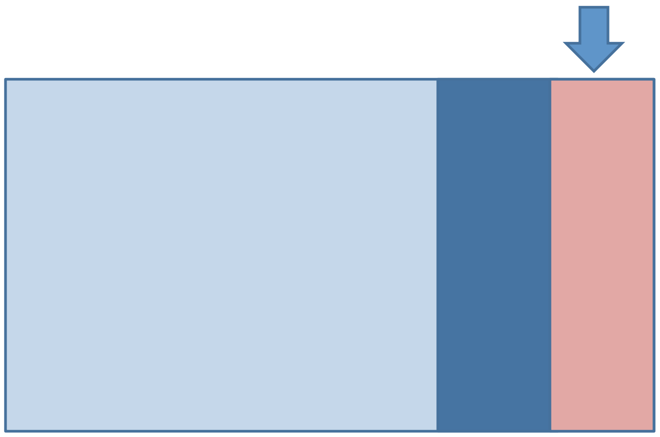
Escaped Defect has Negative Value



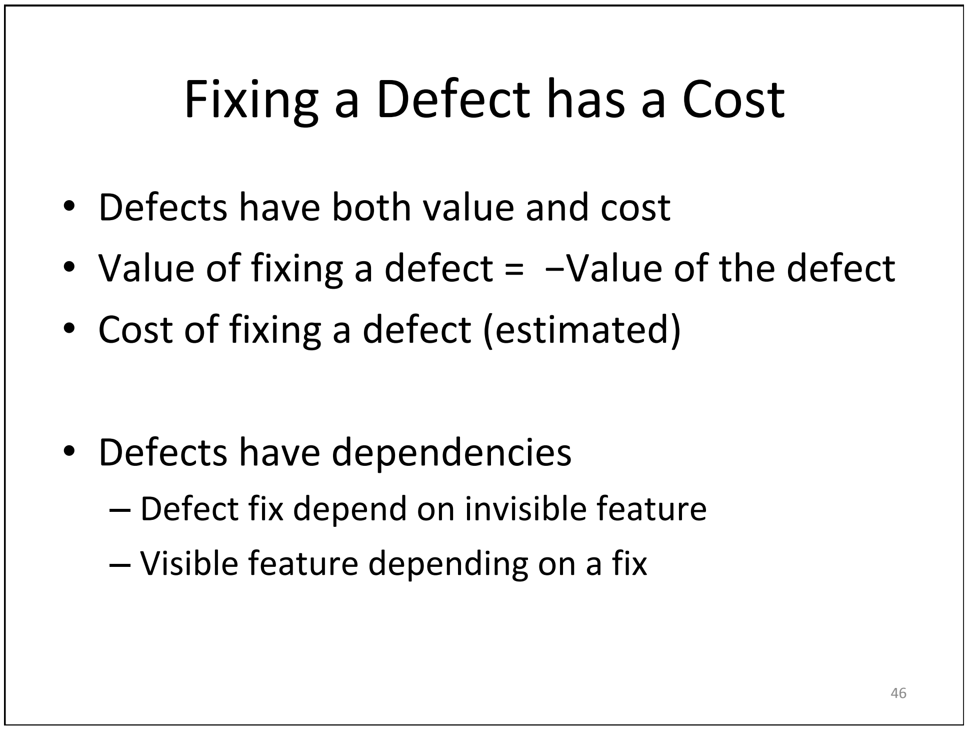
44



Buffer for in-process defects



45

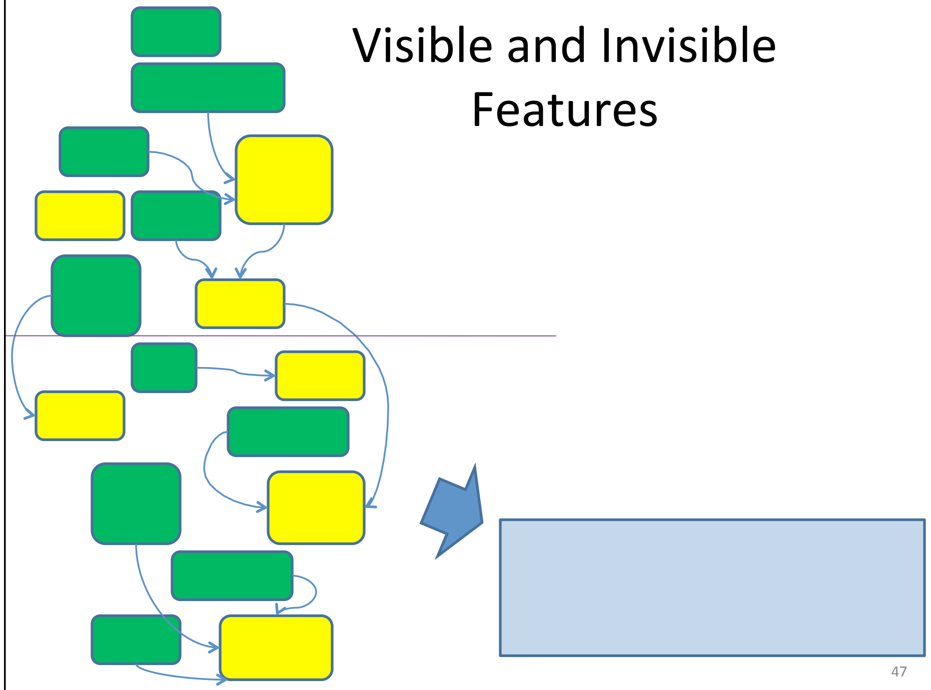


Fixing a Defect has a Cost

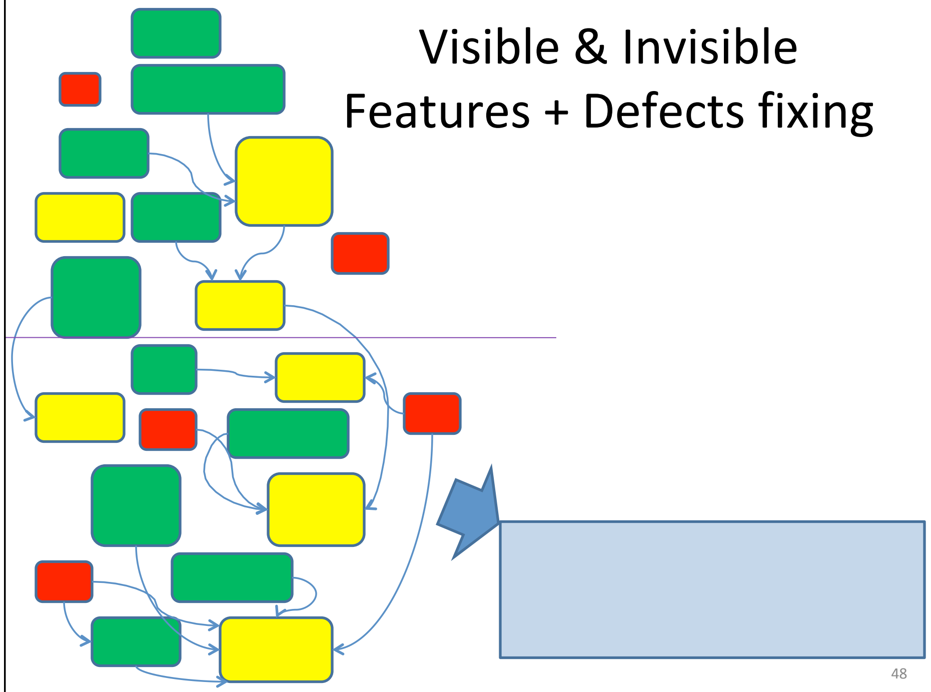
- Defects have both value and cost
- Value of fixing a defect = −Value of the defect
- Cost of fixing a defect (estimated)

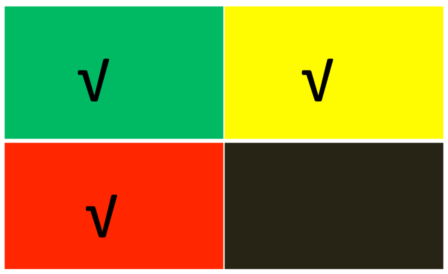
- Defects have dependencies
  - Defect fix depend on invisible feature
  - Visible feature depending on a fix

Visible and Invisible Features



Visible & Invisible Features + Defects fixing





What colour is your backlog?

(so far)

Technical Debt

- Concept introduced by Ward Cunningham
- Often mentioned, rarely studied
- All experienced SW developers “feel” it.
- Drags long-lived projects and products down
- Friction

Cunningham, OOPSLA 1992 50

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

51

Technical Debt Definition (2013)

- 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 (including increased cost over time).

McConnell 2013

52

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

53

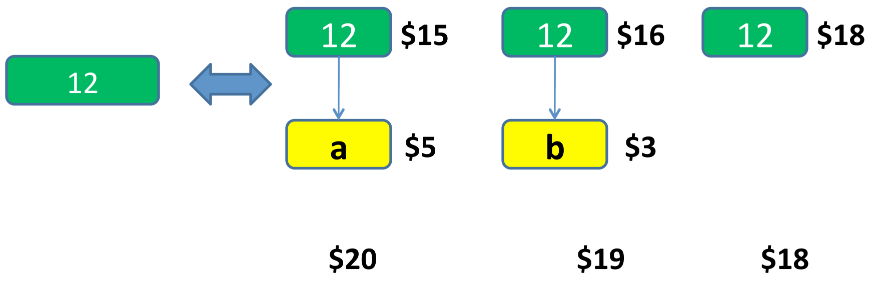
Technical Debt (M. Fowler)

	Reckless	Prudent
Deliberate	*"We don't have time for design"*	*"We must ship now and deal with consequences"*
Inadvertent	*"What's Layering?"*	*"Now we know how we should have done it"*

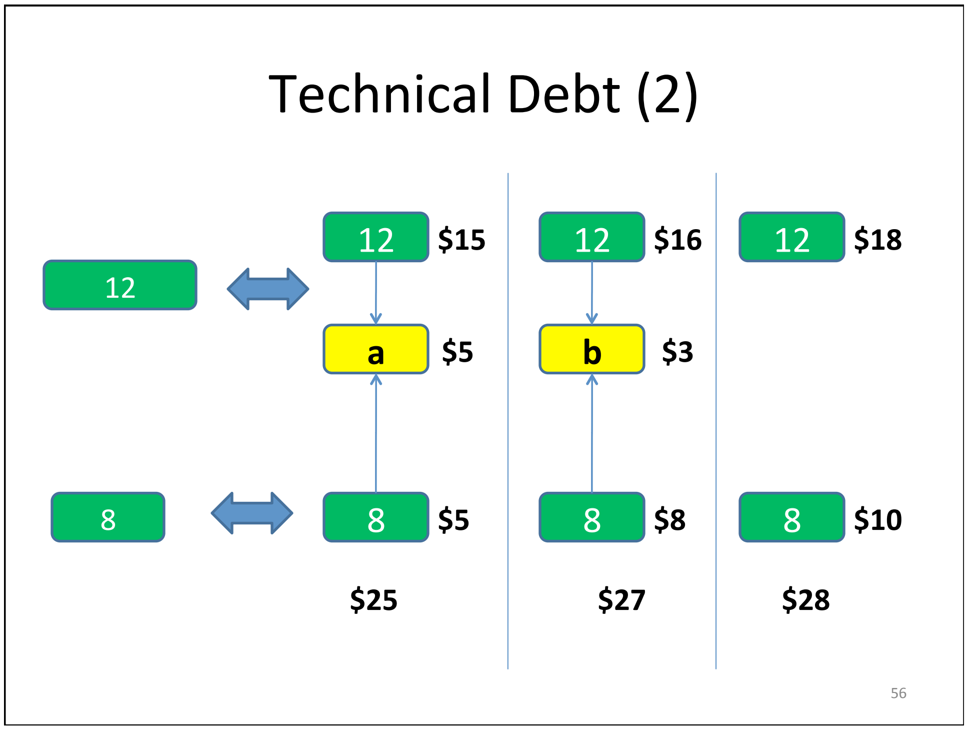
Fowler 2009, 2010

54

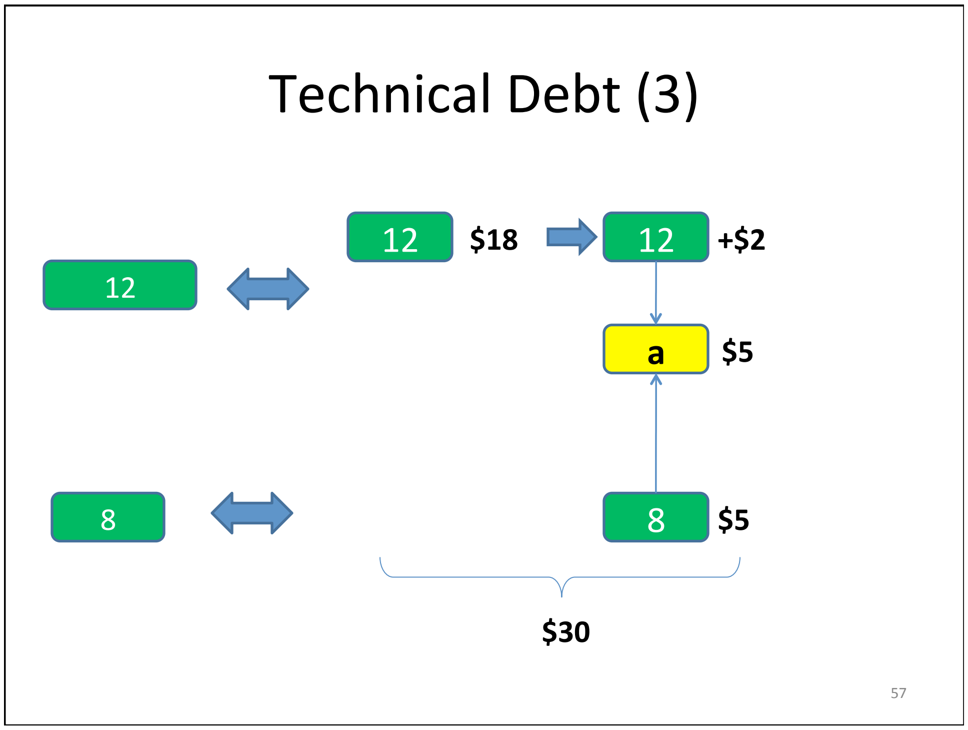
Technical Debt (1)



55



Technical Debt (2)



Technical Debt (3)

Interests

- In presence of technical debt:
  Cost of adding new features is higher
- When repaying (fixing), additional cost for retrofitting already implemented features
- Technical debt not repaid => lead to increased cost, forever
- Cost of fixing increases over time

M. Fowler

58

TD litmus test

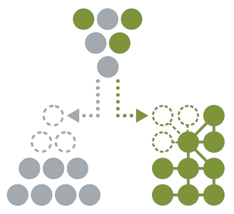
- If you are not incurring any interest, then it probably is not a debt

McConnell 2013

59

Technical Debt Definition 2016

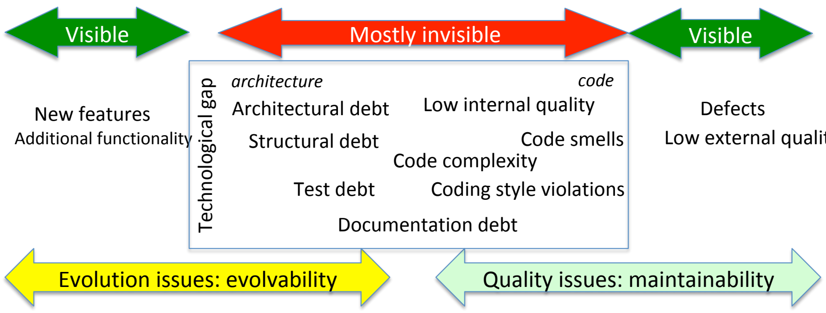
In software-intensive systems, technical debt is the collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible.



"Managing Technical Debt in Software Engineering," Dagstuhl Reports, Vol. 6, Issue 4. http://www.dagstuhl.de/16162.

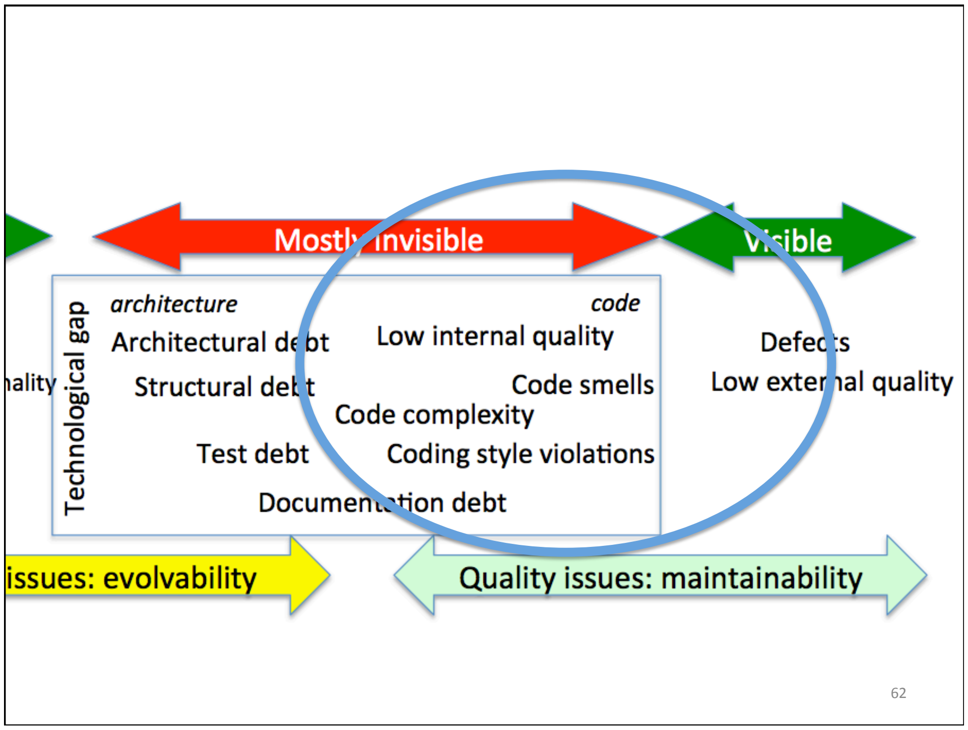
60

Technical debt landscape

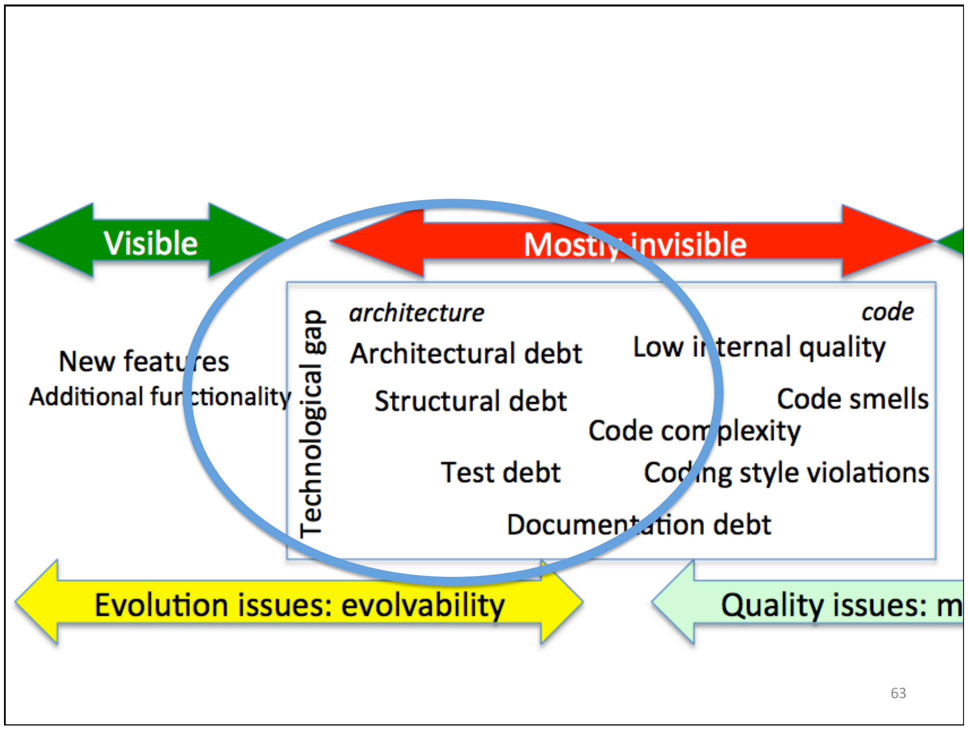


Kruchten et al 2012

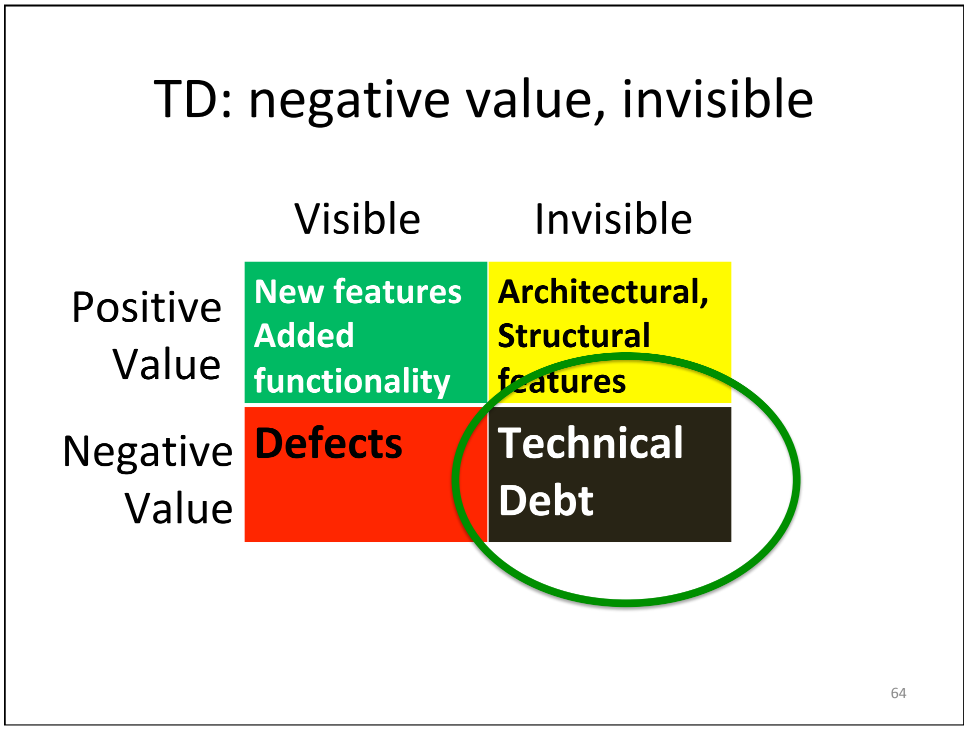
61



Mostly invisible
Visible
architecture
code
Technological gap
Architectural debt
Low internal quality
Structural debt
Code smells
Code complexity
Test debt
Coding style violations
Documentation debt
Defects
Low external quality
ality
issues: evolvability
Quality issues: maintainability
62



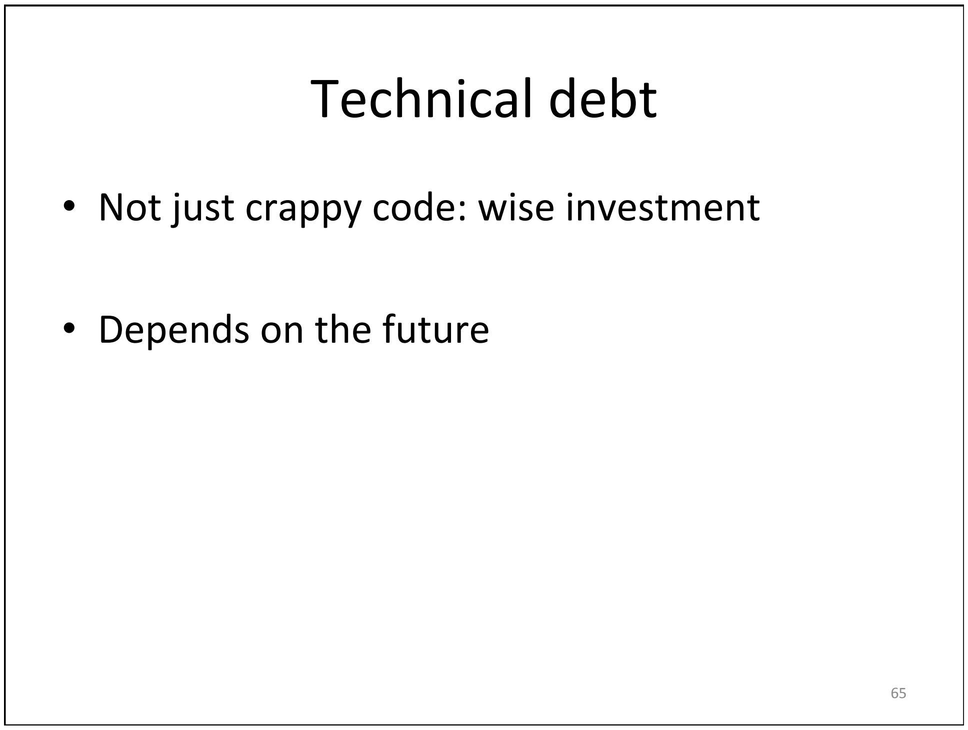
Visible
Mostly invisible
New features
Additional functionality
Technological gap
architecture
code
Architectural debt
Low internal quality
Structural debt
Code smells
Code complexity
Test debt
Coding style violations
Documentation debt
Evolution issues: evolvability
Quality issues: m
63



TD: negative value, invisible

	Visible	Invisible
Positive Value	New features Added functionality	Architectural, Structural features
Negative Value	Defects	Technical Debt

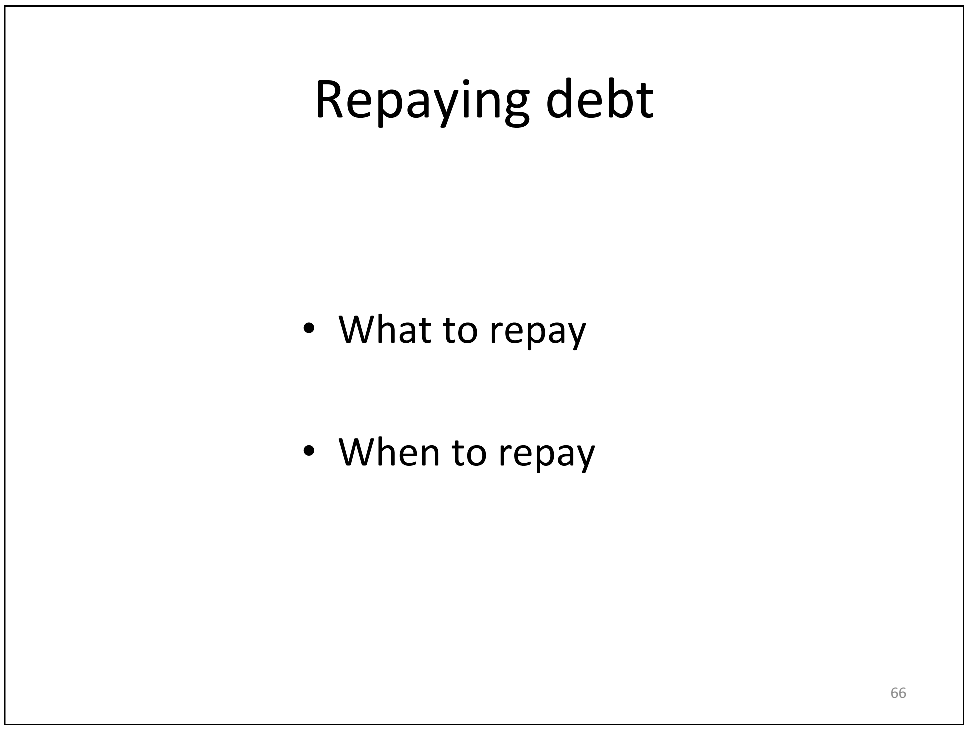
64



Technical debt

- Not just crappy code: wise investment
- Depends on the future

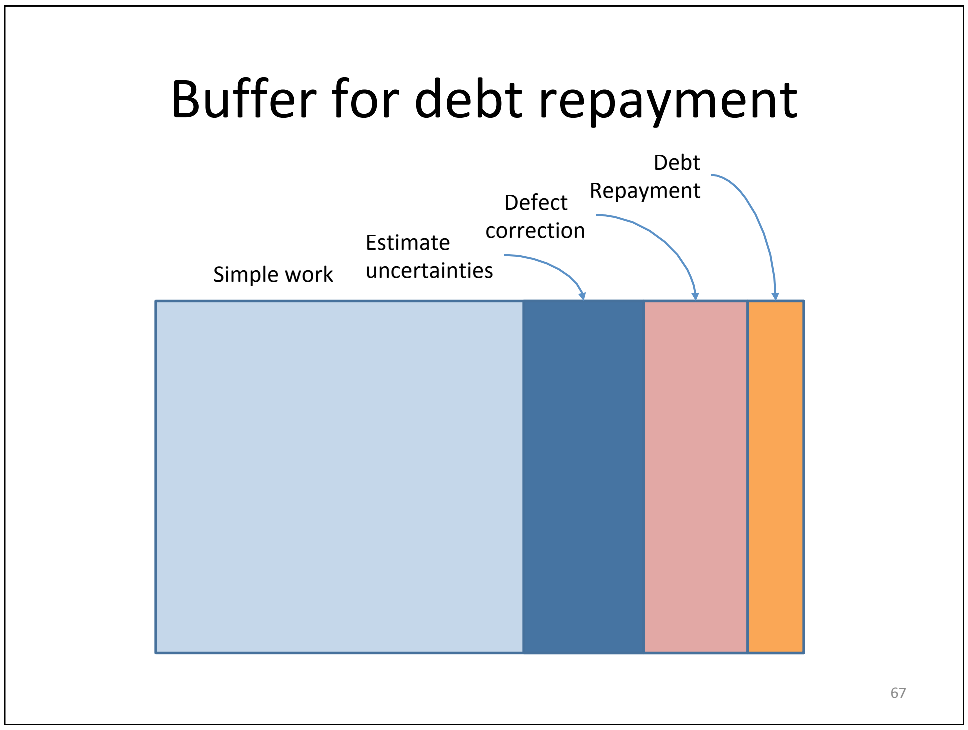
65



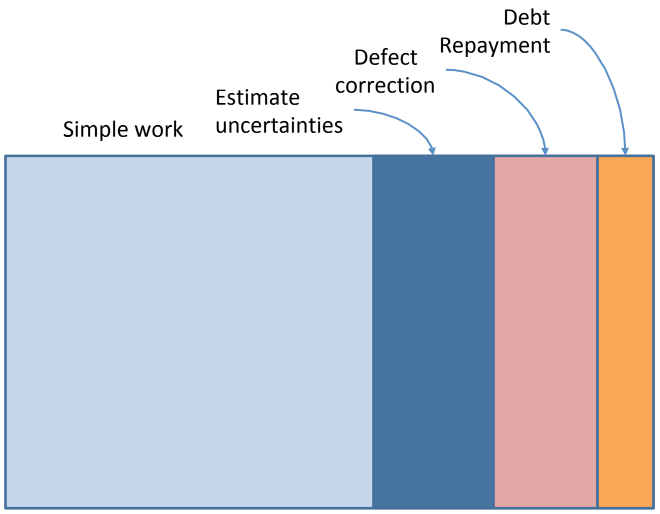
Repaying debt

- What to repay
- When to repay

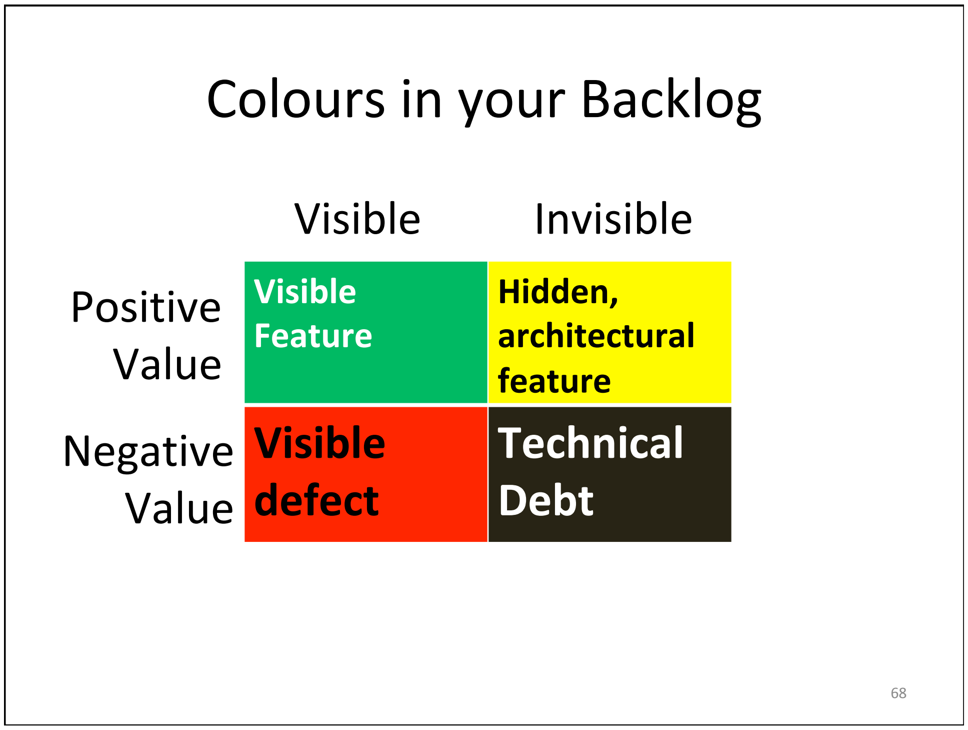
66



Buffer for debt repayment

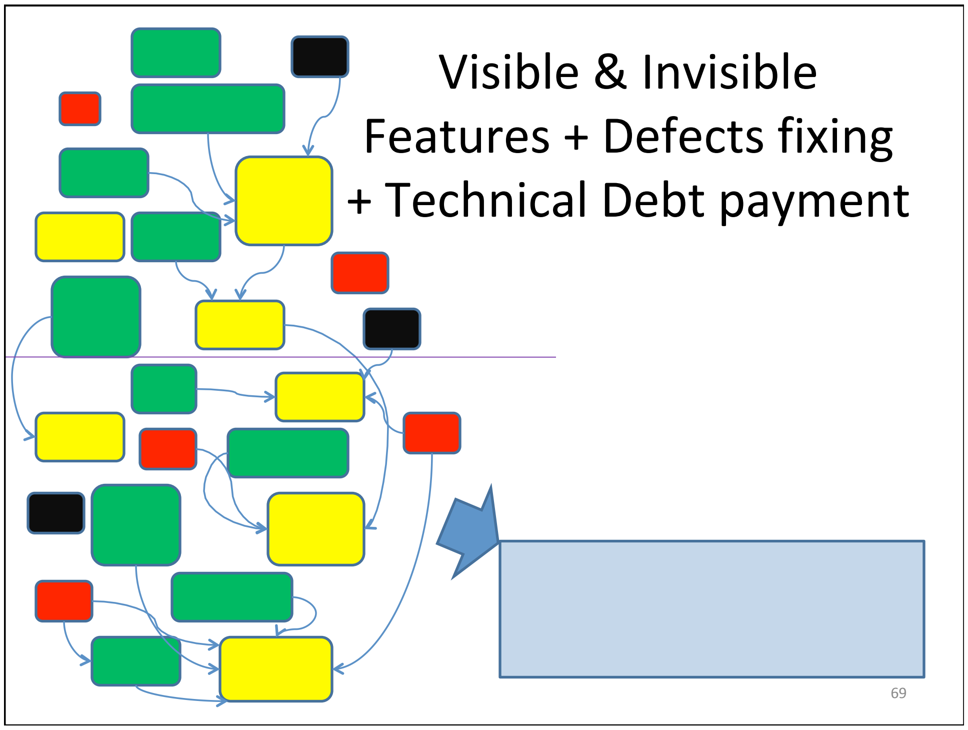


67

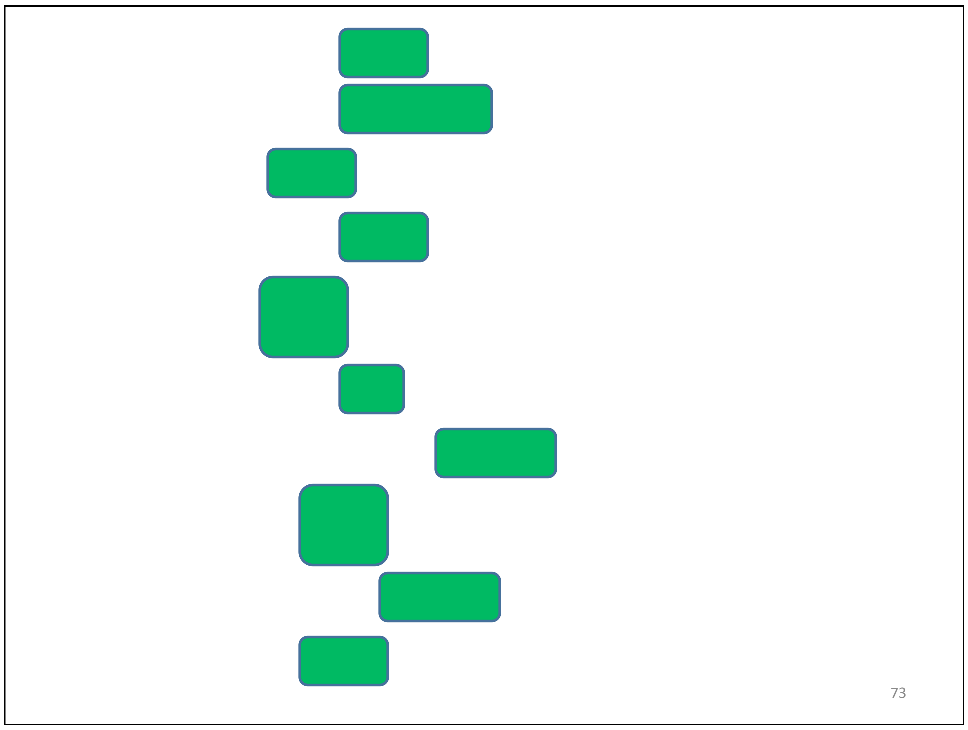


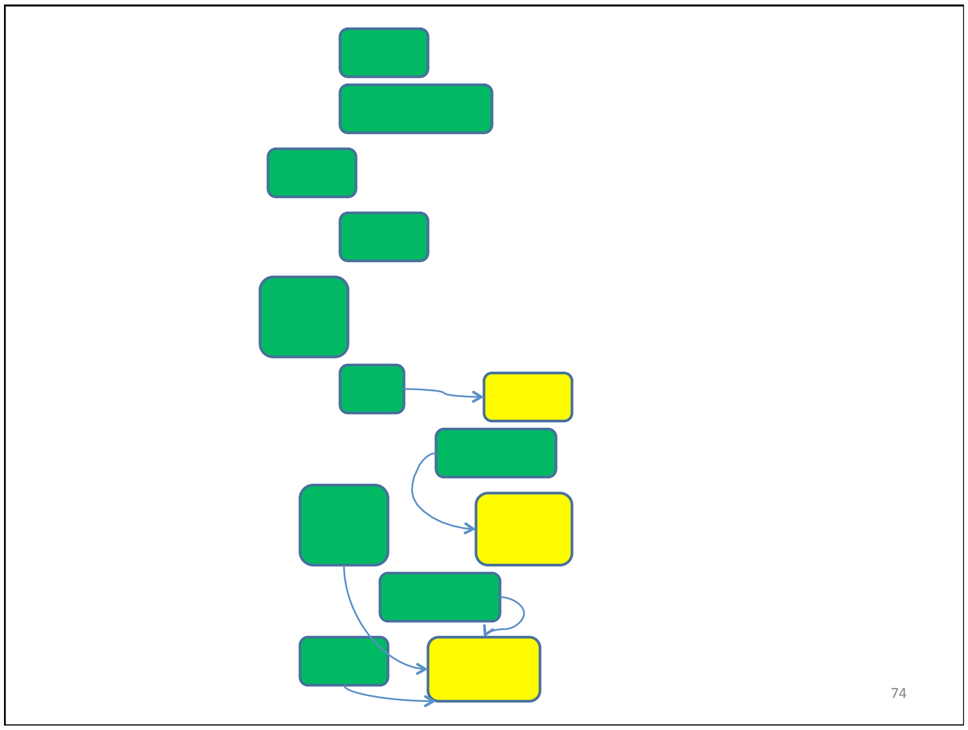
Colours in your Backlog

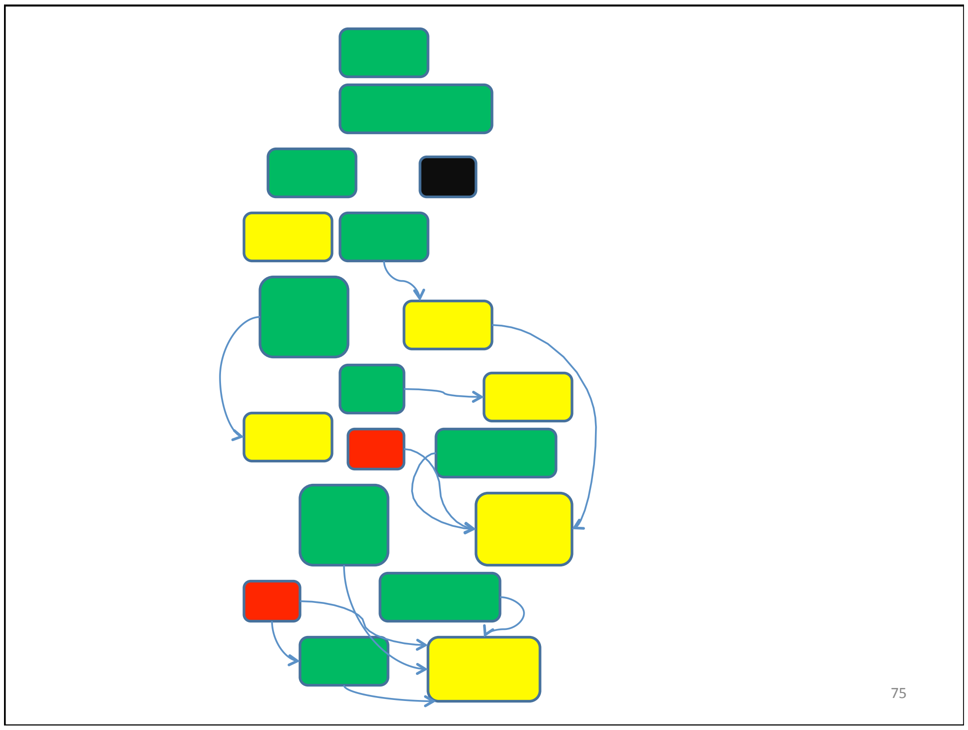
	Visible	Invisible
Positive Value	Visible Feature	Hidden, architectural feature
Negative Value	Visible defect	Technical Debt

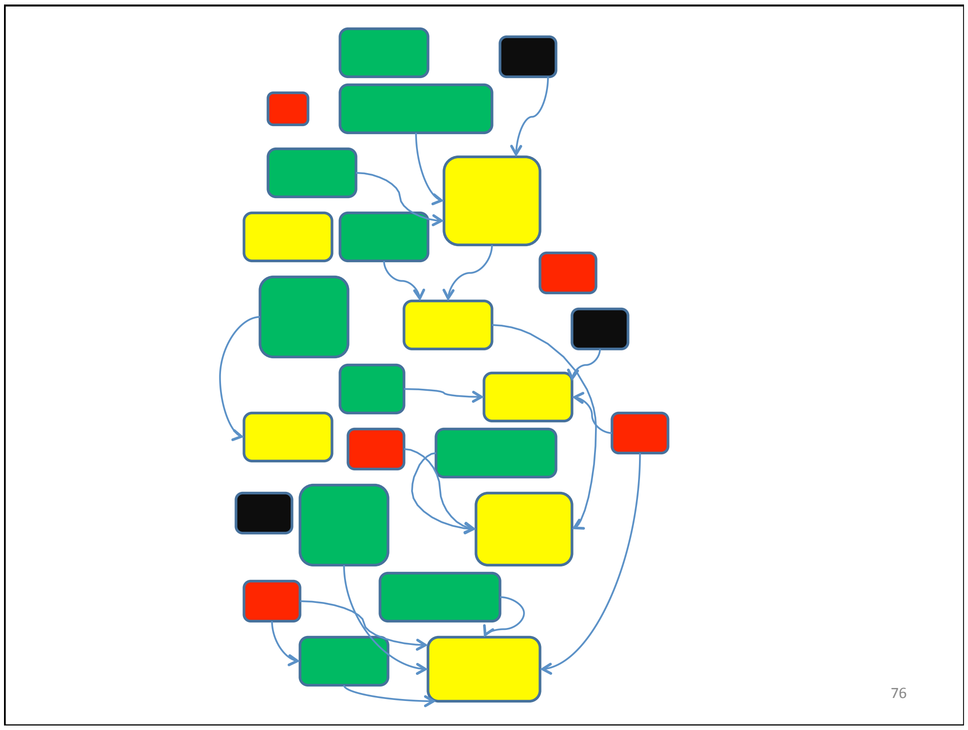


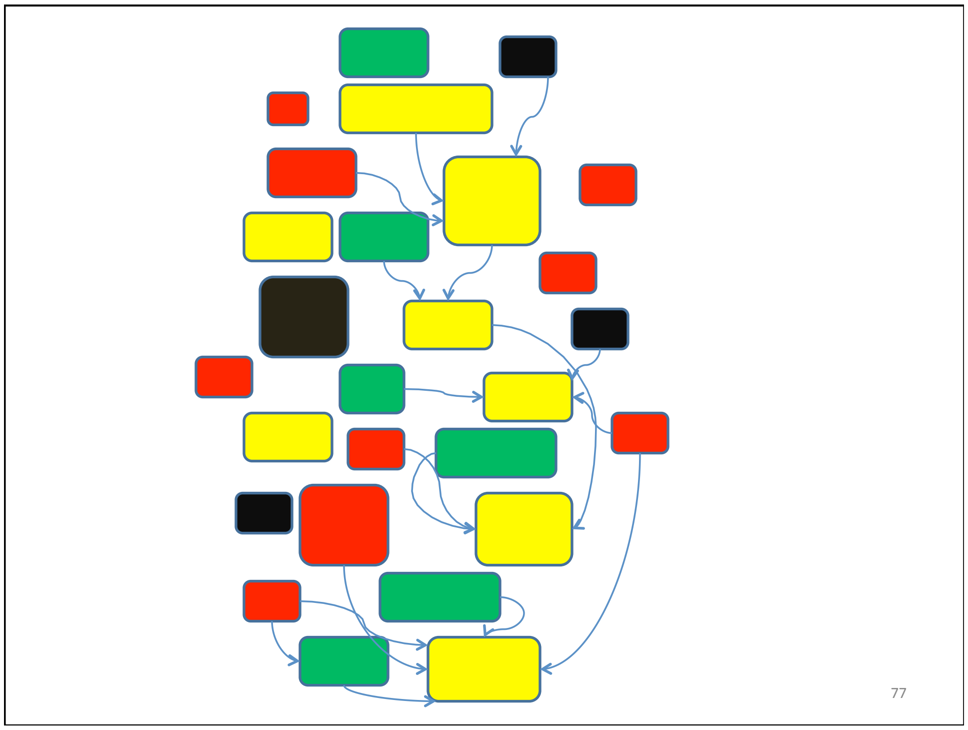
Visible & Invisible Features + Defects fixing + Technical Debt payment

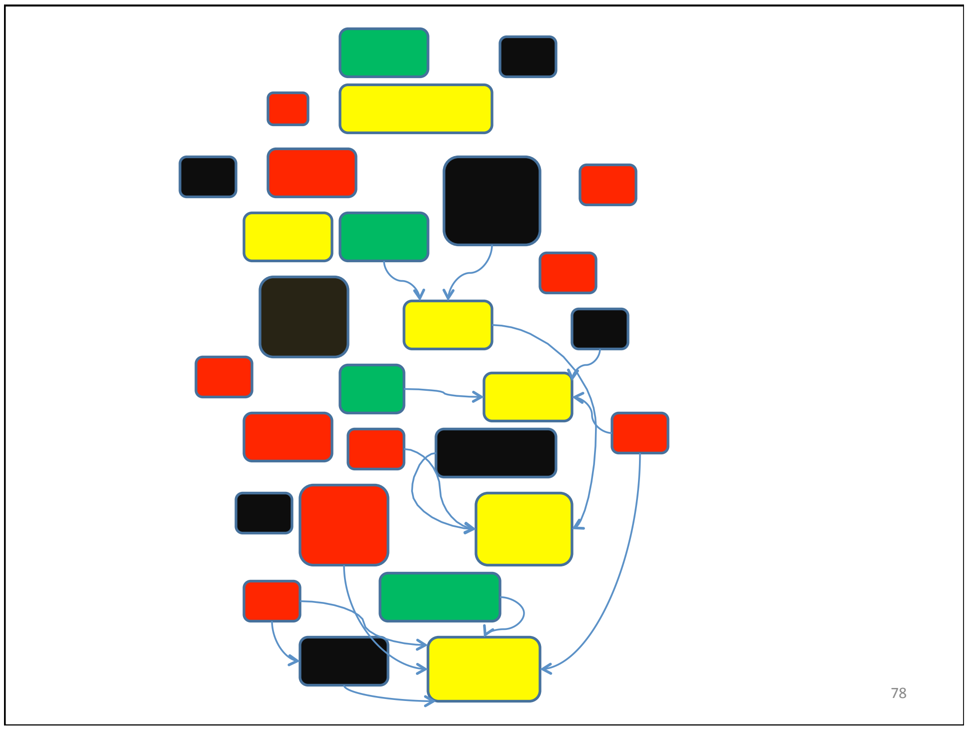




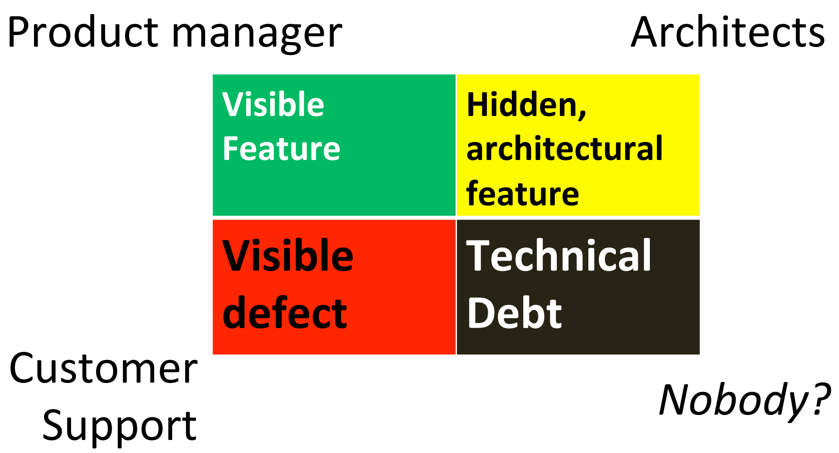

75


76


77


78

Tensions



79

Causes

- Schedule pressure
- Schedule pressure
- Ignorance
- Success
- Environment evolution
  - Technical and business
- Sloppiness

80

Technical debt: Myths

- Ward Cunningham invented technical debt in 1992
- Technical debt is just another fancy name for defects
- We can measure technical debt incurred by a system.
- Tools can identify technical debt incurred by a software system.
- Technical debt is very bad and should be avoided at all cost.

81

Not a new concept

- Just a new term, exploiting a simple metaphor
- Technical debt existed before 1992
  - Software evolution
  - Design erosion
  - Diseconomy of scale

82

TD /= defects

- The software **does work**.
- If it does not, call it a defect, and fix it (calling it technical debt is just a cop-out)
- Technical debt does increase the likelihood of introducing defects => risk !

- *Sometimes, but rarely, the boundary is uncertain.*

83

Bring me tools!

- Static analyzers will detect much of type 1 technical debt.
- More significant technical debt items (structure, architectural) cannot be detected by tools.
- People know about them, though….
- They may be mentioned in discussions, but not visible in the code.

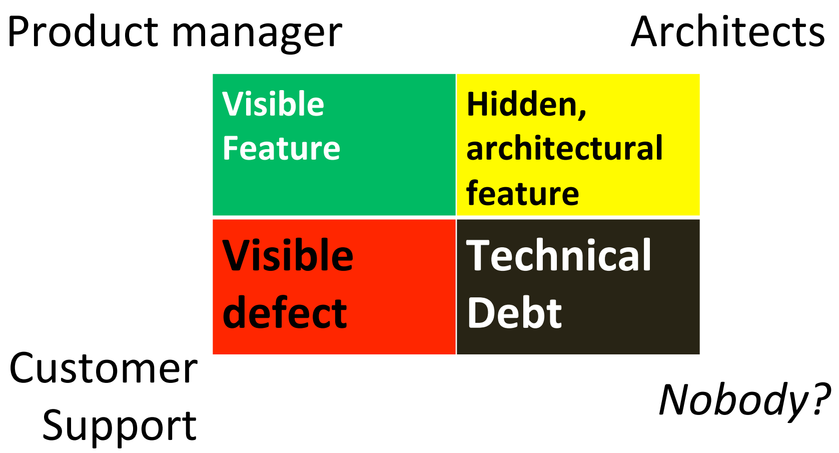
84

Measuring TD

- To measure is to assign a numerical value to an attribute of a thing
- Cost (Technical debt item) = ?
- Naively, the effort to bring the system to a state where the technical debt is not there anymore.

85

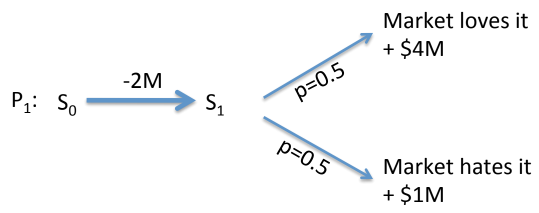
Tensions



87

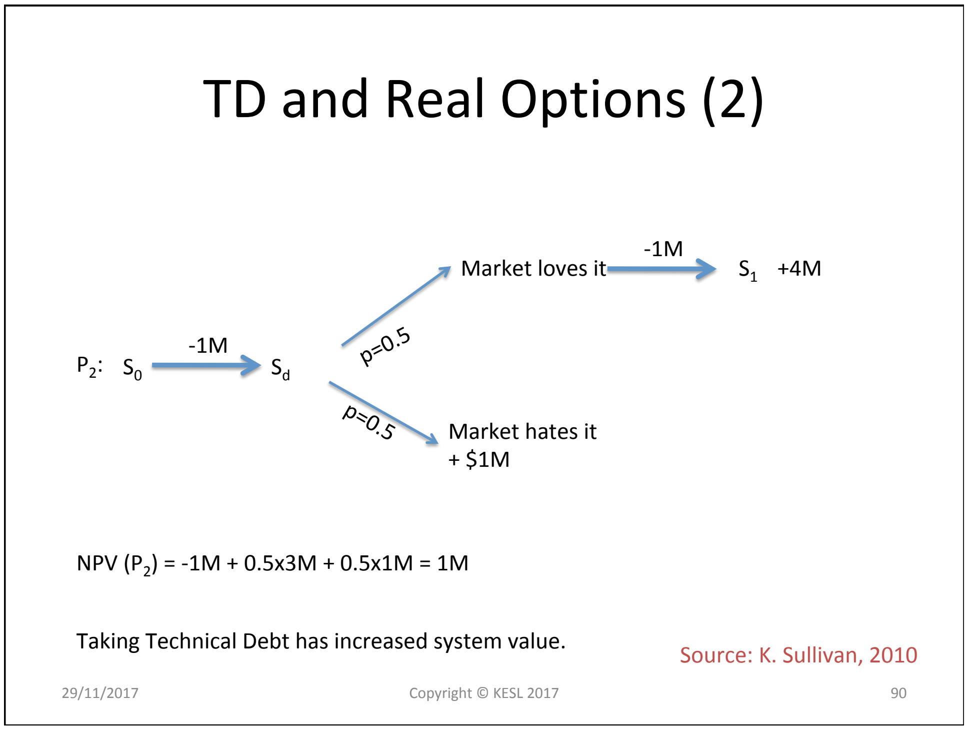
TD as wise investment?

TD and Real Options

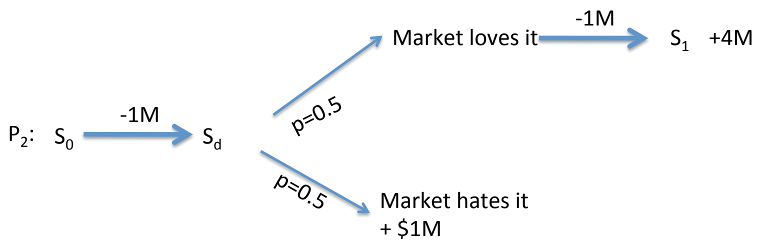


NPV (P_1) = -2M + 0.5x4M + 0.5x1M = 0.5M

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



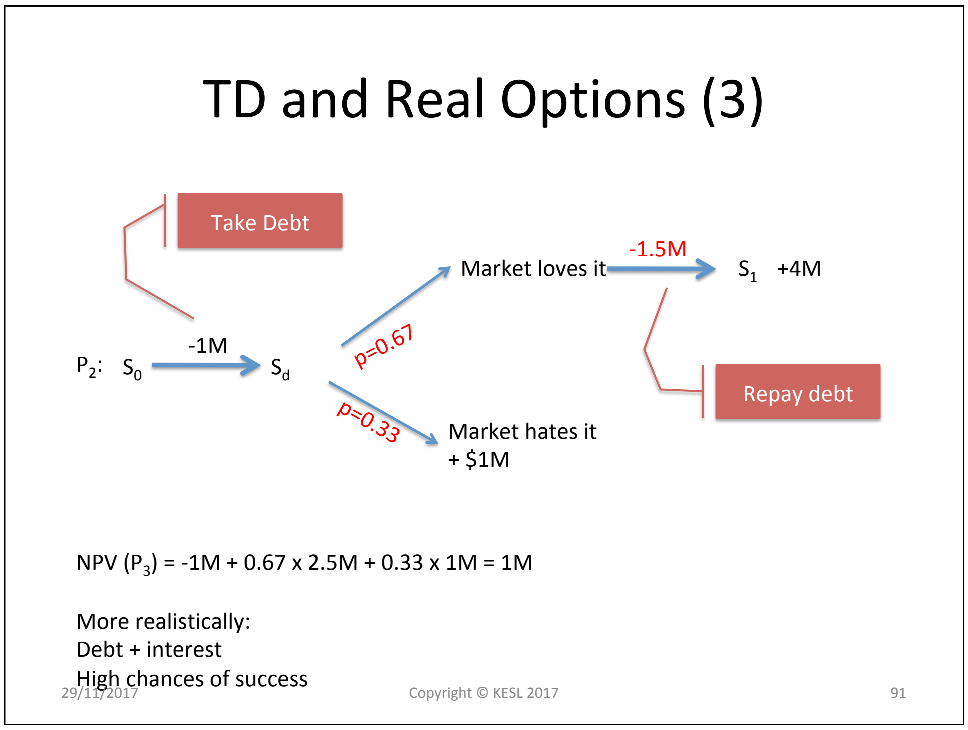
TD and Real Options (2)



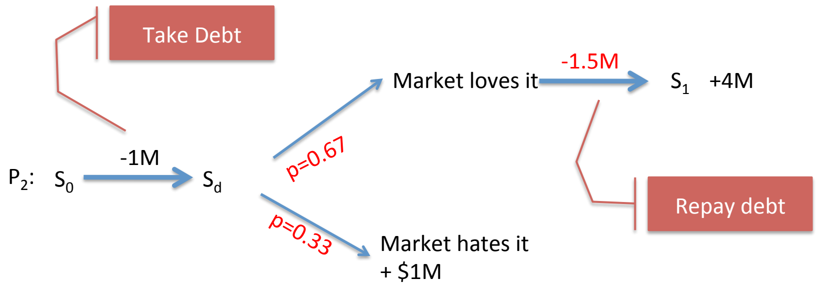
NPV (P_2) = -1M + 0.5x3M + 0.5x1M = 1M

Taking Technical Debt has increased system value.

Source: K. Sullivan, 2010



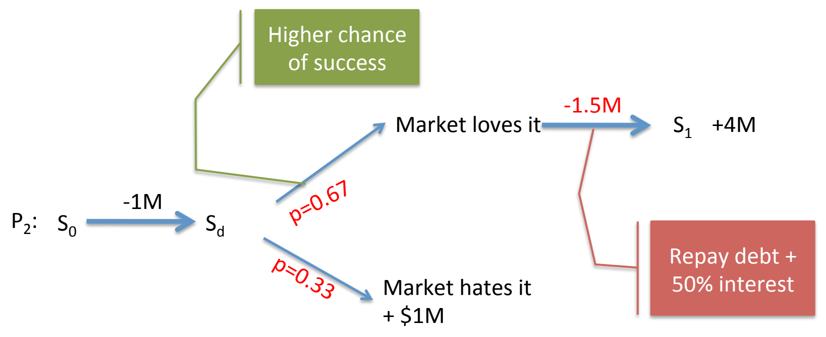
TD and Real Options (3)



NPV (P_3) = -1M + 0.67 x 2.5M + 0.33 x 1M = 1M

More realistically:
Debt + interest
High chances of success

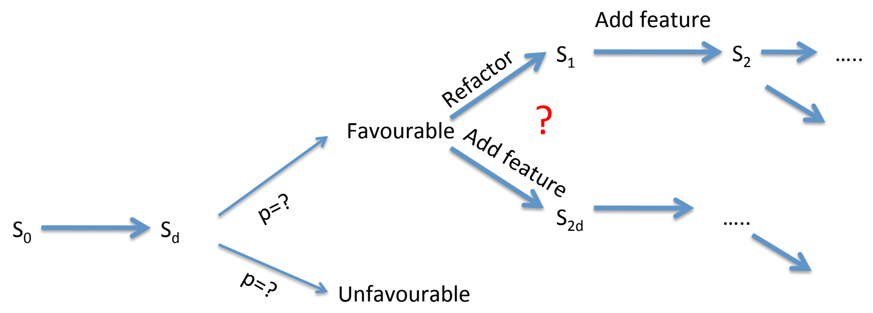
TD and Real Options (3)



NPV (P_3) = -1M + 0.67 x 2.5M + 0.33 x 1M = 1M

More realistically:
Debt + interest
High chances of success

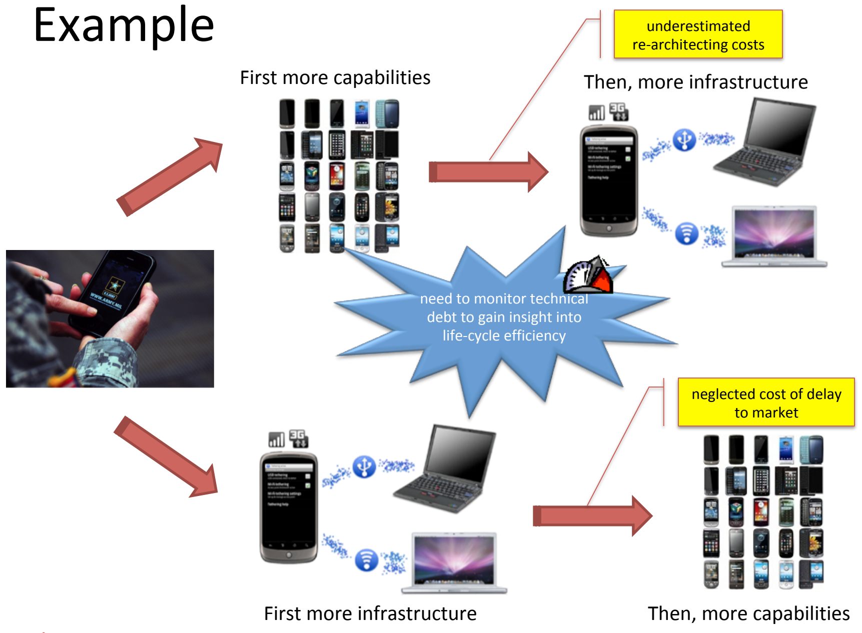
TD and Real Options (4)



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

Source: K. Sullivan, 2010

Example


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

Ozkaya, SEI,2010

94

Practical steps

From tactical (and simple) to more strategic (and sophisticated)

95

- Tactical
  - Short-term actions − limited scope
  - Actual means: use tools, add process steps, make an 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 from 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 analyzer 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.

98

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.

99

Practical Steps (4) Architectural debt

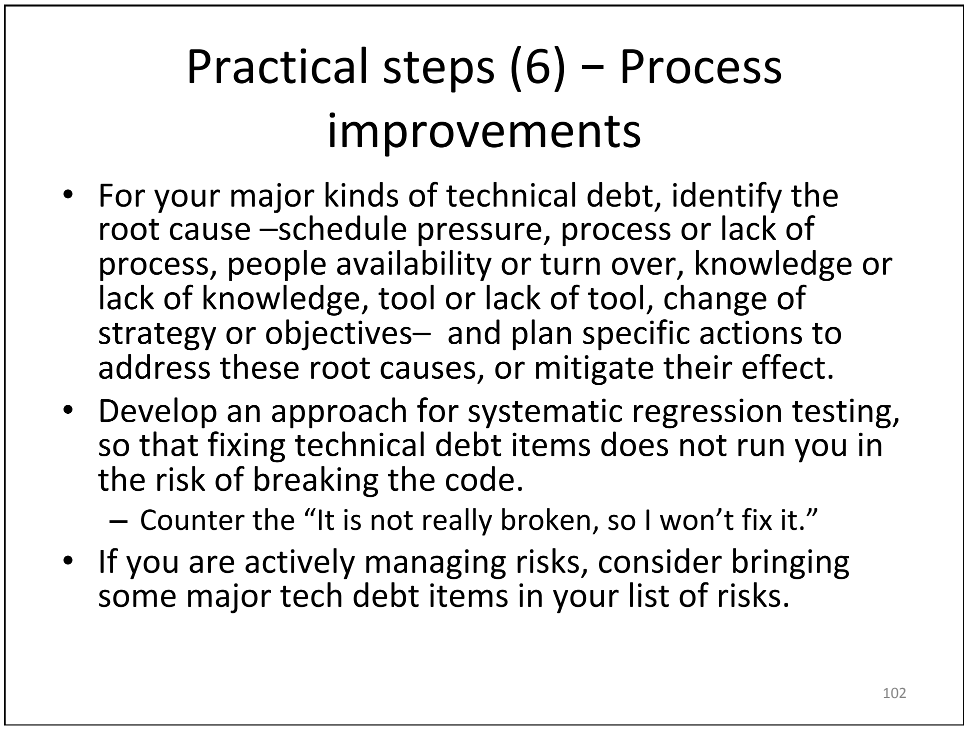
- Refine in your issue tracker the TechDebt category into 2 subcategories: simple, localized, *code-level debt*, and wide ranging, structural or *architectural debt*.
- Acquire and deploy a tool that will give you hints about structural issues in your code: dependency analysis

100

Practical Steps (5) Architectural debt

- 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.

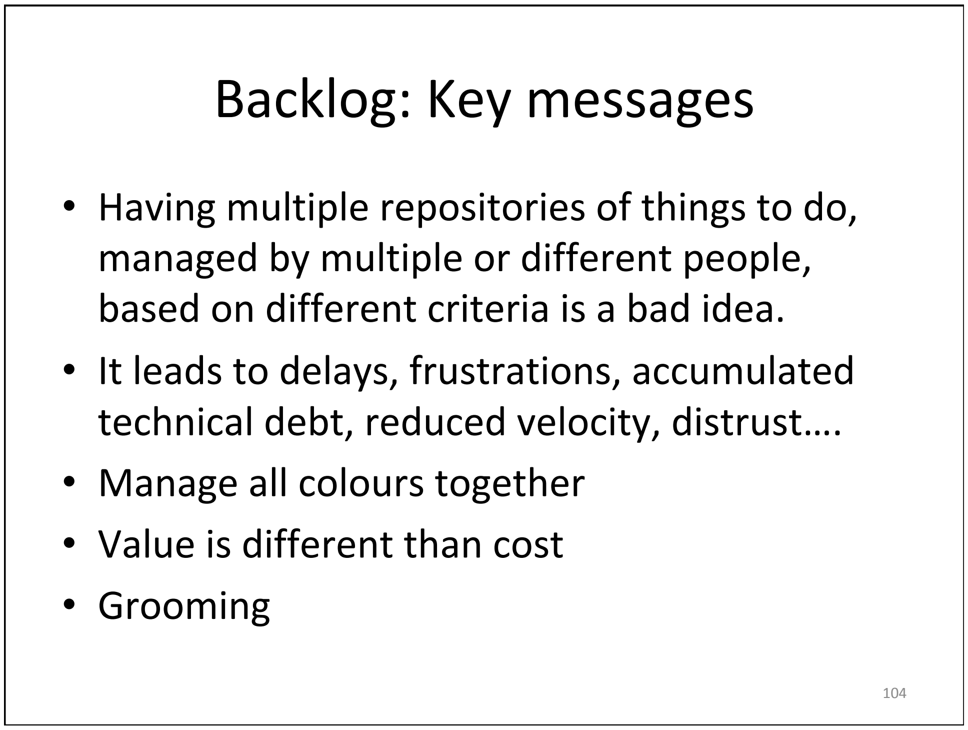
101



Practical steps (6) − 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.

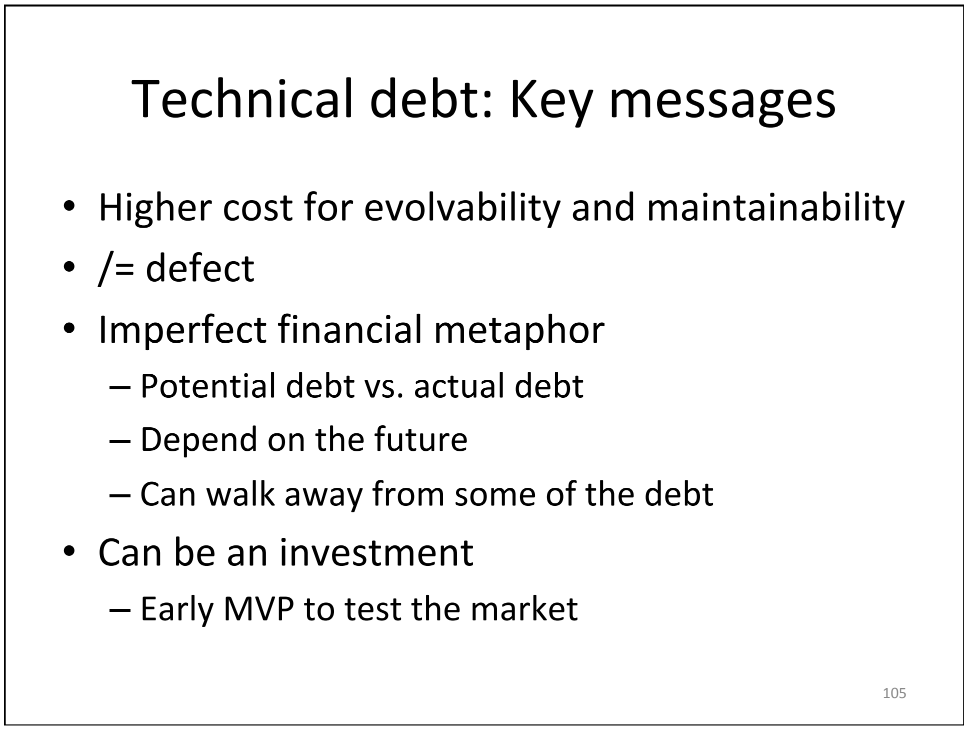
102



Backlog: Key messages

- Having multiple repositories of things to do, managed by multiple or different people, based on different criteria is a bad idea.
- It leads to delays, frustrations, accumulated technical debt, reduced velocity, distrust….
- Manage all colours together
- Value is different than cost
- Grooming

104



Technical debt: Key messages

- Higher cost for evolvability and maintainability
- /= defect
- Imperfect financial metaphor
  - Potential debt vs. actual debt
  - Depend on the future
  - Can walk away from some of the debt
- Can be an investment
  - Early MVP to test the market