AGILE!

The good,
The hype &
The ugly

Bertrand Meyer

Topics

Part 1: Overview

Part 2: The "enemy": Big Upfront Everything

Part 3: What is "agile"?

- 3.1 Agile principles
- 3.2 Agile roles
- 3.3Agile tools and artifacts
- 3.4 Agile practices
- 3.5 Agile methods

Part 4: Critical analysis

- 4.1 Conceptual
- 4.2 Empirical

Part 5: Conclusion

Complementary material: Bibliography

Principles

"Separate news from editorial":

- Parts 1 to 3 are focused on the description
- Parts 4 and 5 are the analysis and critique

But, throughout, the symbol

indicates skepticism or obvious objections that need to be addressed

Part 1: Overview

Part 2: The "enemy': Big Upfront Everything

Part 3: What is "agile"?

- 3.1 Agile principles
- 3.2 Agile roles
- 3.3Agile tools & artifacts
- 3.4 Agile practices
- 3.5 Agile methods

Part 4: Critical analysis

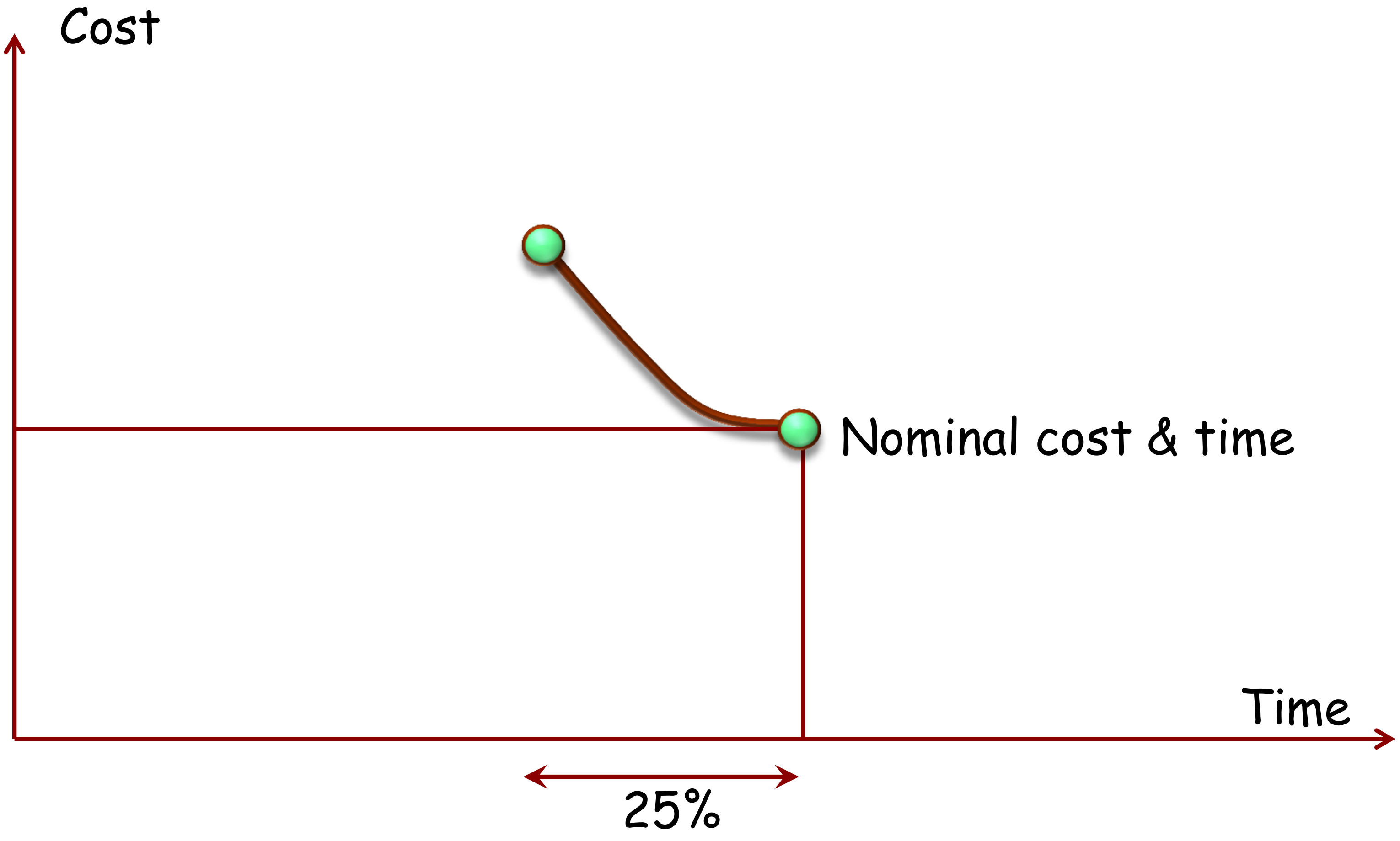
- 4.1 Conceptual
- 4.2 Empirical

Part 5: Conclusion

Complementary material: Bibliography

Reminder: software engineering has laws

Example: Boehm, McConnell, Putnam, Capers Jones...



Assertions

Revolutionary

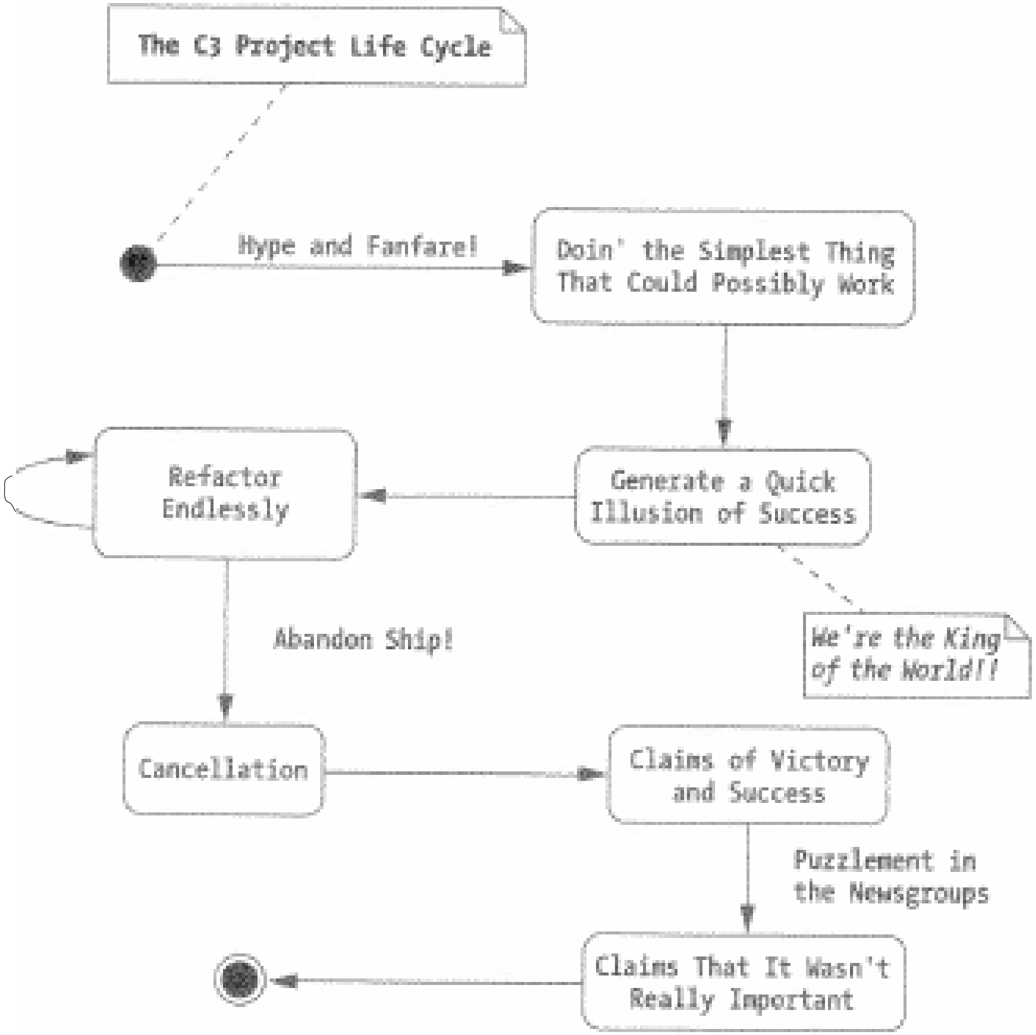
Better

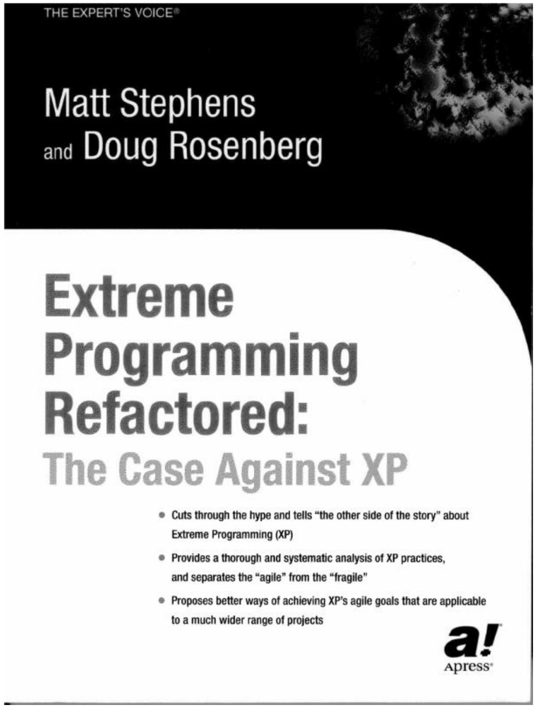
Everyone else is doing it wrong

All or nothing

Not everyone is ecstatic…

Source: Stephens 03


The C3 Project Life Cycle
Hype and Fanfare!
Doin' the Simplest Thing That Could Possibly Work
Generate a Quick Illusion of Success
We're the King of the World!!
Refactor Endlessly
Abandon Ship!
Cancellation
Claims of Victory and Success
Puzzlement in the Newsgroups
Claims That It Wasn't Really Important


THE EXPERT'S VOICE®

Matt Stephens and Doug Rosenberg

Extreme Programming Refactored: The Case Against XP

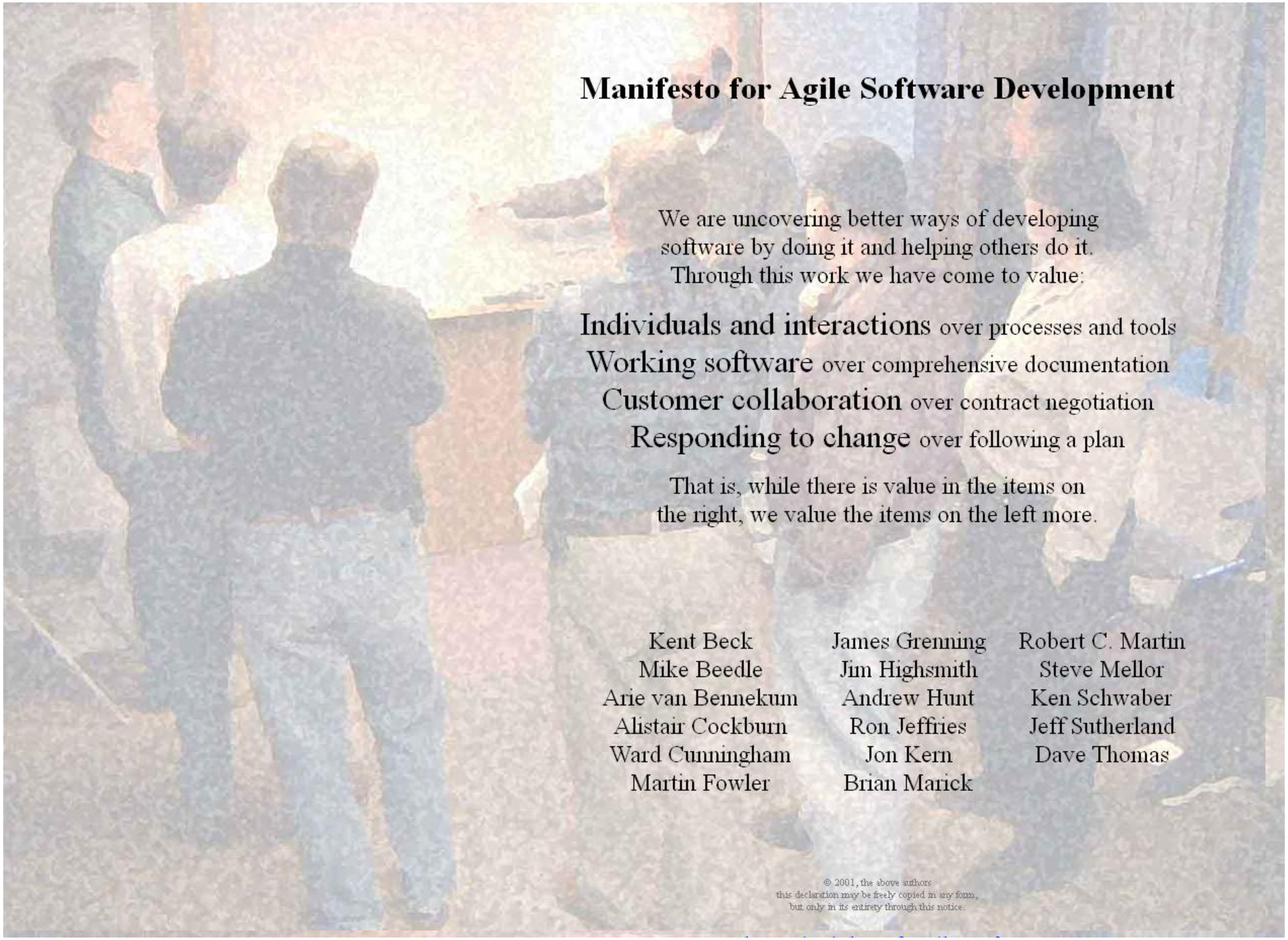
- Cuts through the hype and tells "the other side of the story" about Extreme Programming (XP)
- Provides a thorough and systematic analysis of XP practices, and separates the "agile" from the "fragile"
- Proposes better ways of achieving XP's agile goals that are applicable to a much wider range of projects

Apress®

- 1 -

Overview

Agile manifesto



Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

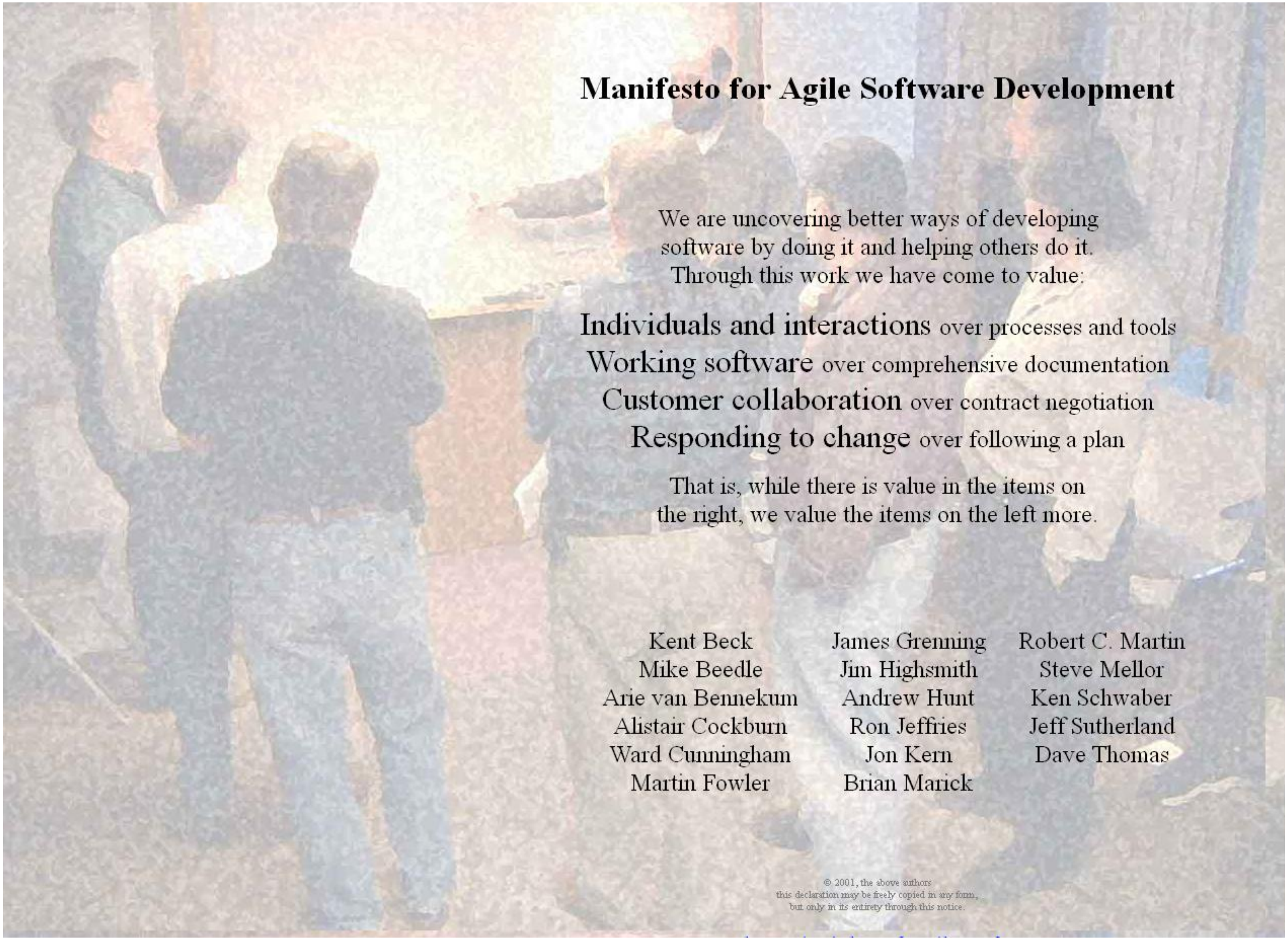
That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler

James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick

Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

Agile manifesto



Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler

James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick

Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

Twelve principles

Source: Agile manifesto

We follow these principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Twelve principles

Source: Agile manifesto

We follow these principles:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

   Redundancy
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

   Redundancy
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

   What about testing?
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

   Practice
7. Working software is the primary measure of progress.

   Assertion
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.

   Assertion
10. Simplicity—the art of maximizing the amount of work not done—is essential.

    Assertion
11. The best architectures, requirements, and designs emerge from self-organizing teams.

    Wrong
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

    Practice

My view: agile assumptions

- **A** New, reduced role for manager
- **B** No "Big Upfront" steps
- **C** Iterative development
- **D** Limited, negotiated scope
- **E** Focus on quality, achieved through testing

My view: agile principles

Organizational

- **1** Place the customer at the center
- **2** Develop minimal software:
  - 2.1 Produce minimal functionality
  - 2.2 Produce only the product requested
  - 2.3 Develop only code and tests
- **3** Accept change
- **4** Let the team self-organize
- **5** Maintain a sustainable pace

Technical

- **6** Produce frequent working iterations
- **7** Treat tests as a key resource:
  - 7.1 Do not start any new development until all tests pass
  - 7.2 Test first
- **8** Express requirements through scenarios

The need for change

Source: Meyer 1997

Are bouts of *esprit de l'escalier* too late in software also? Bad managers suppress them, telling the implementers, in effect, to code and shut up. Good managers try to see whether they can take advantage of belated specification ideas, without attracting the attention of whoever is in charge of enforcing waterfall-style ukases against changing the specification at implementation time.

With O-O development it becomes clear that *esprit de l'escalier* is not just the result of laziness in analysis, but follows from the intrinsic nature of software development. It is not just that we sometimes understand aspects of the problem only at the time of the solution, but more profoundly that the solution affects the problem and suggests better functionalities.

Remember the example of command undoing and redoing: an implementation technique, the "history list" actually suggested a new way of providing end-users of our system with a convenient interface for undoing and redoing commands.

The "lean" view

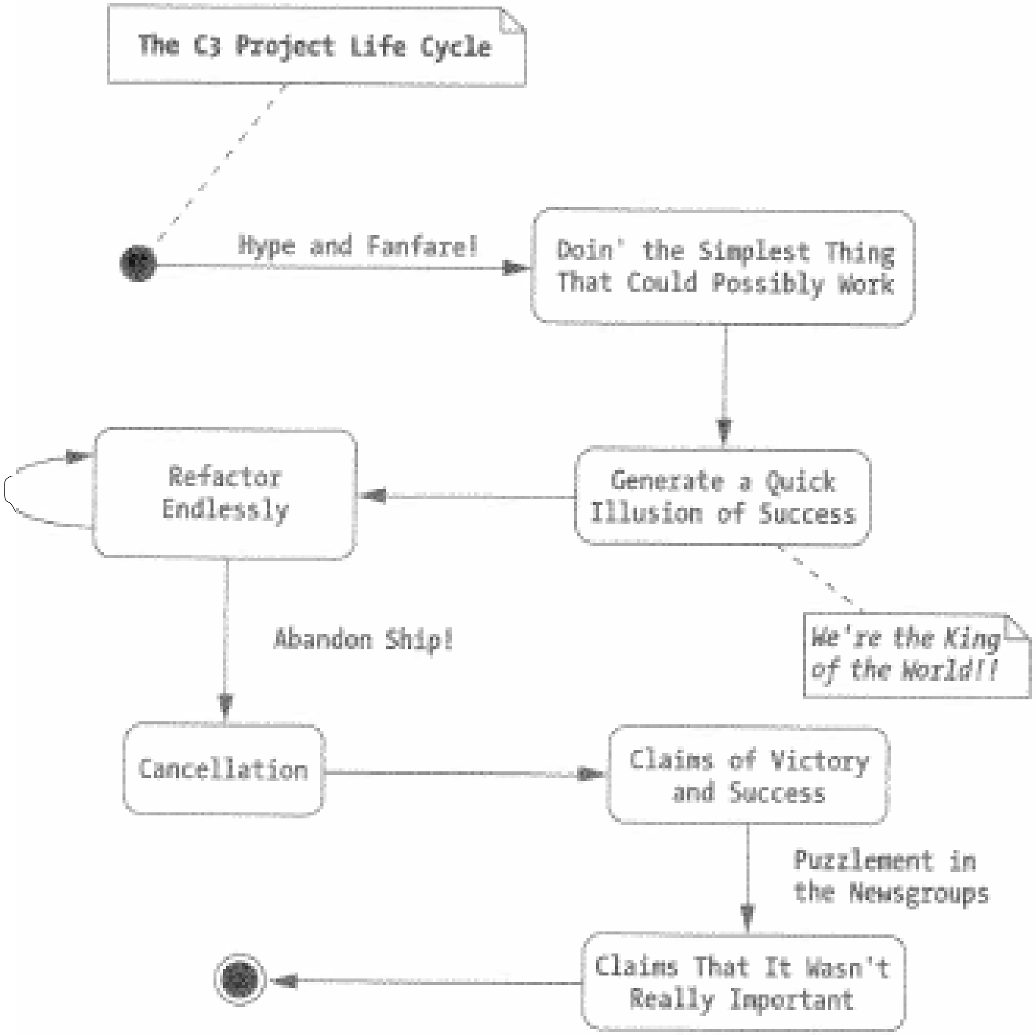
Source: Poppendieck

Seven wastes of software development:

- Extra/Unused features *(Overproduction)*
- Partially developed work not released to production *(Inventory)*
- Intermediate/unused artifacts *(Extra Processing)*
- Seeking Information *(Motion)*
- Escaped defects not caught by tests/reviews *(Defects)*
- Waiting (including Customer Waiting)
- Handoffs *(Transportation)*

Not everyone is ecstatic…

Source: Stephens 03



The C3 Project Life Cycle

Hype and Fanfare!

Doin' the Simplest Thing That Could Possibly Work

Generate a Quick Illusion of Success

We're the King of the World!!

Refactor Endlessly

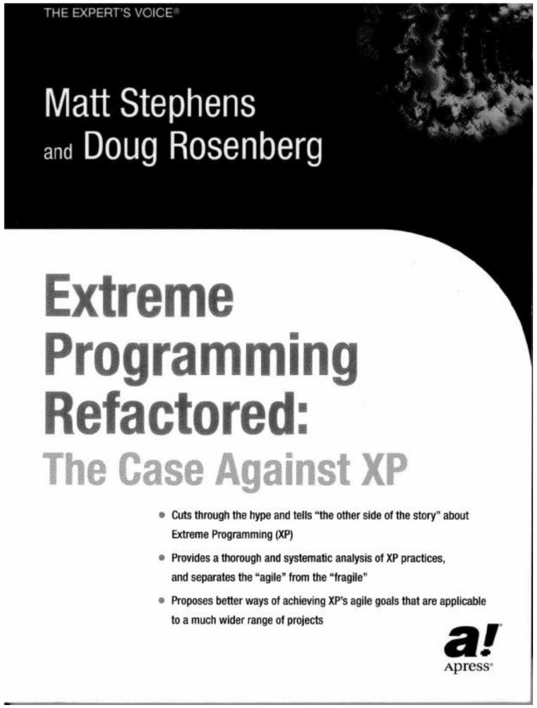
Abandon Ship!

Cancellation

Claims of Victory and Success

Puzzlement in the Newsgroups

Claims That It Wasn't Really Important



THE EXPERT'S VOICE®

Matt Stephens and Doug Rosenberg

Extreme Programming Refactored: The Case Against XP

- Cuts through the hype and tells "the other side of the story" about Extreme Programming (XP)
- Provides a thorough and systematic analysis of XP practices, and separates the "agile" from the "fragile"
- Proposes better ways of achieving XP's agile goals that are applicable to a much wider range of projects

a! Apress®

- 2 -

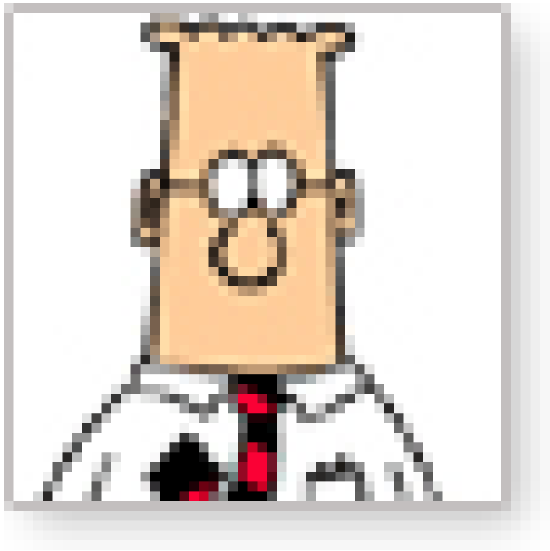
The "enemy": Big Upfront Everything

Big

Upfront

Anything

The revolt of the cubicles



A slogan

Source: Meyer 1997

All you need is code

Code is all you need

The bOOtles

Heavyweight methods

(Sometimes called formal or heavyweight)

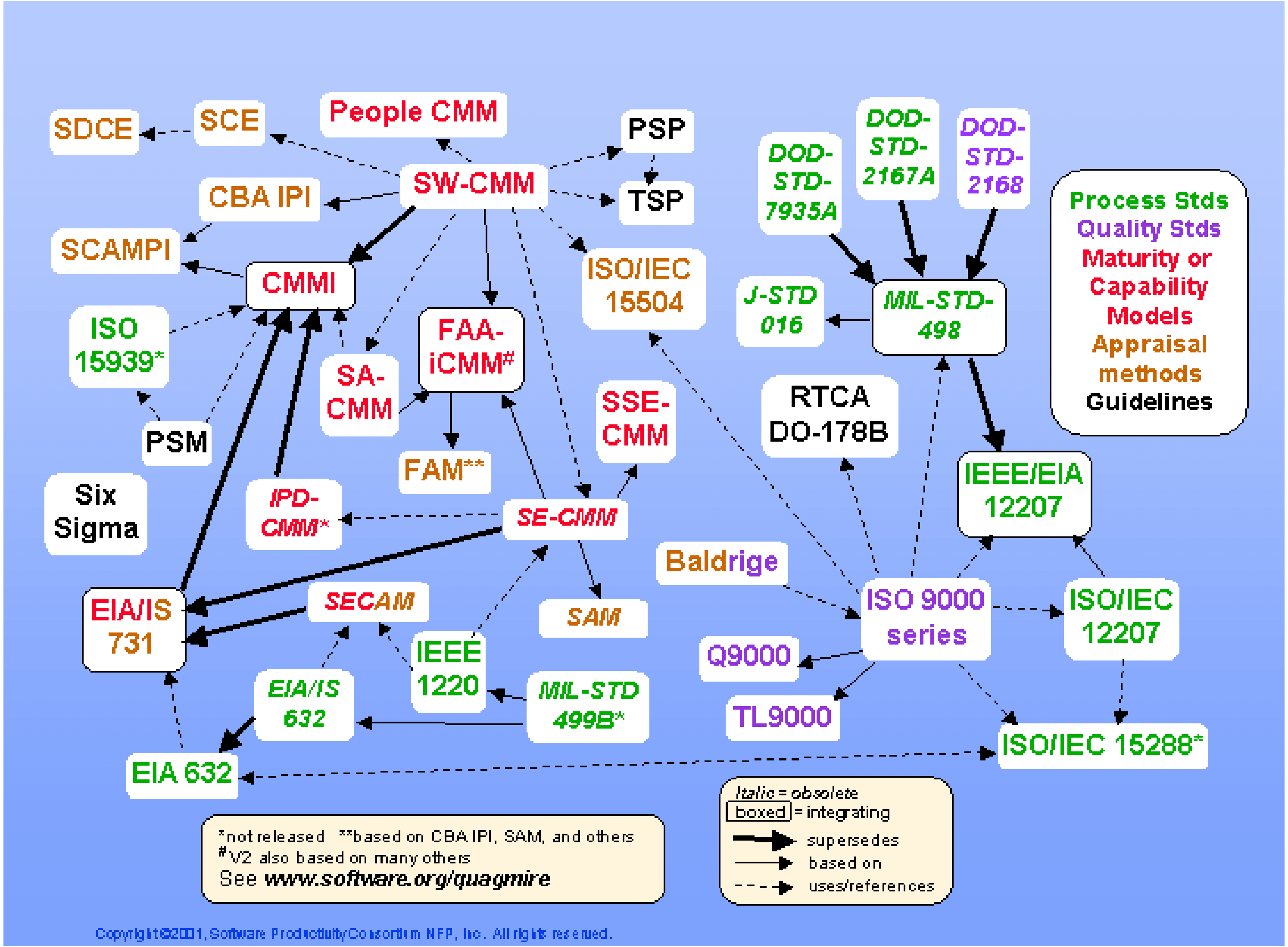
Examples:

- Waterfall model (from 1970 on)
- Military standards
- CMM, then CMMI
- Unified Modeling Language (UML)
- ISO 9000 series of standards
- Rational Unified Process (RUP)
- Personal and Team Software Process (PSP/TSP)
- Cluster model

Overall idea: to enforce a strong engineering discipline on the software development process

- Controllability, manageability
- Traceability
- Reproducibility

The world of standards



*not released **based on CBA IPI, SAM, and others
#V2 also based on many others
See **www.software.org/quagmire**

Italic = *obsolete*
boxed = integrating
supersedes
based on
uses/references

http://www.software.org/quagmire/

Lifecycle models

Origin: Royce, 1970, Waterfall model

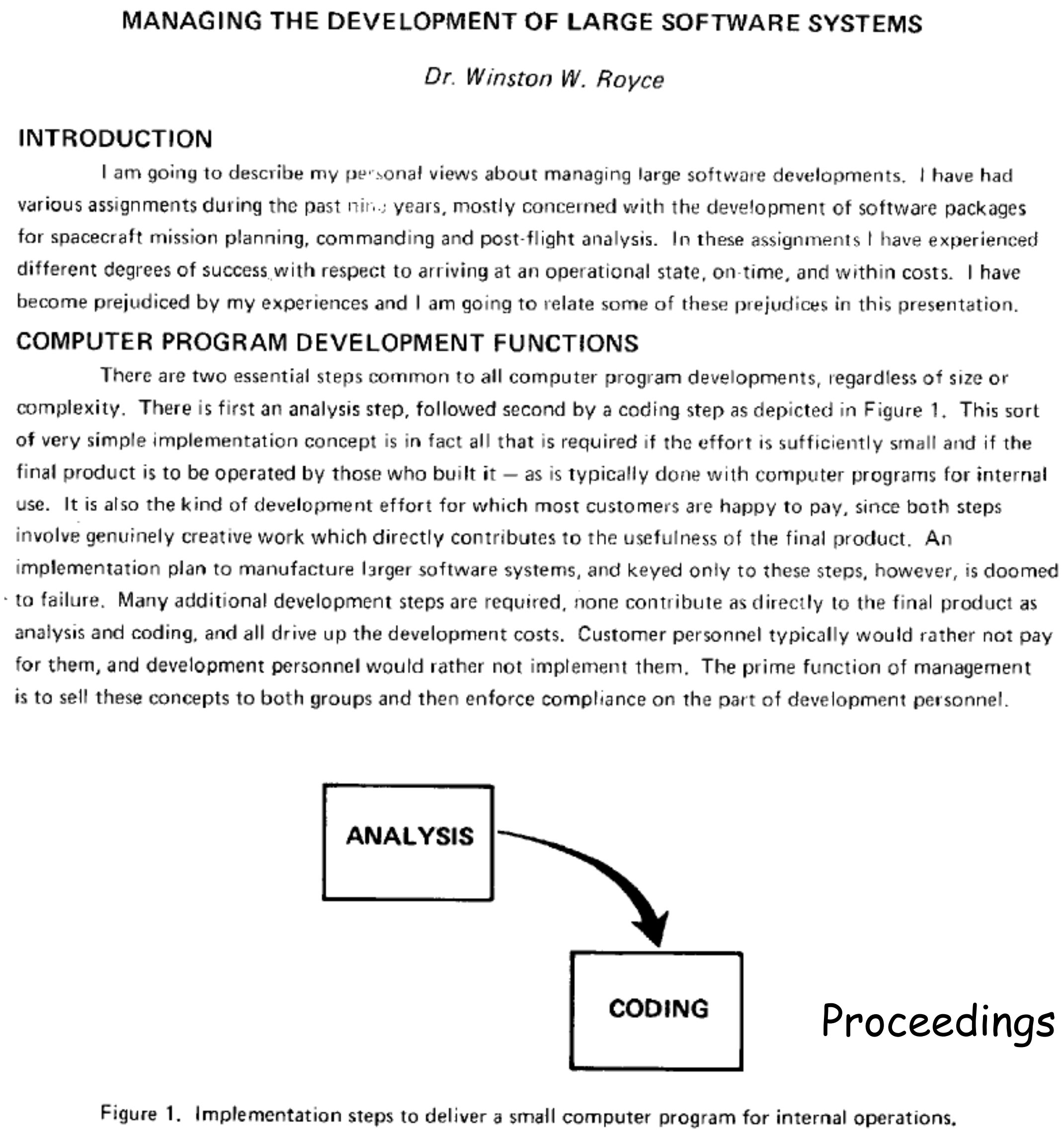
Scope: describe the set of processes involved in the production of software systems, and their sequencing

"Model" in two meanings of the term:

- Idealized description of reality
- Ideal to be followed

The original waterfall article

Source: Royce 1970



MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS

Dr. Winston W. Royce

INTRODUCTION

I am going to describe my personal views about managing large software developments. I have had various assignments during the past nine years, mostly concerned with the development of software packages for spacecraft mission planning, commanding and post-flight analysis. In these assignments I have experienced different degrees of success with respect to arriving at an operational state, on-time, and within costs. I have become prejudiced by my experiences and I am going to relate some of these prejudices in this presentation.

COMPUTER PROGRAM DEVELOPMENT FUNCTIONS

There are two essential steps common to all computer program developments, regardless of size or complexity. There is first an analysis step, followed second by a coding step as depicted in Figure 1. This sort of very simple implementation concept is in fact all that is required if the effort is sufficiently small and if the final product is to be operated by those who built it — as is typically done with computer programs for internal use. It is also the kind of development effort for which most customers are happy to pay, since both steps involve genuinely creative work which directly contributes to the usefulness of the final product. An implementation plan to manufacture larger software systems, and keyed only to these steps, however, is doomed to failure. Many additional development steps are required, none contribute as directly to the final product as analysis and coding, and all drive up the development costs. Customer personnel typically would rather not pay for them, and development personnel would rather not implement them. The prime function of management is to sell these concepts to both groups and then enforce compliance on the part of development personnel.

ANALYSIS

CODING

Figure 1. Implementation steps to deliver a small computer program for internal operations.

Proceedings of IEEE WESCON, pages 1-9, 1970

Waterfall (continued)

Source: Royce 1970

A more grandiose approach to software development is illustrated in Figure 2. The analysis and coding steps are still in the picture, but they are preceded by two levels of requirements analysis, are separated by a program design step, and followed by a testing step. These additions are treated separately from analysis and coding because they are distinctly different in the way they are executed. They must be planned and staffed differently for best utilization of program resources.

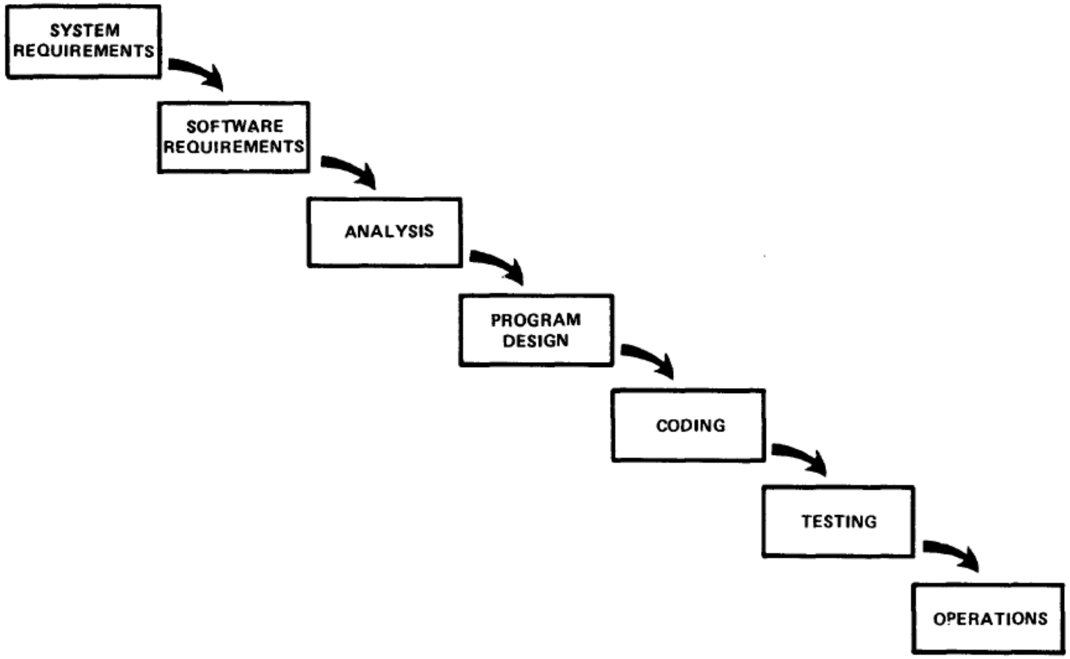


Figure 2. Implementation steps to develop a large computer program for delivery to a customer.

Waterfall (continued)

Source: Royce 1970

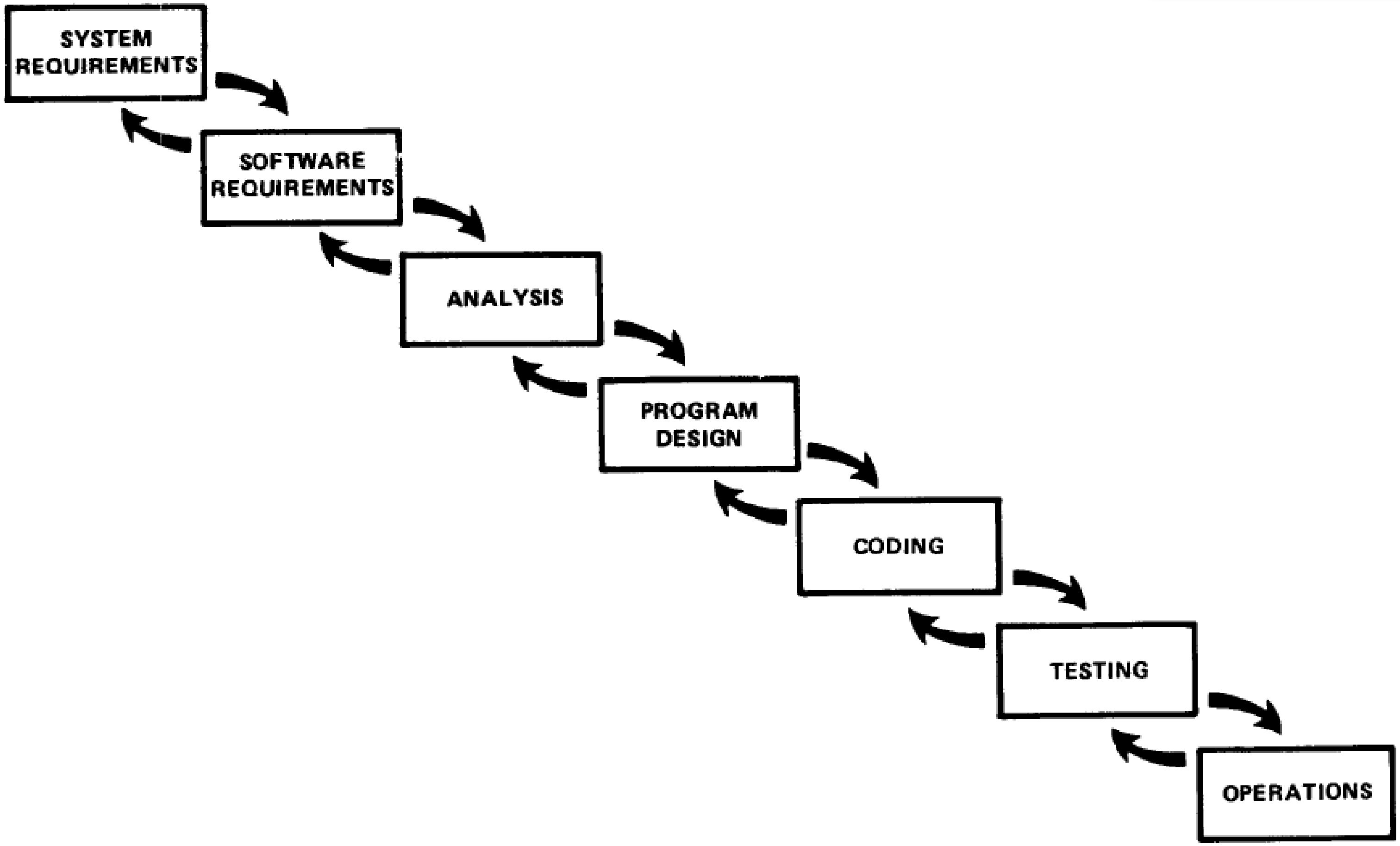


Figure 3. Hopefully, the iterative interaction between the various phases is confined to successive steps.

Waterfall (continued)

Source: Royce 1970

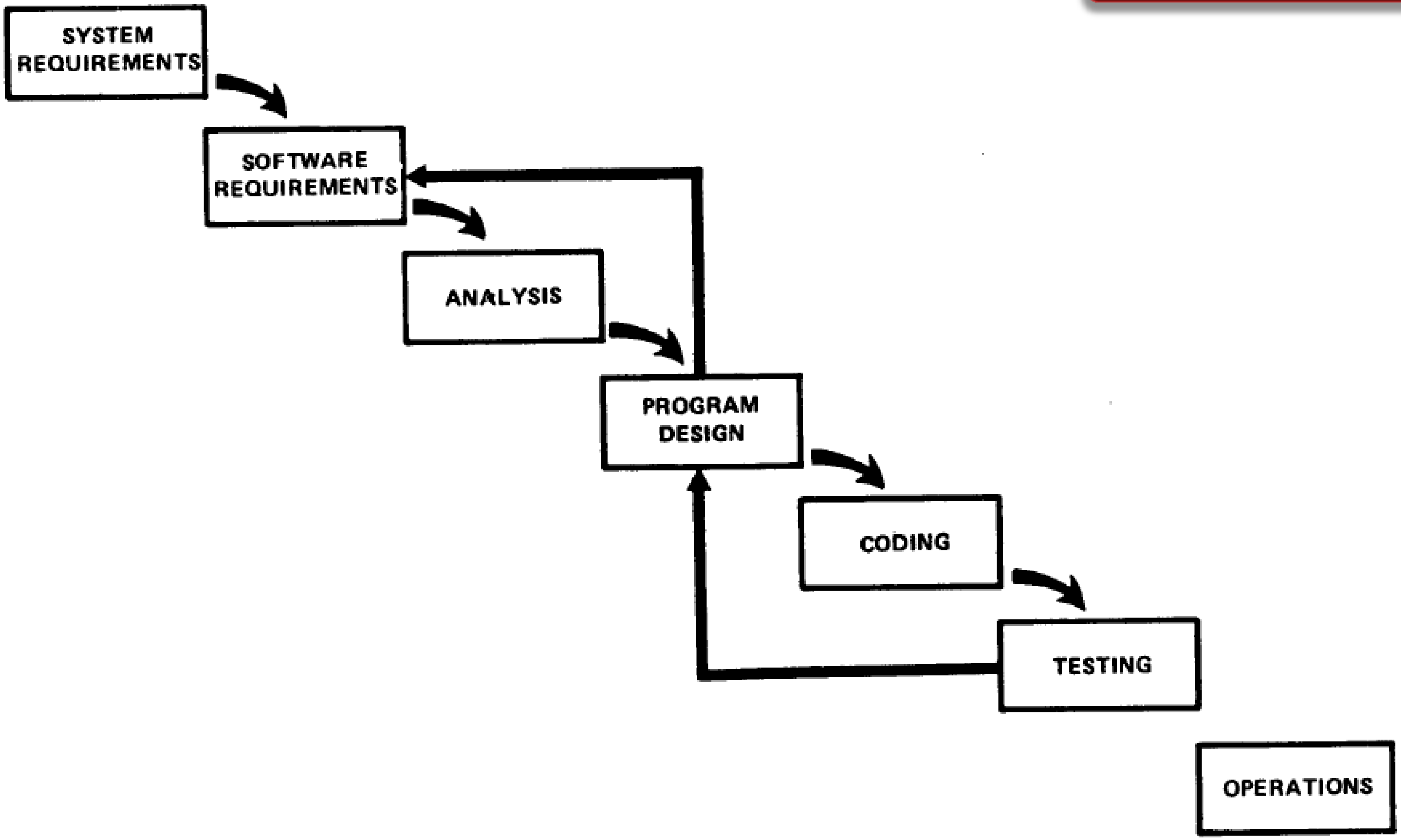
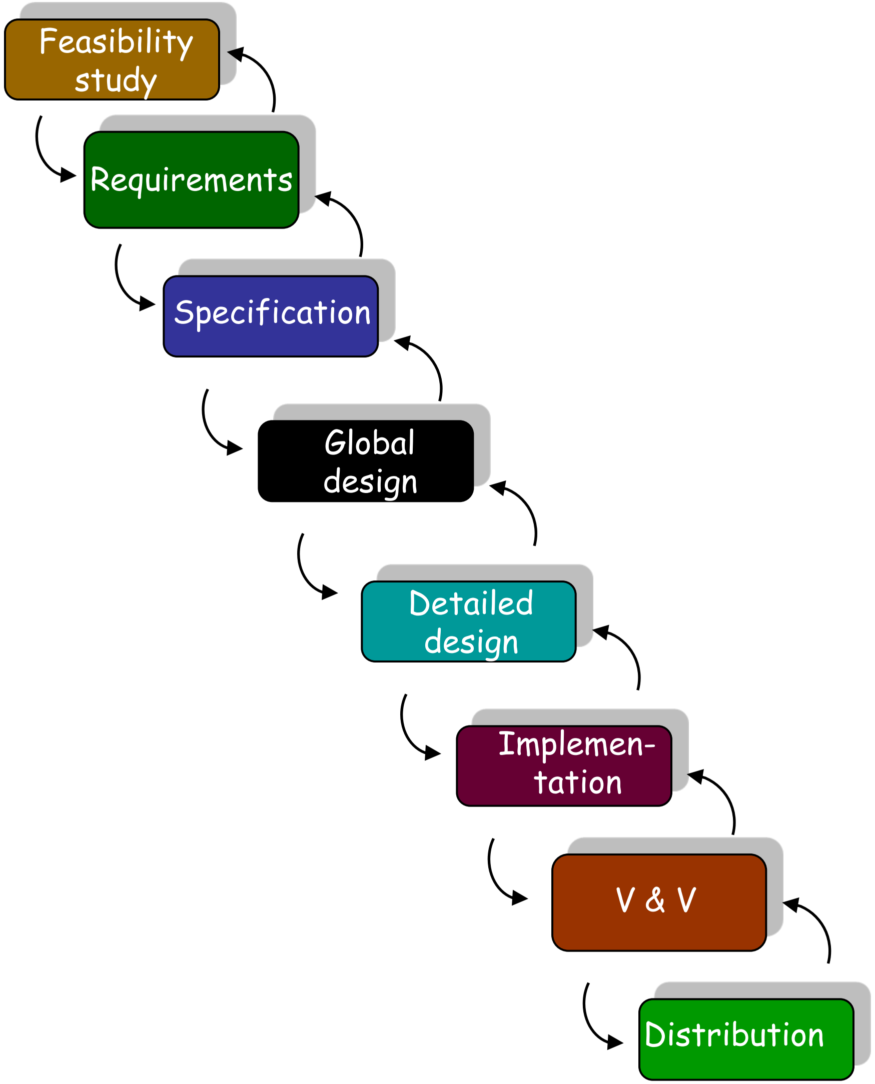


Figure 4. Unfortunately, for the process illustrated, the design iterations are never confined to the successive steps.

The waterfall model of the lifecycle



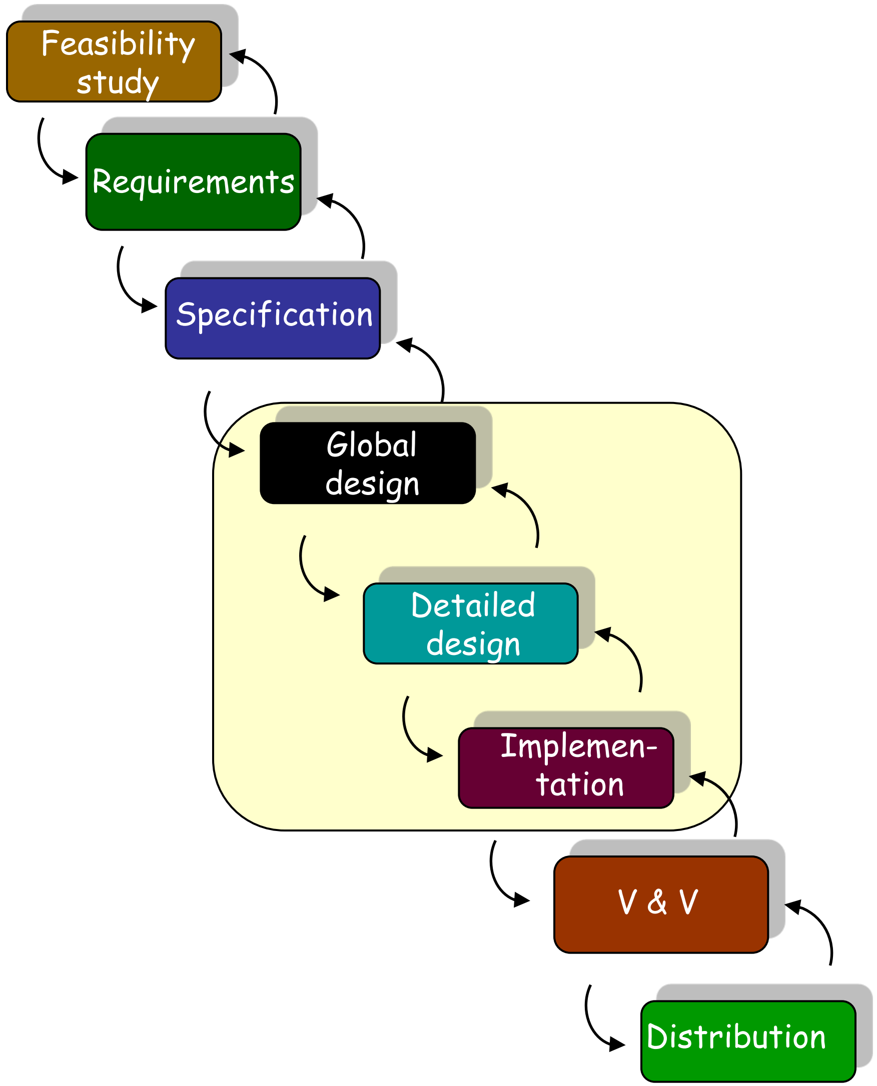
Arguments for the waterfall

Source: Boehm 81

(After B.W. Boehm: *Software engineering economics*)

- The activities are necessary
  - (But: merging of middle activities)
- The order is the right one.

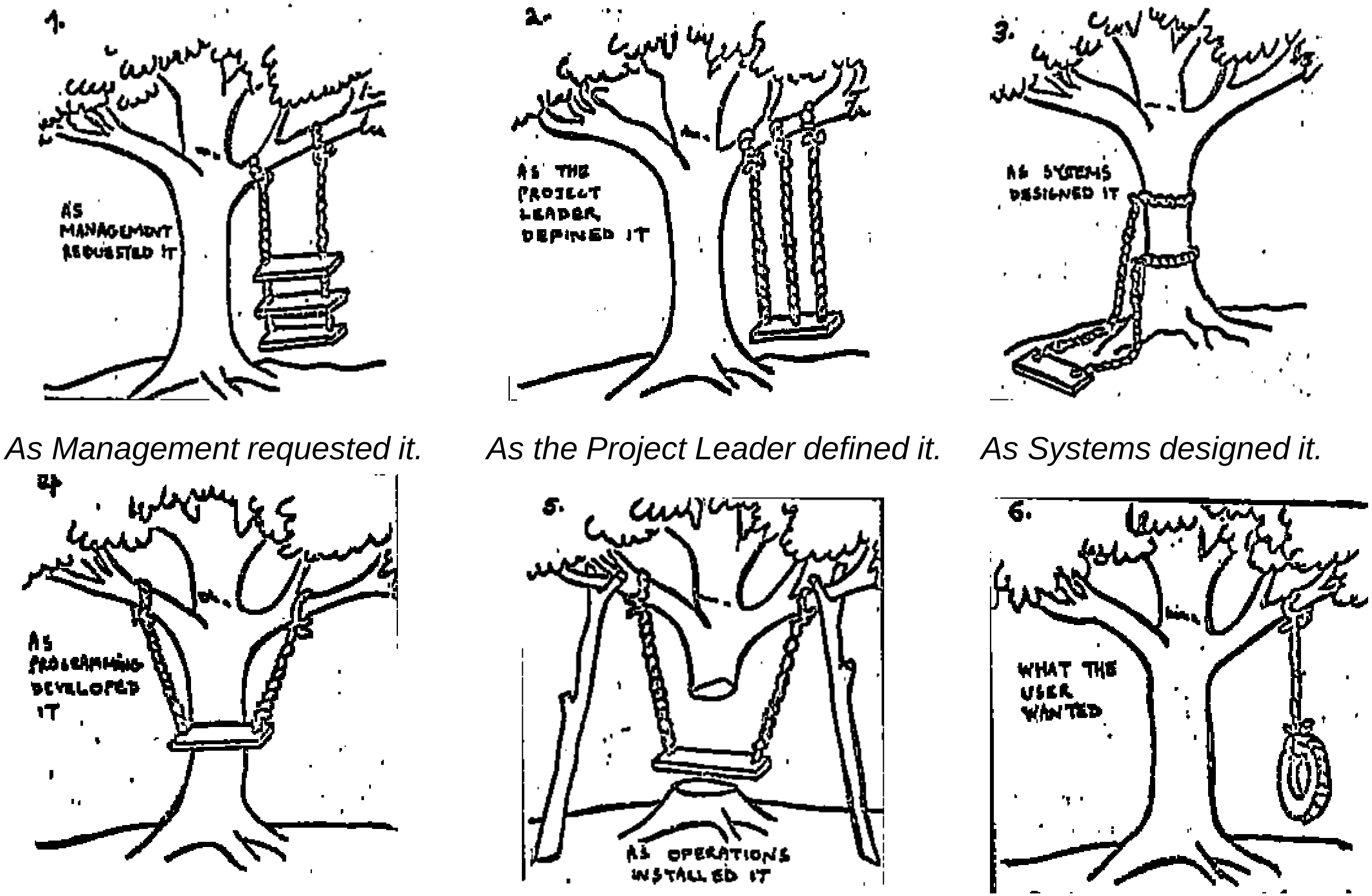
The waterfall model



Problems with the waterfall

- Late appearance of actual code
- Lack of support for requirements change — and more generally for extendibility and reusability
- Lack of support for the maintenance activity (70% of software costs?)
- Division of labor hampering Total Quality Management
- Impedance mismatches
- Highly synchronous model

Impedance mismatches

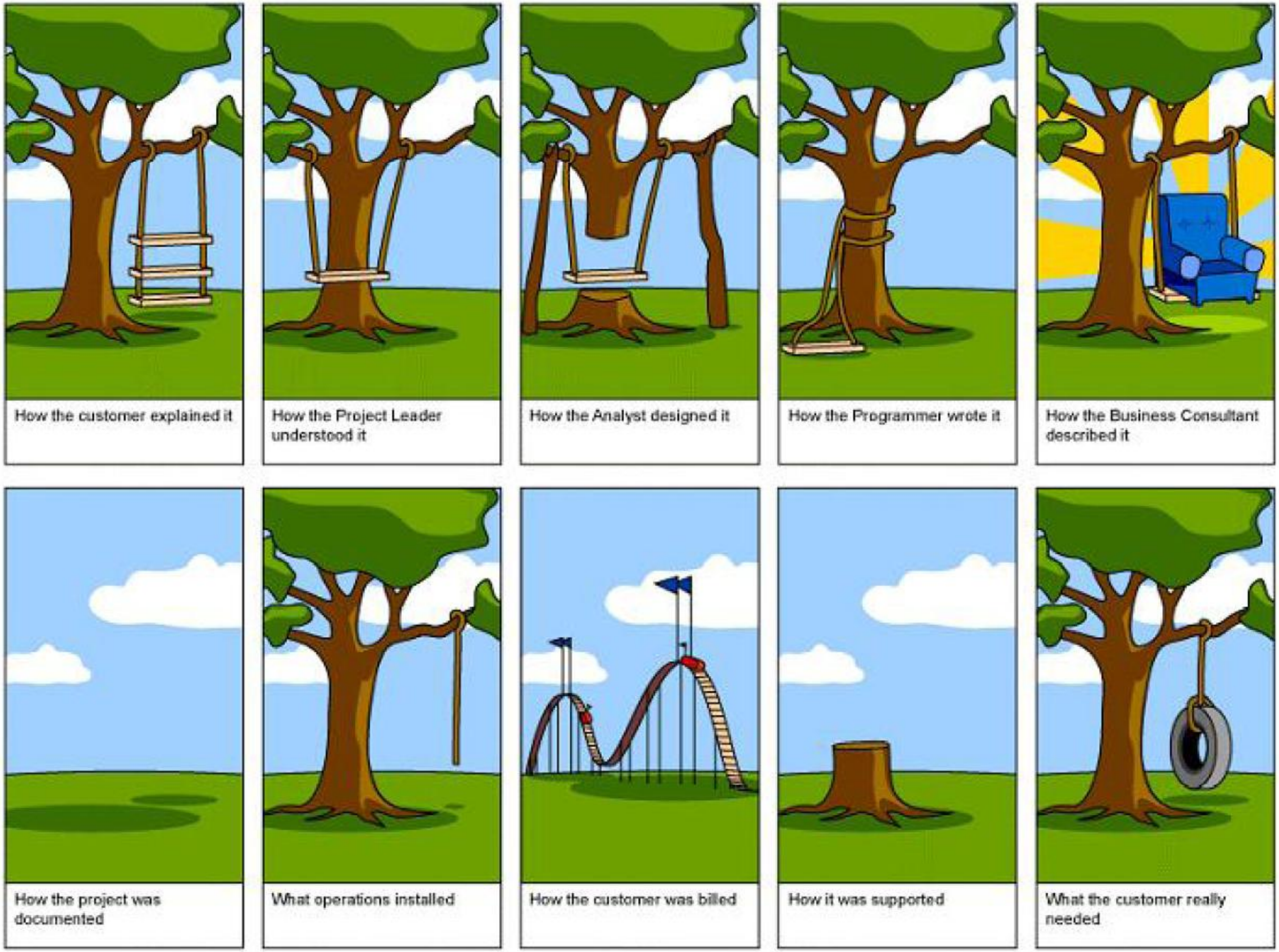

1.
AS MANAGEMENT REQUESTED IT
2.
AS THE PROJECT LEADER DEFINED IT
3.
AS SYSTEMS DESIGNED IT
4.
AS PROGRAMMING DEVELOPED IT
5.
AS OPERATIONS INSTALLED IT
6.
WHAT THE USER WANTED

As Management requested it. *As the Project Leader defined it.* *As Systems designed it.*

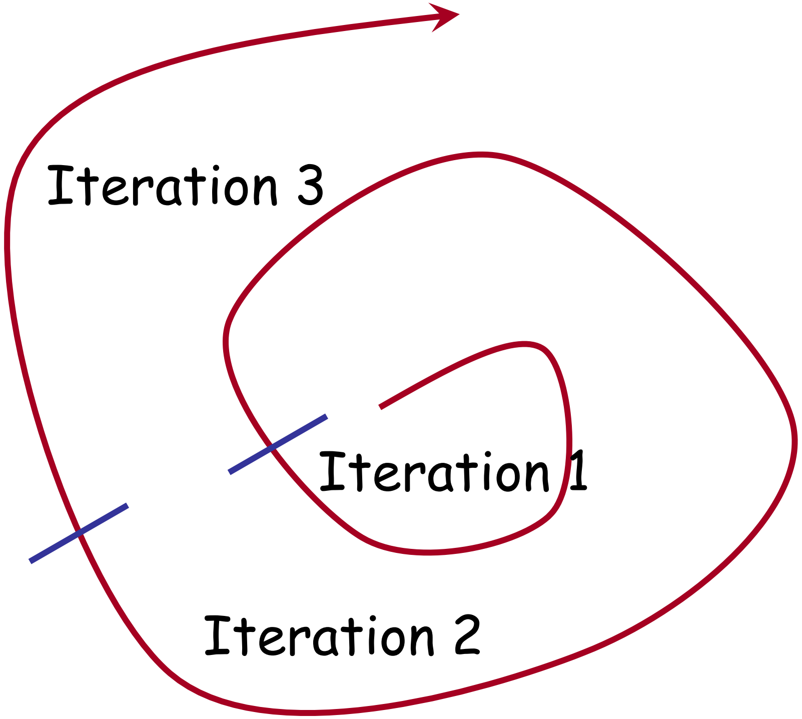
As Programming developed it. *As Operations installed it.* *What the user wanted.*

(Pre-1970 cartoon; origin unknown)

A modern variant

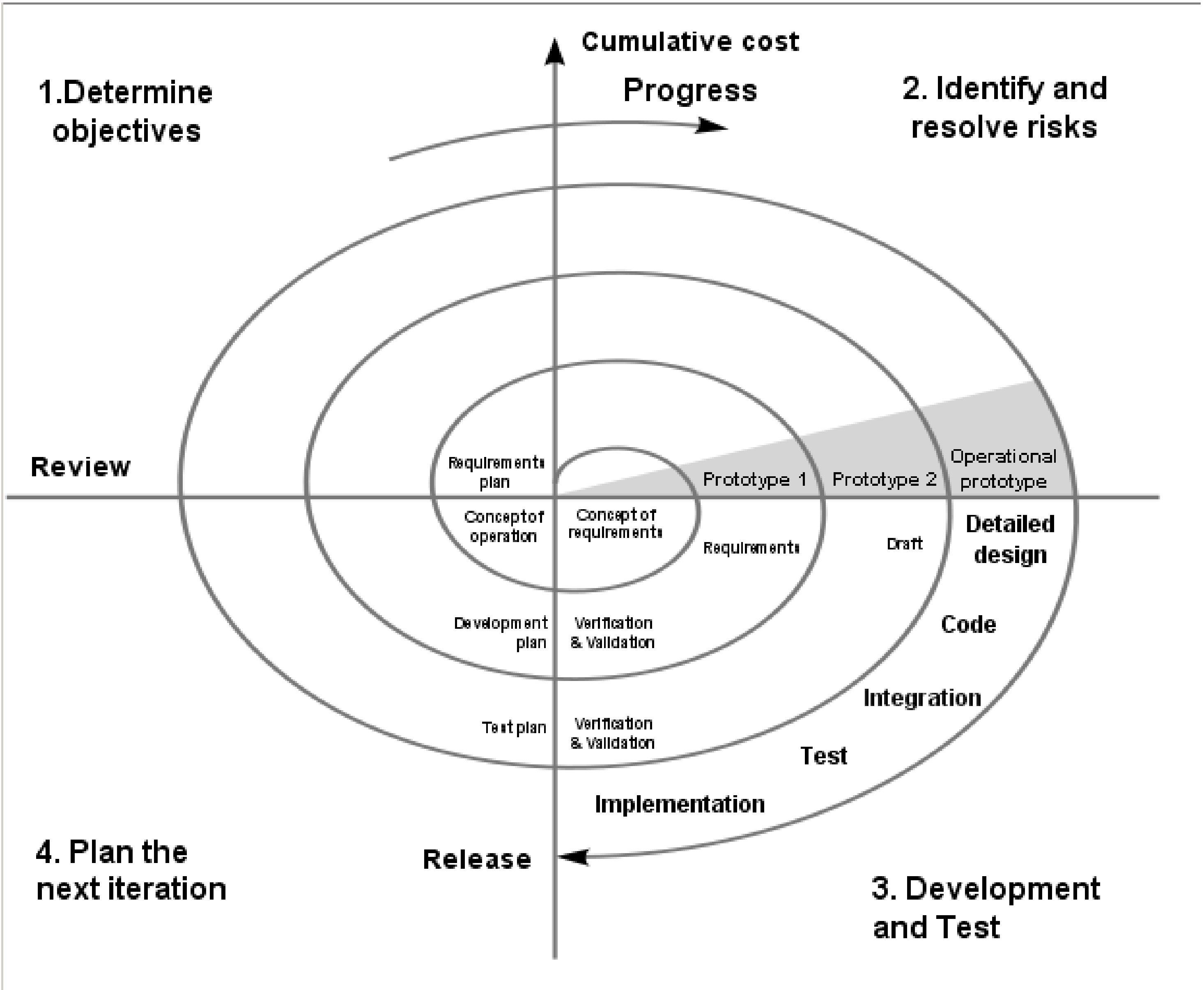

How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

The spiral model



The spiral model

Apply a waterfall-like approach to successive prototypes



CMMI background

Source for some of the CMMI material: Peter Kolb (from our ETH "Distributed and Outsourced Software Engineering" course)

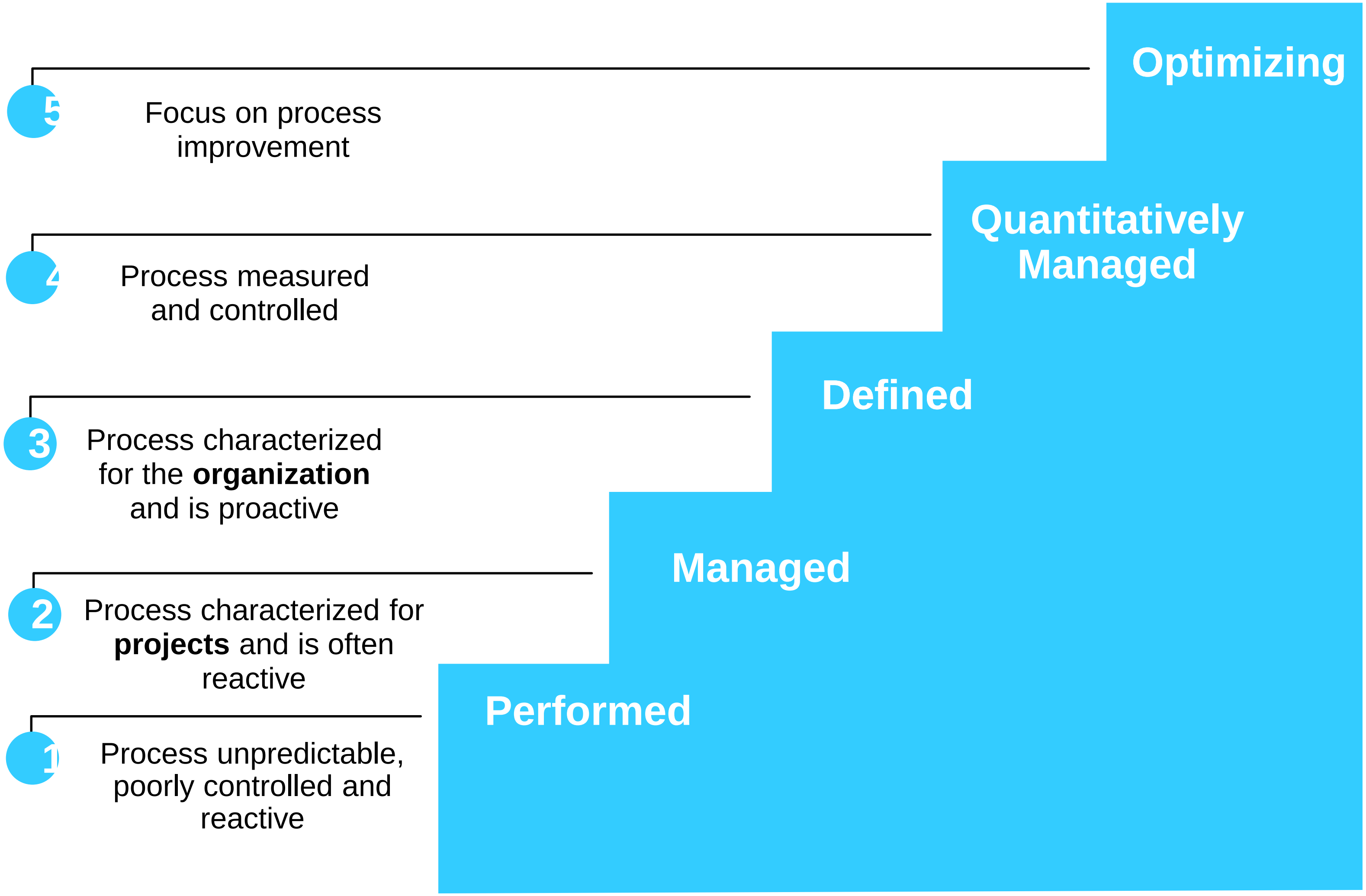
Initially: Capability Maturity Model (CMM), developed by Software Engineering Institute (at Carnegie-Mellon University, Pittsburgh) for the US Department of Defense, 1987-1997; meant for software

Widely adopted by Indian outsourcing companies

Generalized into CMMI (version 1.1 in 2002)

SEI itself offers assessments: SCAMPI (Standard CMMI Appraisal Method for Process Improvement)

CMMI maturity levels



Optimizing

5 Focus on process improvement

Quantitatively Managed

4 Process measured and controlled

Defined

3 Process characterized for the **organization** and is proactive

Managed

2 Process characterized for **projects** and is often reactive

Performed

1 Process unpredictable, poorly controlled and reactive

CMMI basic ideas

Basic goal: determine the maturity level of the **process** of an organization

Focused on process, not technology

Emphasizes **reproducibility** of results

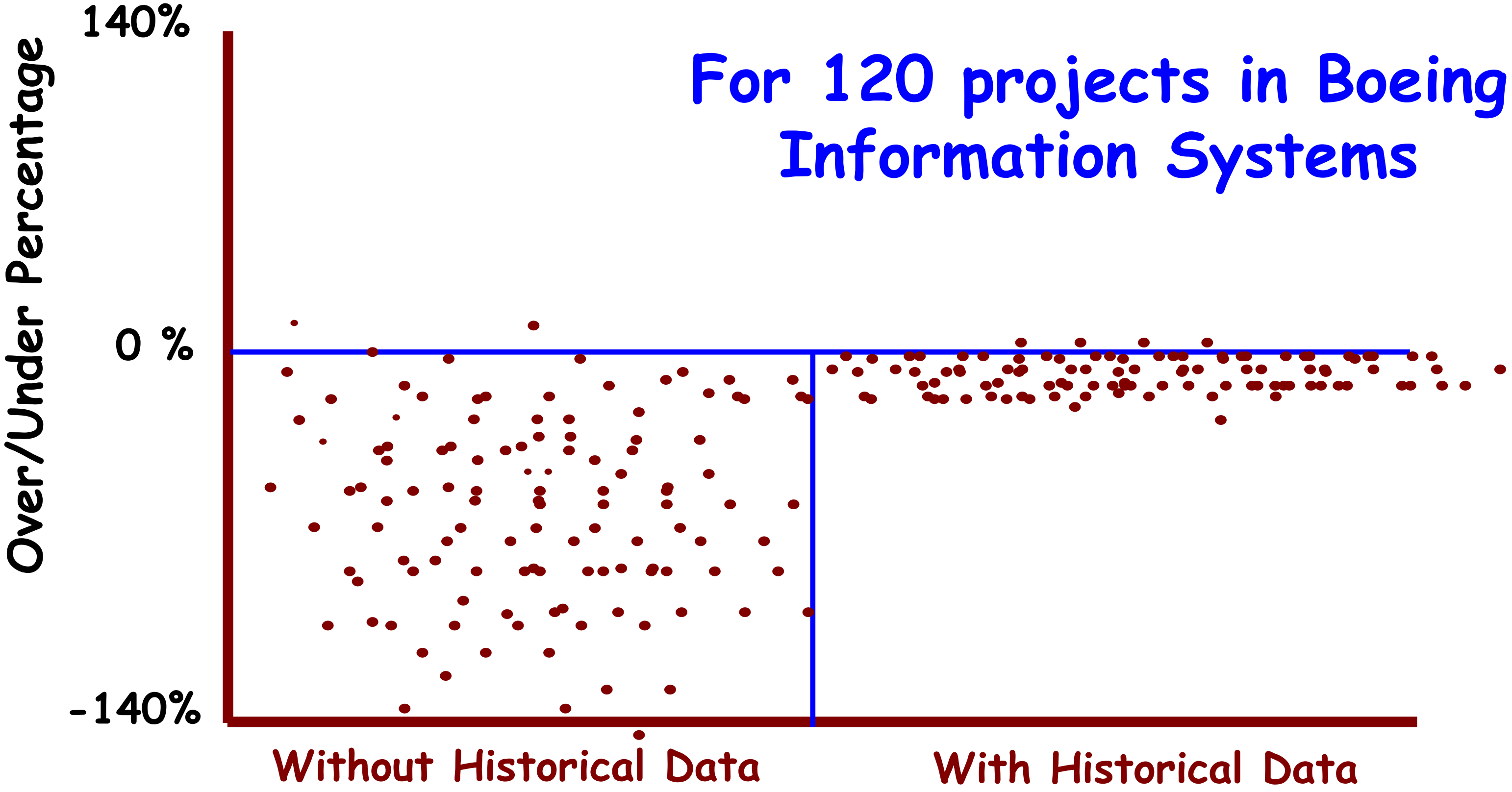
(Moving away from "heroic" successes to controlled processes)

Emphasizes **measurement**, based on statistical quality control techniques pioneered by W. Edward Deming & others

Relies on **assessment** by external team

Predictability

Source: Peter Kolb



For 120 projects in Boeing Information Systems

Variance: + 20% to -145%
(Mostly Level 1 & 2)

Variance:- 20% to + 20%
(Level 3)

John Vu: *Software Process Improvement Journey: From Level 1 to Level 5*, 7th SEPG Conference, 1997, see www.processgroup.com/john-vu-keynote2001.pdf

Generic goals and practices

Level	Capability Generic Goals	Generic Practices	
1	**Achieve Specific Goals**	GP 1.1	Perform Base Practices
2	**Institutionalize a Managed Process**	GP 2.1	Establish an Organizational Policy
		GP 2.2	Plan the Process
		GP 2.3	Provide Resources
		GP 2.4	Assign Responsibility
		GP 2.5	Train People
		GP 2.6	Manage Configurations
		GP 2.7	Identify and Involve Relevant Stakeholders
		GP 2.8	Monitor and Control the Process
		GP 2.9	Objectively Evaluate Adherence
		GP 2.10	Review Status with Higher Level Mgmt
3	**Institutionalize a Defined Process**	GP 3.1	Establish a Defined Process
		GP 3.2	Collect Improvement Information
4	**Institutionalize a Quantitatively Managed Process**		

CMMI: summary

Defines goals and practices shown to be useful to the software industry

Primarily directed to large organizations

Focus on process: explicit, documented, reproducible, measurable, self-improving

Essential to outsourcing industry

Technology-neutral

TSP, PSP

PSP: Personal Software Process

TSP: Team Software Process

Transposition of CMMI-like ideas to work of individual teams and developers

Management support

The initial TSP objective is to convince management to let the team be self-directed, meaning that it:

- Sets its own goals
- Establishes its own roles
- Decides on its development strategy
- Defines its processes
- Develops its plans
- Measures, manages, and controls its work

Management support

Source for PSP material: Software Engineering Institute

Management will support you as long as you:

- Strive to meet their needs
- Provide regular reports on your work
- Convince them that your plans are sound
- Do quality work
- Respond to changing needs
- Come to them for help when you have problems

Management support

Management will agree to your managing your own work as long as they believe that you are doing a superior job.

To convince them of this, you must:

- Maintain and publish precise, accurate plans
- Measure and track your work
- Regularly show that you are doing superior work

The PSP helps you do this

PSP essential practices

- Measure, track, and analyze your work
- Learn from your performance variations
- Incorporate lessons learned into your personal practices

What does a PSP provide?

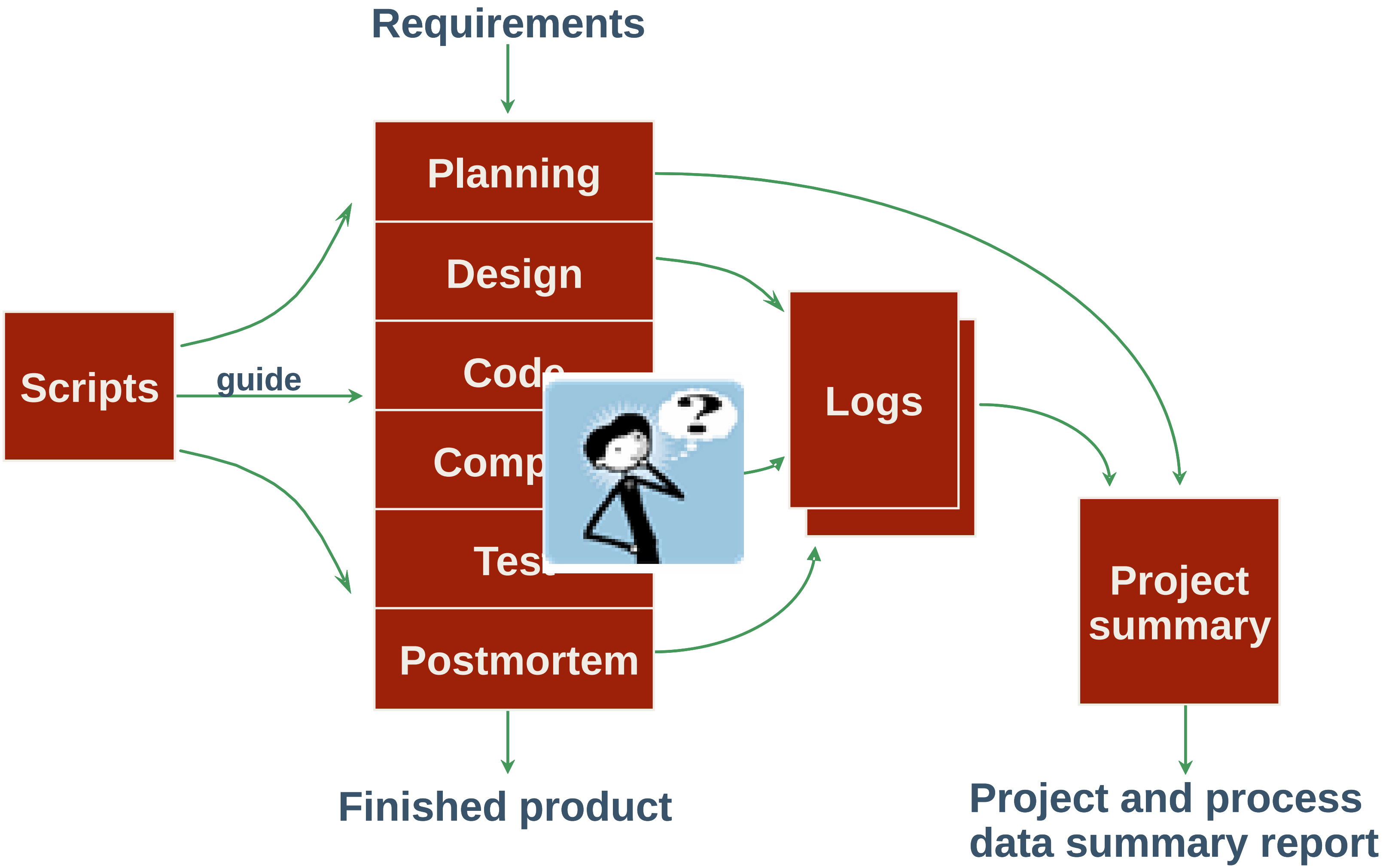
A stable, mature PSP allows you to

- Estimate and plan your work
- Meet your commitments
- Resist unreasonable commitment pressures

You will also

- Understand your current performance
- Improve your expertise as a professional

The PSP process flow



Arguments for reviews over tests

In testing, you must

- Detect unusual behavior
- Figure out what the test program was doing
- Find where the problem is in the program
- Figure out which defect could cause such beh

This can take a lot of time

With reviews you

- Follow your own logic
- Know where you are when you find a defect
- Know what the program should do, but did not
- Know why this is a defect
- Are in a better position to devise a correct fix

What does a PSP provide?

A stable, mature PSP allows you to

- Estimate and plan your work
- Meet your commitments
- Resist unreasonable commitment pressures

You will also

- Understand your current performance
- Improve your expertise as a professional

Code reviews

General principles (not specifically from PSP):

- Uncoupled from evaluation process
- Meeting must have chair, secretary
- Chair is not supervisor
- Purpose is to identify faults
- Purpose is not to correct them
- Purpose is not to evaluate developer; keep focus technical
- Strict time limit (e.g. 2 hours)
- Announced sufficiently long in advance
- Participant number: 5 to 10
- Code available in advance, as well as any other documents
- Meeting must be conducted professionally and speedily; chair keeps it focused

Code review checklist

Reviews are most effective with personal checklist customized to your own defect experience:

- Use your own data to select the checklist items
- Gather and analyze data on the reviews
- Adjust the checklist with experience

Do the reviews on a printed listing, not on screen

The checklist defines steps and suggests their order:

- Review for one checklist item at a time
- Check off each item as you complete it

Design review principles

In addition to reviewing code, you should also review your designs

Requires that you

- Produce designs that can be reviewed
- Follow an explicit review strategy
- Review design in stages
- Verify that logic correctly implements requirements

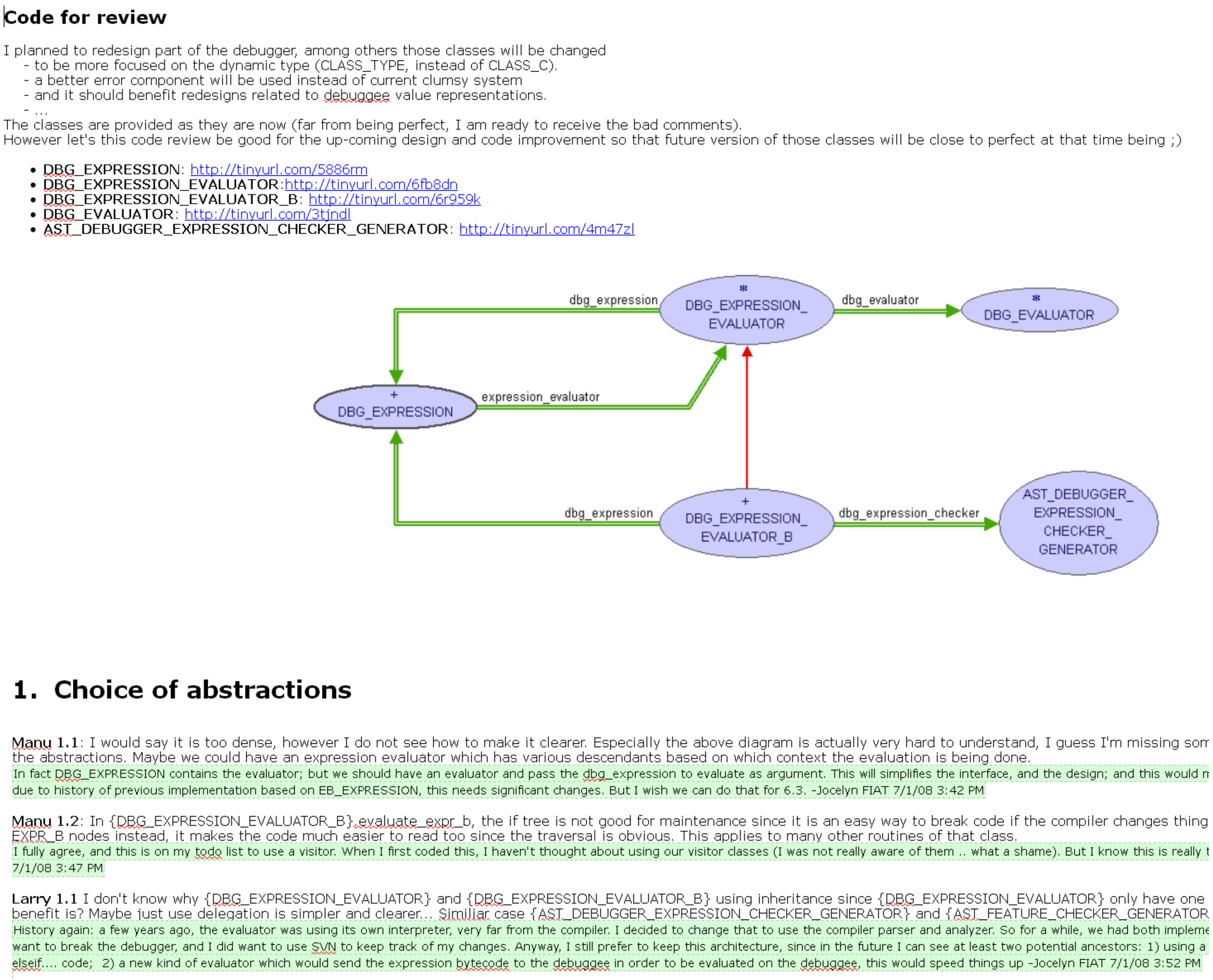
Digression: better code reviews

Source: Meyer 08

With the Web code reviews become much more interesting:

- Classes circulated three weeks in advance
- Comment categories: choice of abstractions, other aspects of API design,architecture choices, algorithms & data structures, implementation, programming style, comments & documentation
- Not just code, but design as well
- Comments in writing on Google Doc page, starting one week ahead
- Author of code responds on same page
- Meeting is devoted to unresolved issues

Distributed code review



Code for review

I planned to redesign part of the debugger, among others those classes will be changed
- to be more focused on the dynamic type (CLASS_TYPE, instead of CLASS_C).
- a better error component will be used instead of current clumsy system
- and it should benefit redesigns related to debuggee value representations.
- ...

The classes are provided as they are now (far from being perfect, I am ready to receive the bad comments).
However let's this code review be good for the up-coming design and code improvement so that future version of those classes will be close to perfect at that time being ;)

- DBG_EXPRESSION: http://tinyurl.com/5886rm
- DBG_EXPRESSION_EVALUATOR:http://tinyurl.com/6fb8dn
- DBG_EXPRESSION_EVALUATOR_B: http://tinyurl.com/6r959k
- DBG_EVALUATOR: http://tinyurl.com/3tjndl
- AST_DEBUGGER_EXPRESSION_CHECKER_GENERATOR: http://tinyurl.com/4m47zl

1. Choice of abstractions

Manu 1.1: I would say it is too dense, however I do not see how to make it clearer. Especially the above diagram is actually very hard to understand, I guess I'm missing som
the abstractions. Maybe we could have an expression evaluator which has various descendants based on which context the evaluation is being done.
In fact DBG_EXPRESSION contains the evaluator; but we should have an evaluator and pass the dbg_expression to evaluate as argument. This will simplifies the interface, and the design; and this would m
due to history of previous implementation based on EB_EXPRESSION, this needs significant changes. But I wish we can do that for 6.3. -Jocelyn FIAT 7/1/08 3:42 PM

Manu 1.2: In {DBG_EXPRESSION_EVALUATOR_B}.evaluate_expr_b, the if tree is not good for maintenance since it is an easy way to break code if the compiler changes thing
EXPR_B nodes instead, it makes the code much easier to read too since the traversal is obvious. This applies to many other routines of that class.
I fully agree, and this is on my todo list to use a visitor. When I first coded this, I haven't thought about using our visitor classes (I was not really aware of them .. what a shame). But I know this is really t
7/1/08 3:47 PM

Larry 1.1 I don't know why {DBG_EXPRESSION_EVALUATOR} and {DBG_EXPRESSION_EVALUATOR_B} using inheritance since {DBG_EXPRESSION_EVALUATOR} only have one
benefit is? Maybe just use delegation is simpler and clearer... Similiar case {AST_DEBUGGER_EXPRESSION_CHECKER_GENERATOR} and {AST_FEATURE_CHECKER_GENERATOR
History again: a few years ago, the evaluator was using its own interpreter, very far from the compiler. I decided to change that to use the compiler parser and analyzer. So for a while, we had both impleme
want to break the debugger, and I did want to use SVN to keep track of my changes. Anyway, I still prefer to keep this architecture, since in the future I can see at least two potential ancestors: 1) using a
elseif.... code; 2) a new kind of evaluator which would send the expression bytecode to the debuggee in order to be evaluated on the debuggee, this would speed things up -Jocelyn FIAT 7/1/08 3:52 PM

Review categories (end of digression)

1. Choice of abstractions
2. Other aspects of API design
3. Contracts
4. Other aspects of architecture, e.g. choice of client links, inheritance hierarchies
5. Implementation, in particular choice of data structures and algorithms
6. Programming style
7. Comments and documentation (including indexing/note clauses)
8. Global comments
9. Actions based on this code review

PSP: an assessment

Ignore technology assumptions (strict design-code-compile-test cycle) which is not in line with today's best practices.

Retain emphasis on professional engineer's approach:

- Plan
- Record what you do both qualitatively and quantitatively:
  - Program size
  - Time spent on parts and activities
  - Defects
- Think about your personal process
- Improve your personal process

Tool support, integrated in IDE, is essential

Topics

Part 1: Overview

Part 2: The “enemy”: Big Upfront Everything

Part 3: What is “agile”?

- 3.1 Agile principles
- 3.2 Agile roles
- 3.3Agile tools and artifacts
- 3.4 Agile practices
- 3.5 Agile methods

Part 4: Critical analysis

- 4.1 Conceptual
- 4.2 Empirical

Part 5: Conclusion

Complementary material: Bibliography

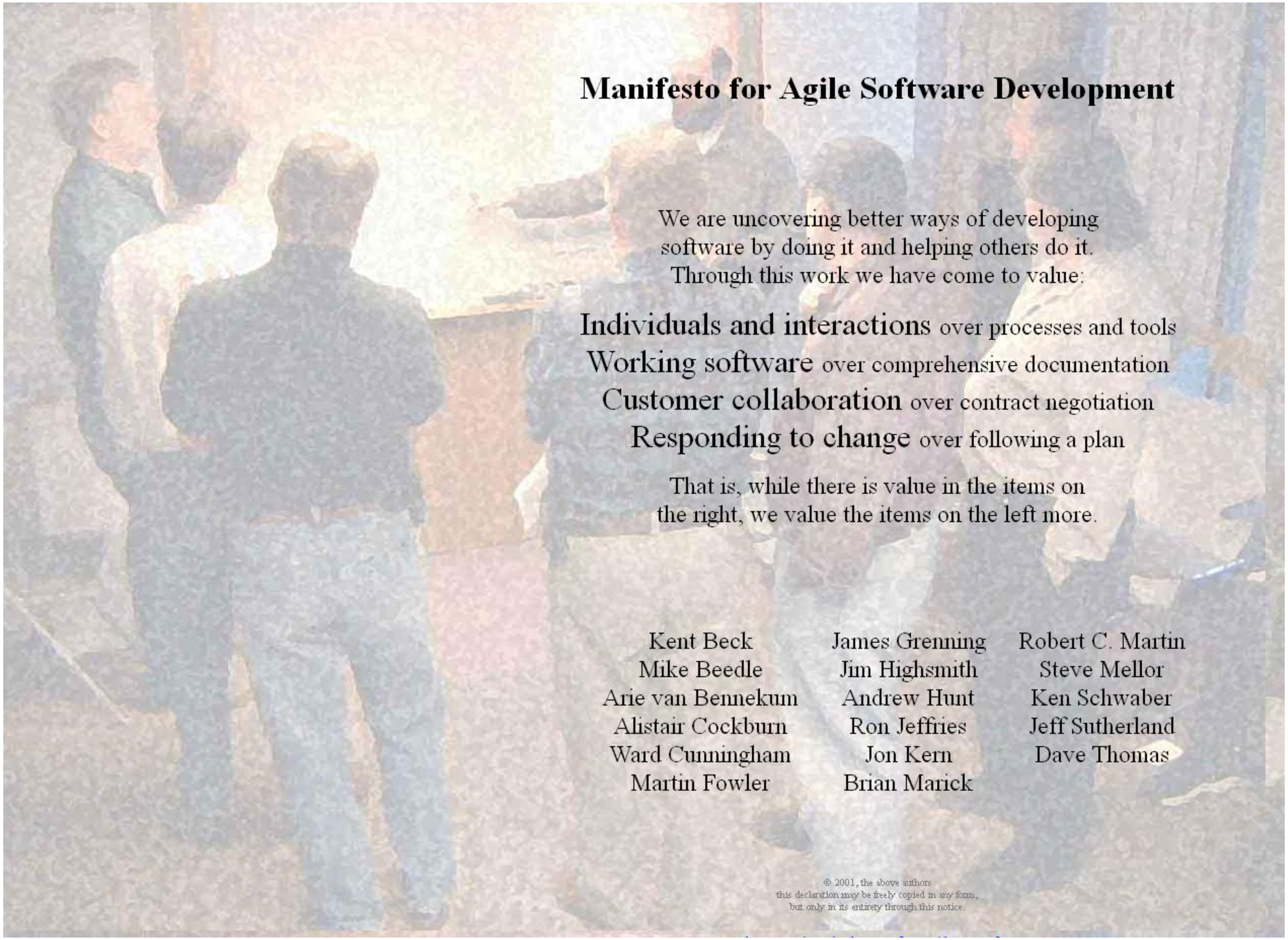
3

What is agile?

- 3.1 -

Agile principles

Agile manifesto



Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler

James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick

Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

Twelve principles

Source: Agile manifesto

We follow these principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

My view: agile assumptions

- **A** New, reduced role for manager
- **B** No "Big Upfront" steps
- **C** Iterative development
- **D** Limited, negotiated scope
- **E** Focus on quality, achieved through testing

My view: agile principles

Organizational

- 1 Place the customer at the center
- 2 Develop minimal software:
  - 2.1 Produce minimal functionality
  - 2.2 Produce only the product requested
  - 2.3 Develop only code and tests
- 3 Accept change
- 4 Let the team self-organize
- 5 Maintain a sustainable pace

Technical

- 6 Produce frequent working iterations
- 7 Treat tests as a key resource:
  - 7.1 Do not start any new development until all tests pass
  - 7.2 Test first
- 8 Express requirements through scenarios

Negotiated scope contract XP

Source: Beck 05

"Write contracts for software development that fix time, costs, and quality but call for an ongoing negotiation of the precise scope of the system. Reduce risk by signing a sequence of short contracts instead of one long one."

You can move in the direction of negotiated scope. Big, long contracts can be split in half or thirds, with the optional part to be exercised only if both parties agree. Contracts with high costs for change requests can be written with less scope fixed up front and lower costs for changes"

Favor verbal communication

Scrum

Source: Cohn (slightly abridged)

There is a grand myth about requirements—if you write them down, users will get exactly what they want. Not true. At best, users will get exactly what was written down, which may or may not be anything like what they really want.

Written words are misleading—they look more precise than they are. Recently, to run a course, I emailed my assistant "Please book the Denver Hyatt". She emailed "The hotel is booked". I mailed back "Thanks".

A week later she emailed "The hotel is booked the days you wanted. Should I try another hotel? Another week? Another city?" We had miscommunicated about "booked." When she wrote "the hotel is booked" she meant, "The Hyatt room is already taken." I read "booked" as a confirmation that she had booked the hotel. Neither of us did anything wrong. Rather, this is an example of how easy it is to miscommunicate, especially in writing. Had we been talking, I would have thanked her when she said "the hotel is booked." My happy voice would have confused her, and we would have caught our miscommunication right then.

Beyond this problem there are other reasons to favor discussions over documents.

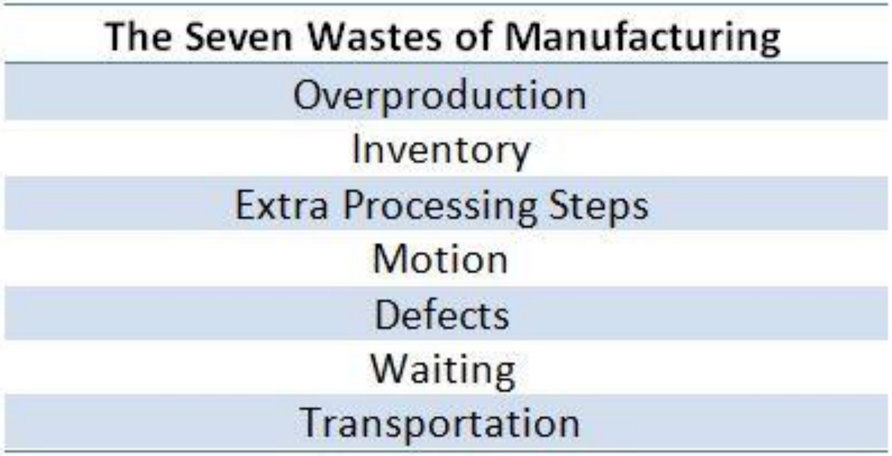
Eliminate waste

Lean

Source: Poppendieck

Everything not adding value to the customer is considered waste:

- Unnecessary code
- Unnecessary functionality
- Delay in process
- Unclear requirements
- Insufficient testing
- Avoidable process repetition
- Bureaucracy
- Slow internal communication
- Partially done coding
- Waiting for other activities, team, processes
- Defects, lower quality
- Managerial overhead



The Seven Wastes of Manufacturing
Overproduction
Inventory
Extra Processing Steps
Motion
Defects
Waiting
Transportation

Value stream mapping: strategy to recognize waste. Eliminate it iteratively

Minimize artifacts (inventory) Lean

Source: Poppendieck

Inventory is waste; advertised benefits not worth the costs:

- Consumes resources
- Slows down response time
- Hides quality problems
- Gets lost
- Degrades and becomes obsolete

In software: inventory is documentation that is not a part of the final program, e.g.

- Requirements documents
- Design documents

Risk: building the wrong system if these documents do not capture true user needs. Even if they do now, they will not necessarily remain valid in the future.

As inventory must be minimized to maximize manufacturing flow, requirements & design documents must be minimized to maximize development flow

Best approach: raise level of abstraction of documentation. Instead of a 100 page detailed specification, write a 10 page set of rules and guidelines, and document only the exceptions

Amplify learning

Lean

Source: Poppendieck

Software development is a continuous learning process

The best approach for improving a software development environment is to amplify learning and speed up the learning process:

- To prevent accumulation of defects, run tests as soon as the code is written
- Instead of adding documentation or planning, try different ideas by writing and testing code and building
- Present screens to end-users and get their input
- Enforce short iteration cycles, each including refactoring and integration testing
- Set up feedback sessions with customers

Set-based development: concentrate on communicating the constraints of the future solution and not the possible solutions, to promote dialog with the customer in devising the solution

Focus Crystal

Focus on individual task, to ensure progress:

- Control flow of progress
- Deal with interruptions:
  - Two-hour period without interruption
  - Assign developer to project for at least two days before switching

Focus on direction of project

- Define goals clearly
- Prioritize goals

Decide as late as possible

Lean

Source: Poppendieck

Delay decisions as much as possible until they can be made based on facts, not assumptions, and customers better understand their needs

The more complex a system, the more capacity for change should be built in

Use iterative approach to adapt to changes and correct mistakes, which might be very costly if discovered after system release

Planning should be involved, but concentrates on the different options and adapting to the current situation, as well as clarifying confusing situations by establishing patterns for rapid action

Evaluating different options is effective, but only if they provide the needed flexibility for late decision making

Deliver as fast as possible

Lean

Scrum

Source: Poppendieck

It is not the biggest that survives, but the fastest

The sooner the end product is delivered, the sooner feedback can be received, and incorporated into the next iteration

For software, the Just-in-Time production ideology means presenting the needed result and letting the team organize itself to obtain it in a specific iteration

At the beginning, the customer provides the needed input. This could be simply presented in small cards or stories — the developers estimate the time needed for the implementation of each card

The work organization changes into self-pulling system — each morning during a stand-up meeting, each member of the team reviews what has been done yesterday, what is to be done today and tomorrow, and prompts for any inputs needed from colleagues or the customer

Minimize dependencies

Scrum

Source: Sutherland

Scrum asserts that it is possible to remove dependencies between user stories, so that at any point any user story can be selected according to the proper criteria (maximizing business value)

Multiple design

Lean

Source: Poppendieck

Another key idea fromToyota is set-based design. If a new brake system is needed, three teams may design solutions to the problem

If a solution is deemed unreasonable, it is cut

At period end, the surviving designs are compared and one chosen, perhaps with modifications based on learning from the others — an example of deferring commitment until the last possible moment

Software decisions could also benefit from this practice to minimize the risk brought on by big up-front design

Build in integrity

Lean XP

Source: Poppendieck

The customer needs an overall experience of the System: how it is advertised, delivered, deployed, used, how well it solves problems

- Conceptual integrity means that the system's separate components work well together as a whole
- To this end, the information flow should be constant from customer to developers and back, avoiding large stressful amount of information after long development in isolation
- One of the healthy ways towards integral architecture is refactoring
- The more features are added to the system, the more loose the starting code base for further improvements. Refactoring is about keeping simplicity, clarity, minimum amount of features in the code
- At the end the integrity should be verified with thorough testing, thus ensuring the System does what the customer expects it to
- Automated tests are also part of the production process: if they do not add value they are waste

See the whole

Lean

Source: Poppendieck

Software systems are the product of their interactions

Defects accumulate during the development process

The root causes of defects should be found and eliminated

The larger the system and the more organizations involved, the greater the importance of well defined relationships between vendors, to ensure smooth component interactions

A strong sub- contractor network with win-win relationships is more beneficial than short-term profit optimizing

"Think big, act small, fail fast; learn rapidly"

Sustainable pace

XP Scrum Crystal

People perform best if they are not overstressed

Developers should not work more than 40 hour weeks,

If there is overtime or week-end work one week, there should not be any in the next week

XP avoids "crunch time" of traditional projects thanks to short release cycles

To help achieve these goals:

- Frequent code-merge
- Always maintain executable, test-covered, high-quality code
- Constant refactoring, helping keep fresh and alert minds
- Collaborative style
- Constant testing

Energized work

XP

See "sustainable pace"

No overtime

XP

See “sustainable pace”

Yourdon, “Death March” (1999)

Personal safety

Crystal

Encourage free expression of ideas

Do not ridicule anyone because of a question or suggestion

Build trust within the team

Humanity

XP

Source: Beck 05

Recognize that software is developed by people

Offer developers what they expect:

- Safety
- Accomplishment
- Belonging
- Growth
- Intimacy

Agile approaches are indebted here to DeMarco's and Lister's *Peopleware* (see bibliography)

Reflective improvement

Crystal

Developers must take breaks from regular development to look for ways to improve the process

Iterations help with this by providing feedback on whether or not the current process is working

Empower the team

Lean XP Scrum

Source: Poppendieck

Traditional view: managers tell workers to do their job

Agile view: managers listen to developers, explain possible actions, provide suggestions for improvements.

“Find good people & let them do their own job“. The leader is there to:

- Encouraging progress
- Help catch errors
- Remove impediments
- Provide support and help in difficult situations
- Make sure that skepticism does not ruin the team's spirit
- Avoid in micro-management

In software development people are not resources. They need motivation and a higher purpose.

Team chooses own commitments

Team has access to customers

Sit together

XP

XP promotes an open workspace:

- Organized around pairing stations
- With whiteboard space
- Locating people according to conversations they should overhear
- With room for personal effects
- With a place for private conversations

Expected benefits: improve communication, resolve problems quickly with the benefits of face-to-face interaction (as opposed to e.g. email)

Informative workspace XP Scrum

Facilitate communication through well-organized workspace:

- Story board with user story cards movable from *not started* to *in progress* to *done* column
- Release charts
- Iteration burndown charts
- Automated indicators showing the status of the latest unit-testing run
- Meeting room with visible charts, whiteboards and flipcharts

Team continuity

XP

Keep the team together and stable

Do not reassign people to other teams or treat them as mere resources

Shrinking teams

XP

Source: Beck 05

As a team grows in capability, keep its workload constant but gradually reduce its size

This frees people to form more teams

When the team has too few members, merge it with another too-small team

Customer always available XP

Source: Wells

"One of the few absolute requirements of Extreme Programming"

All project phases require communication with customer, preferably face to face. Recommended technique: assign one or more customers to the development team.

Projects of significant size require full-time commitment from customers, who:

- Write user stories, with developer help, to allow time estimates & assign priority
- Help make sure most of the desired functionality is covered by stories
- Provide further functional details as user stories are incomplete
- During planning meeting, negotiate selection of user stories for each release
- Negotiates release timing (use release planning meeting for this purpose)
- Make all decisions that affect their business goals
- Try system early to provide feedback
- Help with functional testing: review test score
- Allow the system to continue into production or stop it
- Because details are left off the user stories the developers will need to talk with customers to get enough detail to complete a programming task.

"This may seem like a lot of the customer's time but the customer's time is saved initially by not requiring a detailed requirements specification and later by not delivering an uncooperative system"

Resolve conflicts between customers by having them participate in group meetings

Customer involvement XP

On-site customer:

- Makes sure team understands customer wishes
- Talks to developer, clarifying feature wishes
- Specifies functional tests for user stories
- Participates in planning of iterations and releases
- Maintains contact with management

Leave optimization till last XP

Source: Wallace 02

According to XP you should always wait until you have finished a story and run your tests before you try to optimize your work

Only then can you analyze what exactly it is that needs optimizing

Do not make work for yourself by trying to anticipate problems before they exist; instead, wait until you have the results of your analysis before you focus on resolving whatever issues arise

All code must have unit tests XP

Core idea of XP:

- Do not write code without associated unit tests
- Do not proceed (with release, with next iteration) unless all unit tests pass

All code must pass unit tests before release

XP

Code that does not pass tests is waste

Code the unit test first XP

Source: Ron Jeffries

"Here is a really good way to develop new functionality:

- 1. Find out what you have to do.
- 2. Write a UnitTest for the desired new capability. Pick the smallest increment of new capability you can think of.
- 3. Run the UnitTest. If it succeeds, you're done; go to step 1, or if you are completely finished, go home.
- 4. Fix the immediate problem: maybe it's the fact that you didn't write the new method yet. Maybe the method doesn't quite work. Fix whatever it is. Go to step 3.

A key aspect of this process: don't try to implement two things at a time, don't try to fix two things at a time. Just do one.

When you get this right, development turns into a very pleasant cycle of testing, seeing a simple thing to fix, fixing it, testing, getting positive feedback all the way.

Guaranteed flow. And you go so fast!

Try it, you'll like it."

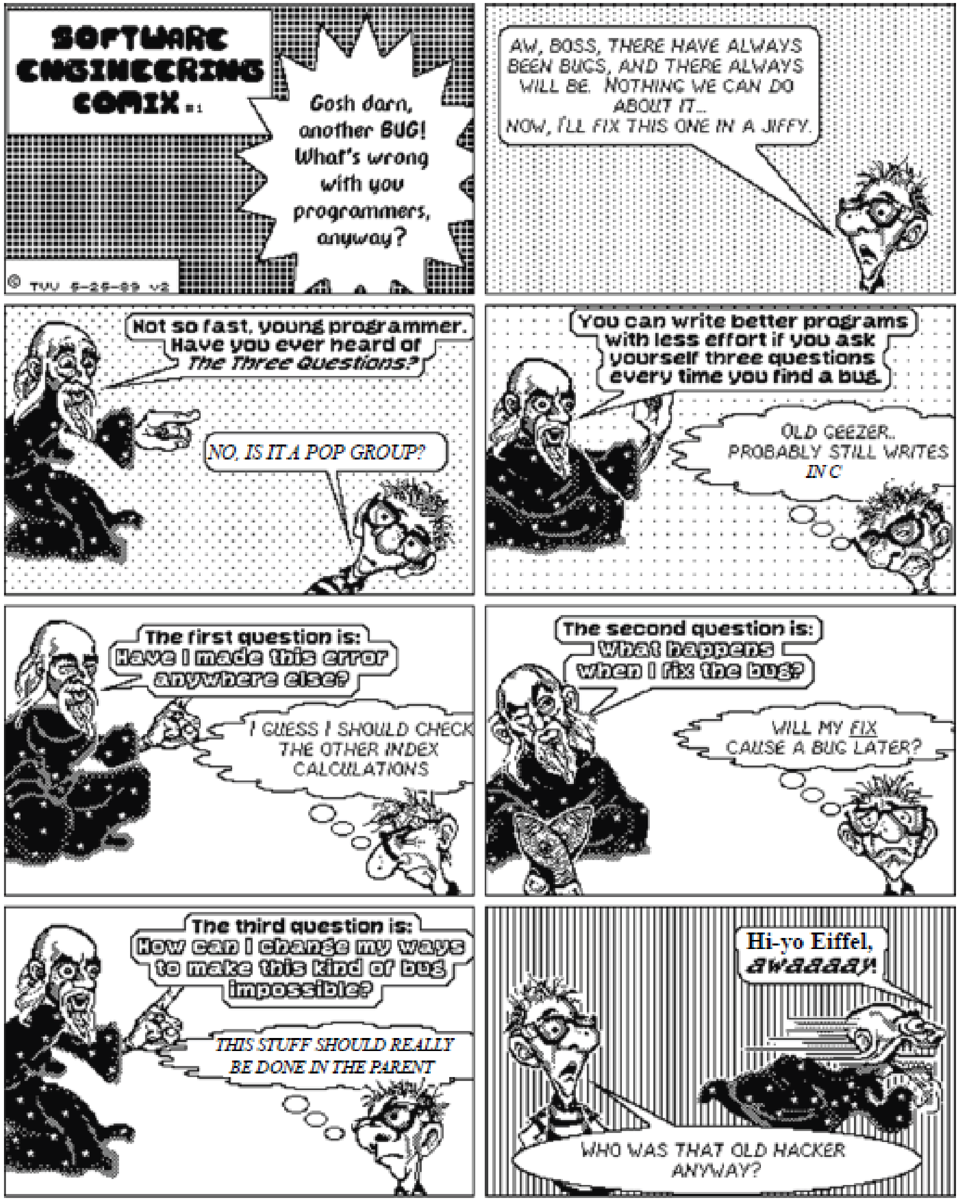
When bug found, create test before fixing it

XP

“A bug is not an error
in logic,
it is a test you forgot
to write”

Root-cause analysis XP Scrum

Every time a defect is found, do not just fix it but analyze its cause and make sure to correct that cause, not just the symptom


SOFTWARE ENGINEERING COMIX #1
Gosh darn, another BUG! What's wrong with you programmers, anyway?
© TVV 5-25-89 v2
AW, BOSS, THERE HAVE ALWAYS BEEN BUGS, AND THERE ALWAYS WILL BE. NOTHING WE CAN DO ABOUT IT... NOW, I'LL FIX THIS ONE IN A JIFFY.
Not so fast, young programmer. Have you ever heard of The Three Questions?
NO, IS IT A POP GROUP?
You can write better programs with less effort if you ask yourself three questions every time you find a bug.
OLD GEEZER.. PROBABLY STILL WRITES IN C
The first question is: Have I made this error anywhere else?
I GUESS I SHOULD CHECK THE OTHER INDEX CALCULATIONS
The second question is: What happens when I fix the bug?
WILL MY FIX CAUSE A BUG LATER?
The third question is: How can I change my ways to make this kind of bug impossible?
THIS STUFF SHOULD REALLY BE DONE IN THE PARENT
Hi-yo Eiffel, awaaaay!
WHO WAS THAT OLD HACKER ANYWAY?

Tom van Vleck, Software Engineering Notes, July 1989, adapted in Meyer 2009

Run acceptance tests often and publish results

XP

Source: Wells

Acceptance tests are black box system tests. Each acceptance test represents some expected result from the system

Acceptance tests should be automated so they can be run often

The acceptance test score is published to the team

It is the team's responsibility to schedule time each iteration to fix any failed tests

Only one pair integrates code at a time XP

Collective code ownership

Development proceeds in parallel

But: to avoid conflicts, only one pair is permitted to integrate its changes at any given time

System metaphor

XP

Source: Tomayko 03

"A metaphor is meant to be agreed upon by all members of a project as a means of simply explaining the purpose of the project and thus guide the structure of the architecture"

Benefits:

- Communication, including between customers & developers
- Clarify project, explain functionality
- Favors simple design
- Helps find common vocabulary

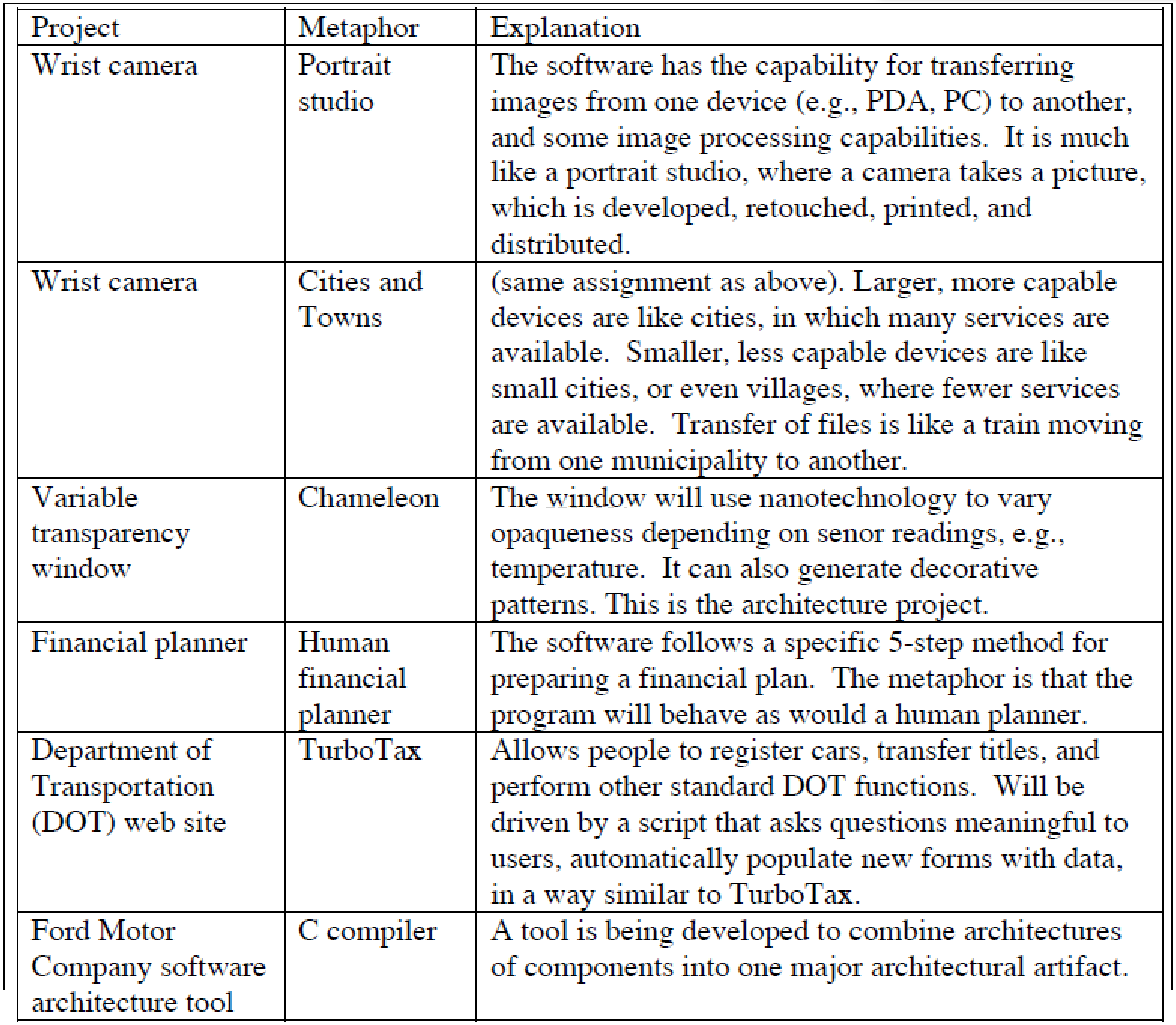
For a financial software tool:

- Bad: "check writer"
- Better: "financial advisor"

Example metaphors

XP

Source: Tomayko 03



Project	Metaphor	Explanation
Wrist camera	Portrait studio	The software has the capability for transferring images from one device (e.g., PDA, PC) to another, and some image processing capabilities. It is much like a portrait studio, where a camera takes a picture, which is developed, retouched, printed, and distributed.
Wrist camera	Cities and Towns	(same assignment as above). Larger, more capable devices are like cities, in which many services are available. Smaller, less capable devices are like small cities, or even villages, where fewer services are available. Transfer of files is like a train moving from one municipality to another.
Variable transparency window	Chameleon	The window will use nanotechnology to vary opaqueness depending on senor readings, e.g., temperature. It can also generate decorative patterns. This is the architecture project.
Financial planner	Human financial planner	The software follows a specific 5-step method for preparing a financial plan. The metaphor is that the program will behave as would a human planner.
Department of Transportation (DOT) web site	TurboTax	Allows people to register cars, transfer titles, and perform other standard DOT functions. Will be driven by a script that asks questions meaningful to users, automatically populate new forms with data, in a way similar to TurboTax.
Ford Motor Company software architecture tool	C compiler	A tool is being developed to combine architectures of components into one major architectural artifact.

Incremental design XP

Source: Shore 08

Developers work in small steps, validating each before moving to the next. Three parts:

- Start by creating the simplest design that could possibly work
- Incrementally add to it as the needs of the software evolve
- Continuously improve design by reflecting on its strengths and weaknesses

" *When you first create a design element, be completely specific. Create a simple design that solves only the problem you face, no matter how easy it may seem to solve more general problems.*

This is hard! Experienced programmers think in abstractions. The ability to think in abstractions is often a sign of a good programmer. Coding for one specific scenario will seem strange, even unprofessional. Waiting to create abstractions will enable you to create designs that are simple and powerful. Do it anyway."

Incremental deployment

XP

Scrum

Deploy functionality gradually

“Big Bang” deployment is risky

Quarterly cycle

A recommendation to have regular reviews of high level system structure, goals and priorities on a quarterly basis, matching the financial reporting practices of many companies

Also an opportunity to reflect on the team practices and state of mind, and discuss any major changes in practices and tools

Period chosen as large enough not to interfere with current concerns, and short enough to allow frequent questioning of practices and updates of long-term goals

Weekly cycle

XP

Source: Beck

Plan work a week at a time. Have a meeting at the beginning of every week. During this meeting:

- 1. Review progress to date, including how actual progress for the previous week matched expected progress
- 2. Have the customers pick a week's worth of stories to implement this week.
- 3. Break the stories into tasks. Team members sign up for tasks and estimate them.

Start week by writing automated tests that will run when the stories are completed. Spend the rest completing the stories and getting the tests to pass. The goal is to have deployable software at the end of the week which everyone can celebrate as progress.

The nice thing about a week is that everyone —programmers, testers, and customers together — is focused on having the tests run on Friday. If you get to Wednesday and it is clear that all the tests won't be running, you still have time to choose the most valuable stories and complete them.

Daily deployment XP

Goes back to Microsoft's Daily Build

"China Shop rules": you break it, you fix it

Difficult to reconcile with other XP principles

Continuous integration XP

Source: Wake

Rather than weekly or daily builds, build system several times per day

Benefits:

- Integration is easier because little has changed
- Team learns more quickly
- Unexpected interactions rooted out early: conflicts are found while team can still change approach
- Problematic code more likely to be fixed because more eyes see it sooner
- Duplication easier to eliminate because visible sooner

Ten-minute build

XP

Source: Shore 08

Make sure that the build can be completed, through an automatic script, in ten minutes or less, to allow frequent integration. Includes:

- Compile source code
- Run tests
- Configure registry settings
- Initialize database schemas
- Set up web servers
- Launch processes
- Build installers
- Deploy

Make sure the build provides a clear indication of success or failure

If it has to take more than ten minutes, split the project into subprojects, and replace end-to-end funcational tests by unit tests

Slack

XP

Source: Beck, deMarco

"In any plan, include some minor tasks that can be dropped if you get behind."

Goals:

- Establishing trust in the team's ability to deliver
- Reduce waste

Single code base

XP

Maintain a single code base: avoid branching, even if permitted by configuration management system

Technical environment

Crystal

Access to automated tests, configuration management, frequent integration, code repository

Shared code

XP

Agile methods reject code ownership in favor of code whose responsibility is shared by entire team

Rationale:

- Most non-trivial features extend across many layers in the application
- Code ownership creates unnecessary dependencies between team members and delays
- What counts is implemented features, not personal responsibility
- Avoid blame game
- Avoid specialization
- Minimize risk (team members leaving)

Code and tests

XP

Maintain only code and tests as permanent artifacts

Pay-per-use

XP

Source: Beck 05

Charge for software by actual usage

Note: this was tried and failed in the 80s:
"Superdistribution" (Cox 1996)

More XP principles

Mutual benefit

Self-similarity

Improvement

Diversity

Reflection

Flow

Opportunity

Redundancy

Failure

Quality

Baby steps

Accepted responsibility

3 What is agile?

- 3.2 -

Agile roles

Product owner

Scrum

Source: Sutherland

The product owner:

- Defines product features
- Decides on release date
- Decides on release content
- Responsible for product profitability (ROI)
- Prioritizes features according to market value
- Can change features and priority over 30 days
- Accepts or rejects work results

ScrumMaster

Scrum

Source: Sutherland

The ScrumMaster:

- Ensures that the team is functional and productive
- Enables cooperation across all roles & functions
- Removes **impediments**
- Shields team from external interferences
- Enforces process: invites to daily scrum, sprint review, planning meetings

Team

Scrum

Source: Sutherland

The team:

- Is cross-functional
- is made of seven +/2 members
- Selects iteration goal
- Specifies work results
- Has right to do everything within boundaries of project guidelines to reach iteration goal
- Organizes itself and its work
- Demos work results to Product Owner

Manager

Scrum

Source: Sutherland

The managers:

- Support team in its use of Scrum
- Contribute wisdom, expertise and assistance
- Do not "play nanny":
  - ~~"Assign tasks, get status reports, and other forms of micromanagement"~~
- Instead, by "play guru":
  - Mentor, coach, play devil's advocate, help remove impediments, help problem-solve,
- May need to evolve their management style, e.g. use Socratic questioning to help team discover solution to a problem, (rather than imposing a solution to team)

Customer

XP

Source: Chromatic 03

Customer responsibilities in XP:

- Trust developers' technical decisions, because developers understand technology
- Analyze risk correctly, weighing stories against each other
- Provide precise stories, enabling developers to produce comprehensive task cards and accurate estimates
- Choose stories with maximum value, scheduling the most valuable stories that could possibly fit in to next iteration
- Work within team, providing guidance and receiving feedback as quickly and accurately as possible

Expert user

Crystal

Person with expert knowledge of the project area, who can answer questions and suggest solutions to problems

Should be actual user and not just a tester from the development team

Minimum of once a week, two-hour meeting with expert user, and ability to make phone calls

Developer

XP Scrum

Source: Chromatic 03

Main job: turn customer stories into working code.

Developer obligations:

- Know and understand technical issues
- Create and maintain the system as it evolves
- Answer: "*How will we implement it?*", "*How long will it take?*" & "*What are the risks?*"
- Work with customer to understand his stories
- From a story, decide implementation
- Estimate work for each story, based on implementation decisions & experience
- Identify features that depend on other features
- Identify risky features and report them to customer
- Follow team guidelines
- Implement only what is necessary
- Communicate constantly with customers

Developer Rights:

- Estimate own work
- Work sensible & predictable schedule, by scheduling only work that can be done
- Produce code that meets the customer's needs, by focusing on testing, refactoring, and customer communication
- Avoid need to make business decisions, by allowing the customer to make them

Tracker

XP Scrum

Source: Chromatic 03

Keeps track of the schedule

Most important metric

- Velocity: ratio of ideal time estimated for tasks to actual time spent implementing them.

Other important data:

- Changes in velocity
- Amount of overtime worked
- Ratio of passing to failing tests

These numbers measure progress and the rate of progress and help determine if the project is on schedule for the iteration

To measure velocity within the iteration, every day or two, the tracker asks each developer how many tasks he has completed

Coach

XP

After: Chromatic 03

Optional role:

- Guides team
- Mentors team
- Leads by example
- Teaches when necessary
- May teach by doing
- May offer ideas to solve thorny problems
- May serve as intermediary with management

In Scrum: this role is mostly taken on by the ScrumMaster

- 3.3 -

Agile artifacts

Use cases (scenarios)

One of the UML diagram types

A use case describes how to achieve a single business goal or task through the interactions between external actors and the system

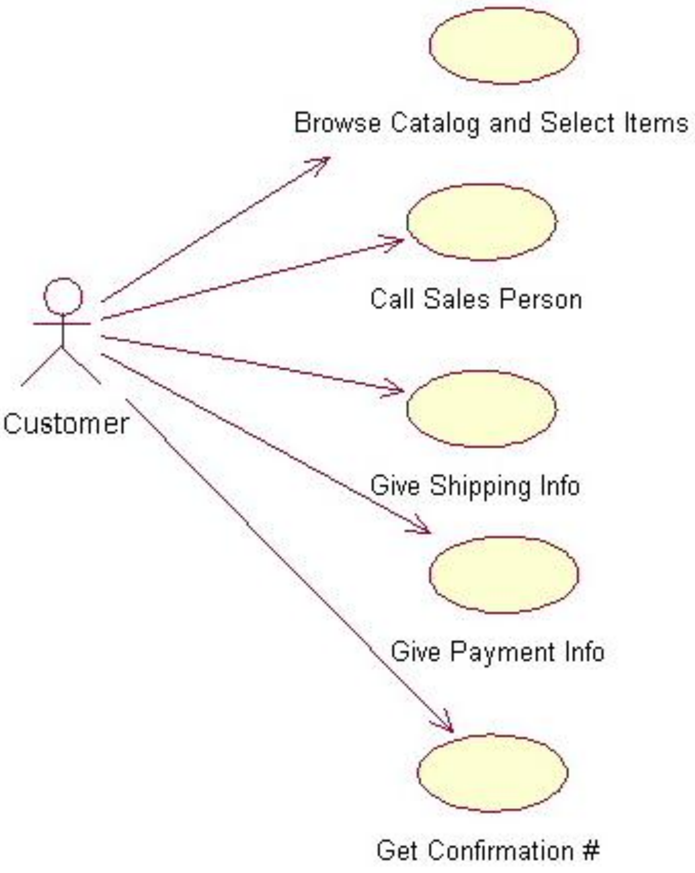
A good use case must:

- Describe a business task
- Not be implementation-specific
- Provide appropriate level of detail
- Be short enough to implement by one developer in one release

Use case example

Place an order:

- Browse catalog & select items
- Call sales representative
- Supply shipping information
- Supply payment information
- Receive conformation number from salesperson



May have precondition, postcondition, invariant

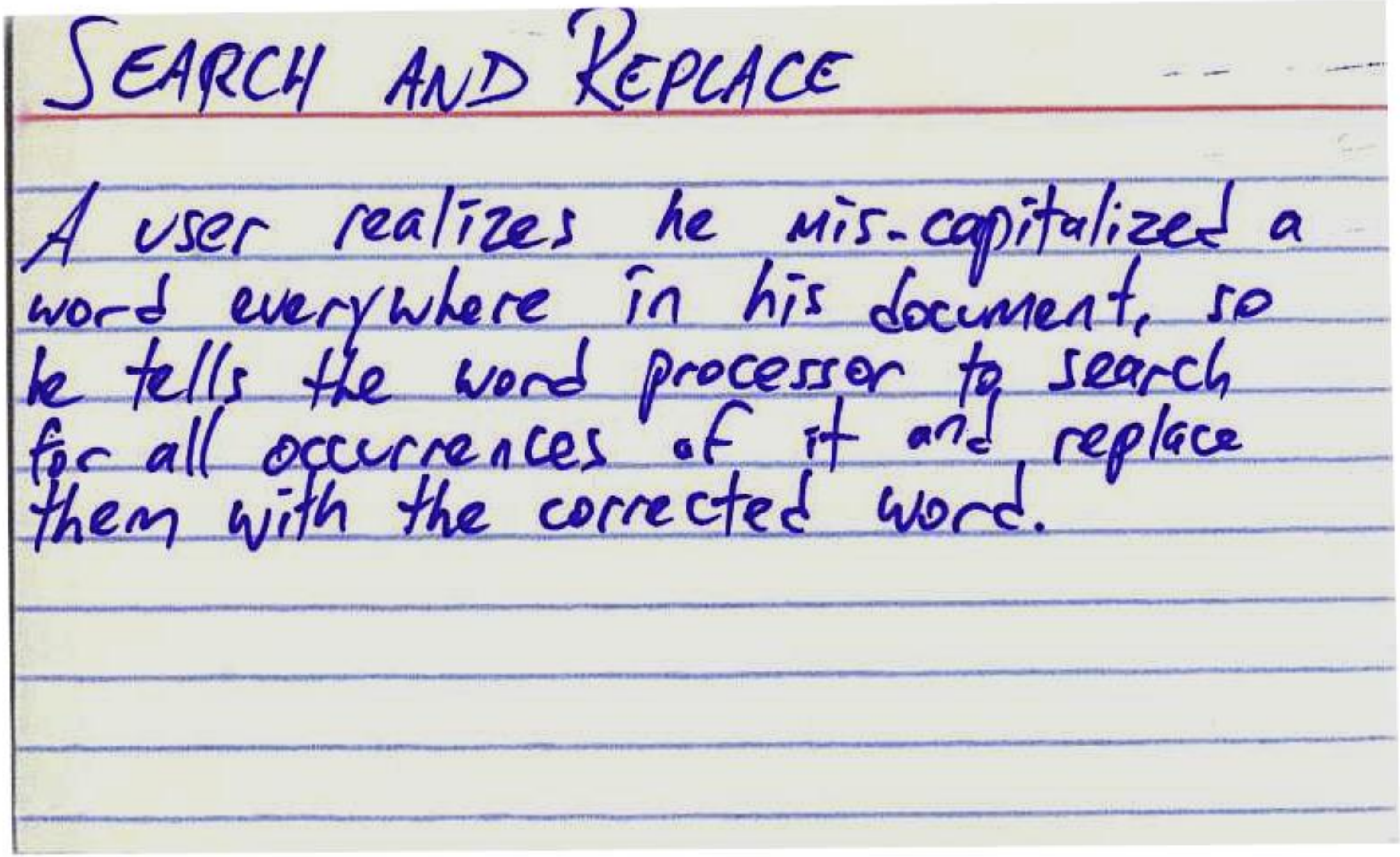
A use case

Source: Steelman & Greene

Name	**UC-8: Search and Replace**
Summary	All occurrences of a search term are replaced with replacement text.
Rationale	While editing a document, many users find that there is text somewhere in the file being edited that needs to be replaced, but searching for it manually by looking through the entire document is time-consuming and ineffective. The search-and-replace function allows the user to find it automatically and replace it with specified text. Sometimes this term is repeated in many places and needs to be replaced. At other times, only the first occurrence should be replaced. The user may also wish to simply find the location of that text without replacing it.
Users	All users
Preconditions	A document is loaded and being edited.
Basic Course of Events	1.The user indicates that the software is to perform a search-and-replace in the document. 2.The software responds by requesting the search term and the replacement text. 3.The user inputs the search term and replacement text and indicates that all occurrences are to be replaced. 4.The software replaces all occurrences of the search term with the replacement text.
Alternative Paths	1.In Step 3, the user indicates that only the first occurrence is to be replaced. In this case, the software finds the first occurrence of the search term in the document being edited and replaces it with the replacement text. The postcondition state is identical, except only the first occurrence is replaced, and the replacement text is highlighted. 2.In Step 3, the user indicates that the software is only to search and not replace, and does not specify replacement text. In this case, the software highlights the first occurrence of the search term and the use case ends. 3.The user may decide to abort the search-and-replace operation at any time during Steps 1, 2, or 3. In this case, the software returns to the precondition state.
Postconditions	All occurrences of the search term have been replaced with the replacement text.

A use case

Source: Steelman & Greene



SEARCH AND REPLACE

A user realizes he mis-capitalized a word everywhere in his document, so he tells the word processor to search for all occurrences of it and replace them with the corrected word.

User story

XP

Scrum

Source: Cohn

"A user story is simply something a user wants"

"Stories are more than just text written on an index card but for our purposes here, just think of user story as a bit of text saying something like

- Paginate the monthly sales report
- Change tax calculations on invoices.

Many teams have learned the benefits of writing user stories in the form of "As a … I … so that …"

Standard form for user stories Scrum

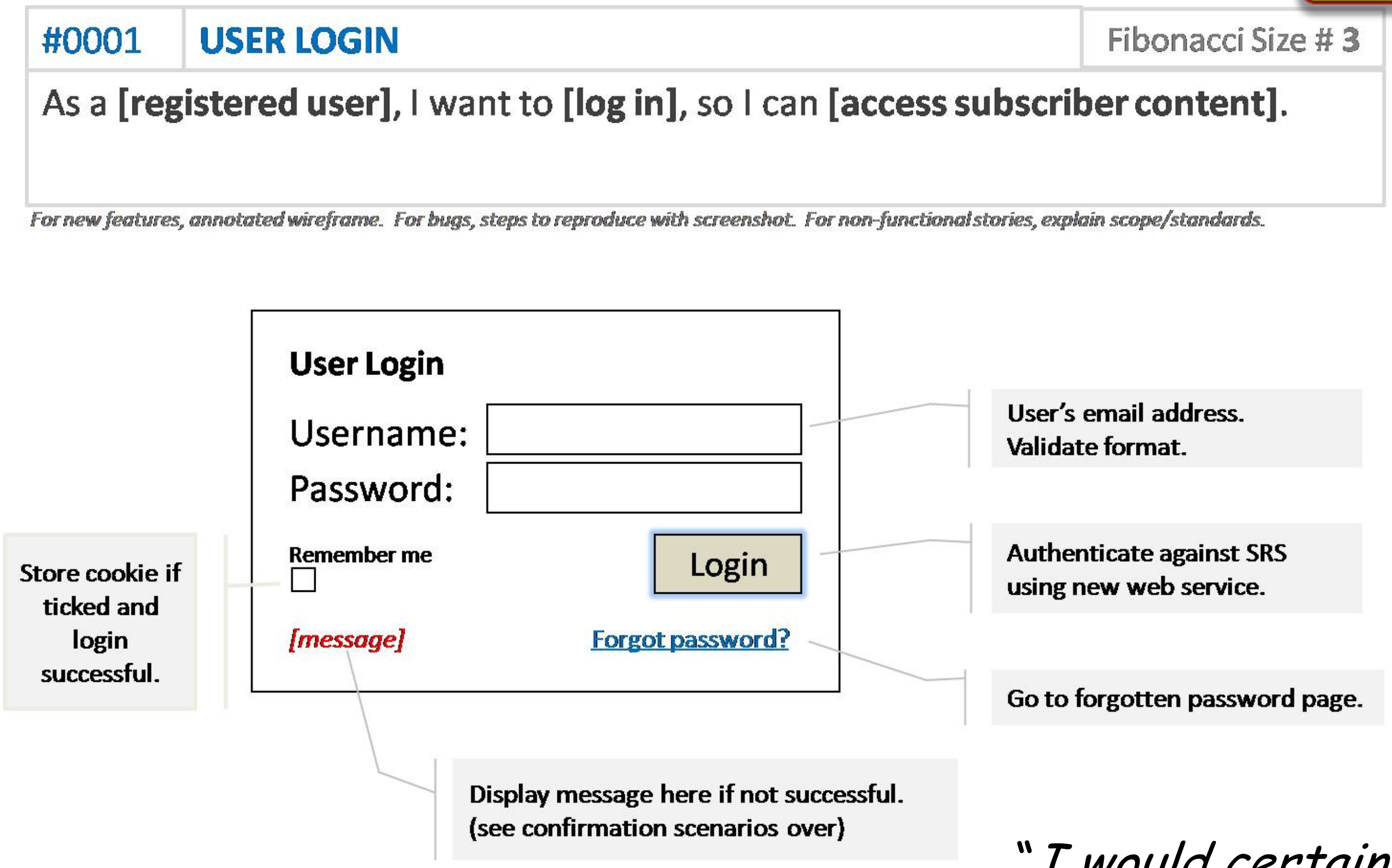
"As a <*user_or_role*>
I want <*business_functionality*>
so that <*business_justification*>"

Example:

"

Example user story

Source: Waters



#0001	USER LOGIN	Fibonacci Size # 3

As a **[registered user]**, I want to **[log in]**, so I can **[access subscriber content]**.

For new features, annotated wireframe. For bugs, steps to reproduce with screenshot. For non-functional stories, explain scope/standards.

User's email address. Validate format.

Authenticate against SRS using new web service.

Go to forgotten password page.

Store cookie if ticked and login successful.

Display message here if not successful. (see confirmation scenarios over)

Further information is attached to this story on VSTS Product Backlog.

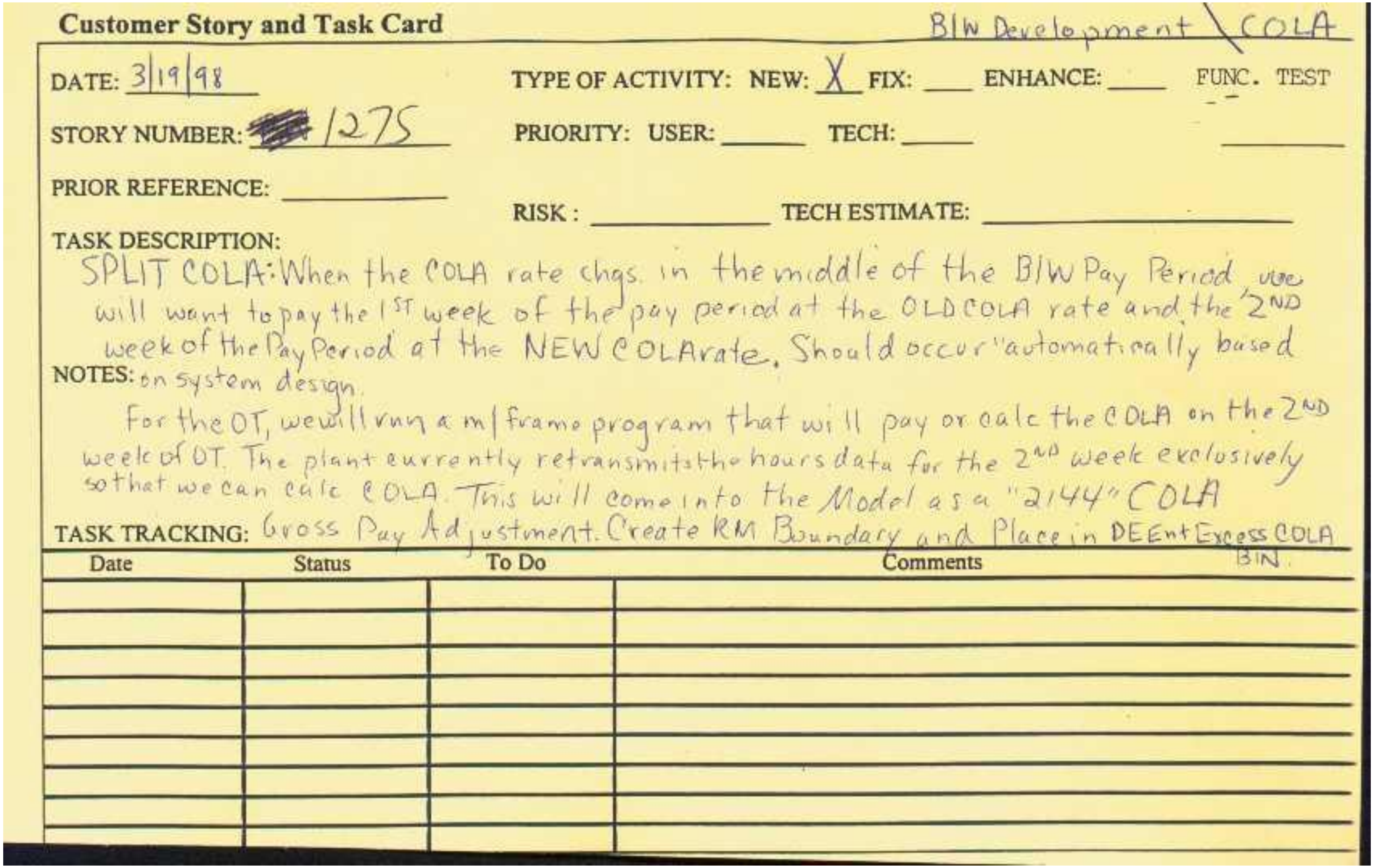
> *"I would certainly argue it is more easily digestible than a lengthy specification, especially for business colleagues"*

Story card

XP

Source: Jeffries

From the original C3 project:



Customer Story and Task Card BIW Development \ COLA

DATE: 3/19/98

TYPE OF ACTIVITY: NEW: X FIX: ___ ENHANCE: ___ FUNC. TEST ___

STORY NUMBER: [redacted] 1275

PRIORITY: USER: ___ TECH: ___

PRIOR REFERENCE: ___

RISK: ___ TECH ESTIMATE: ___

TASK DESCRIPTION:

SPLIT COLA: When the COLA rate chgs. in the middle of the BIW Pay Period we will want to pay the 1ST week of the pay period at the OLD COLA rate and the 2ND week of the Pay Period at the NEW COLA rate. Should occur "automatically based on system design.

NOTES:

For the OT, we will run a m/frame program that will pay or calc the COLA on the 2ND week of OT. The plant currently retransmits the hours data for the 2ND week exclusively so that we can calc COLA. This will come into the Model as a "2144" COLA

TASK TRACKING: Gross Pay Adjustment. Create RM Boundary and Place in DE Ent Excess COLA BIN.

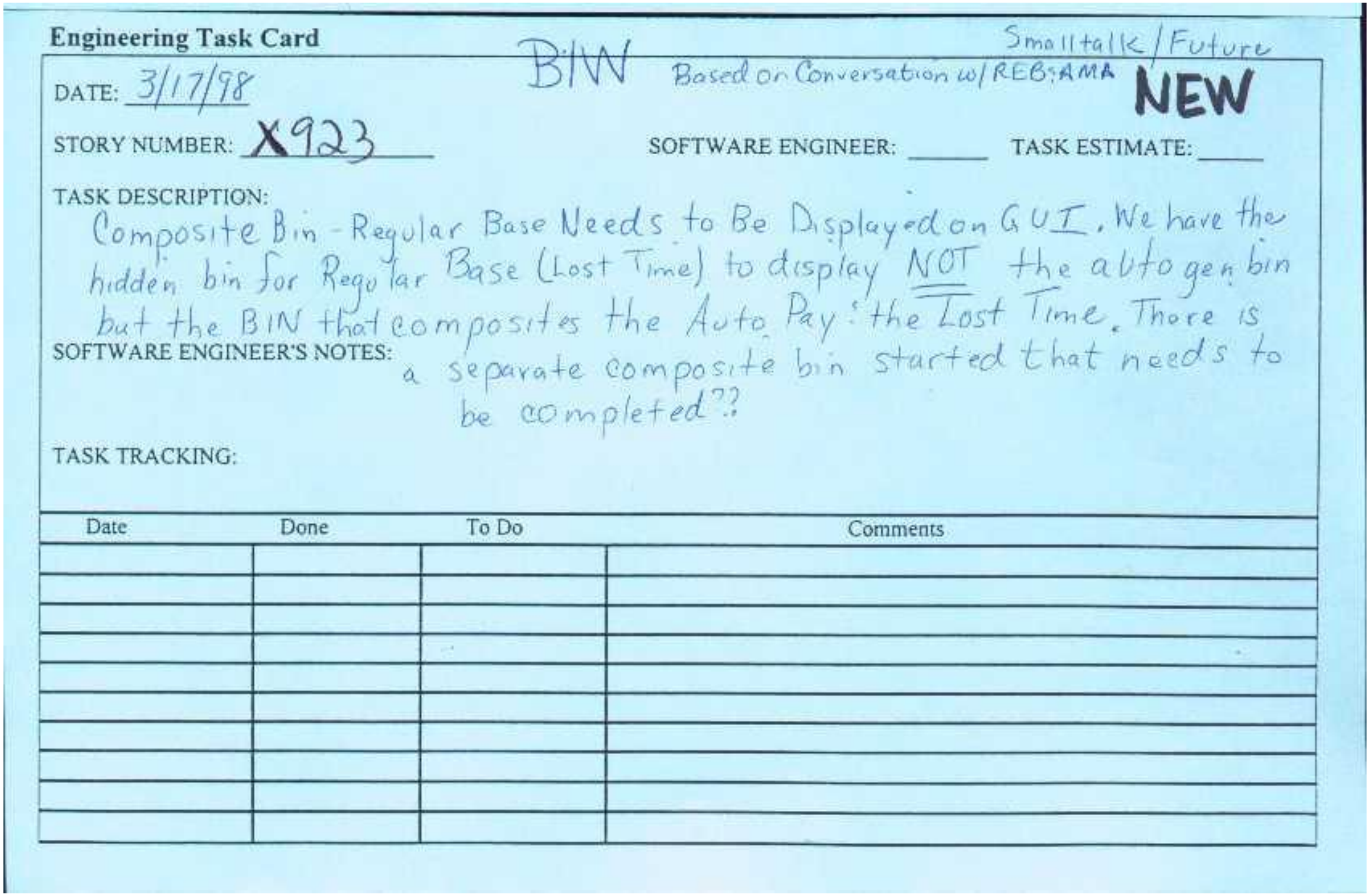
Date	Status	To Do	Comments

Task card

XP

Source: Jeffries

From the original C3 project:



Engineering Task Card

Smalltalk / Future

BIW

Based on Conversation w/ REB; AMA

NEW

DATE: 3/17/98

STORY NUMBER: X923

SOFTWARE ENGINEER: ______ TASK ESTIMATE: ______

TASK DESCRIPTION:

Composite Bin - Regular Base Needs to Be Displayed on GUI. We have the hidden bin for Regular Base (Lost Time) to display NOT the auto gen bin but the BIN that composites the Auto Pay & the Lost Time. There is a separate composite bin started that needs to be completed??

SOFTWARE ENGINEER'S NOTES:

TASK TRACKING:

Date	Done	To Do	Comments

Use cases vs user stories

Source: Steelman & Greene

Differences:

- User stories are about needs; use cases are about the behavior to be built into the software to meet those needs.
- User stories are easy for users to read; user cases describe a complete interaction between the software and users (and possibly other systems).

Alistair Cockburn:

More on the difference

Source: Cockburn

"*Think of a User Story as a Use Case at 2 bits of precision*":

- A user story is very simple and is written by the customer. It is incomplete, possibly inaccurate, and does not handle exceptional cases because not a lot of effort is expended making sure it is correct. It serves as a starting point for additional discussions with the customer about the full extent of his needs.
- A use case is more complex and is written by the developer in cooperation with the customer. It attempts to be complete, accurate, and handle all possible cases. A lot of effort it expended to make sure it is correct. It is intended to answer any developer questions about customer requirements so that developers may proceed without having to track down the customer.

User stories

X	f (x)
0	0
1	1
2	4
3	9
4	16
…	…

Product backlog Scrum

High-level list maintained throughout project

- Aggregates backlog items: broad descriptions of all potential features, prioritized as an absolute ordering by business value
- Open and editable by anyone
- Contains rough estimates of both business value and development effort
- Property of the product owner
- Associated development effort set by the Team

The **task board** is used to see and change the state of the tasks of the current sprint, like “to do”, “in progress” and “done”.

Task board

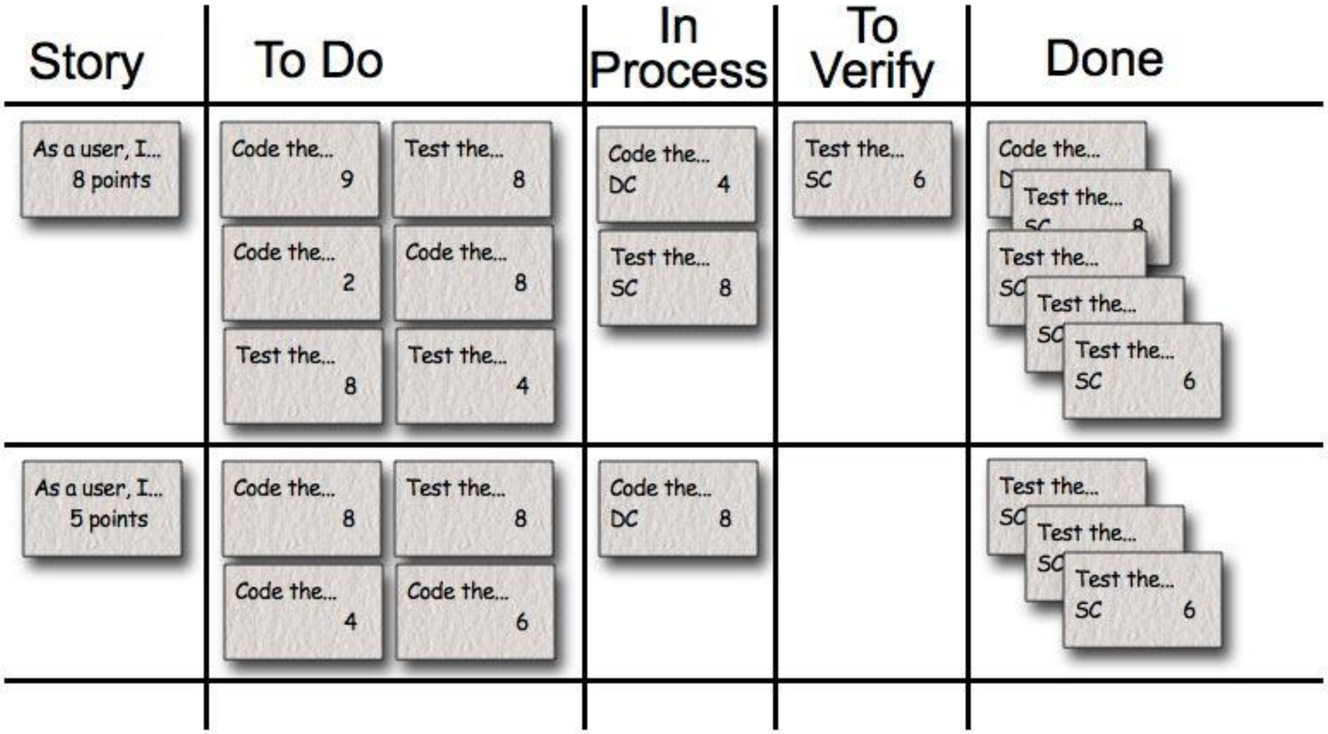
Scrum

Source: Cohn, Anand

Used to see and change the state of the tasks of the current sprint: "to do", "in progress", "done".

Benefits:

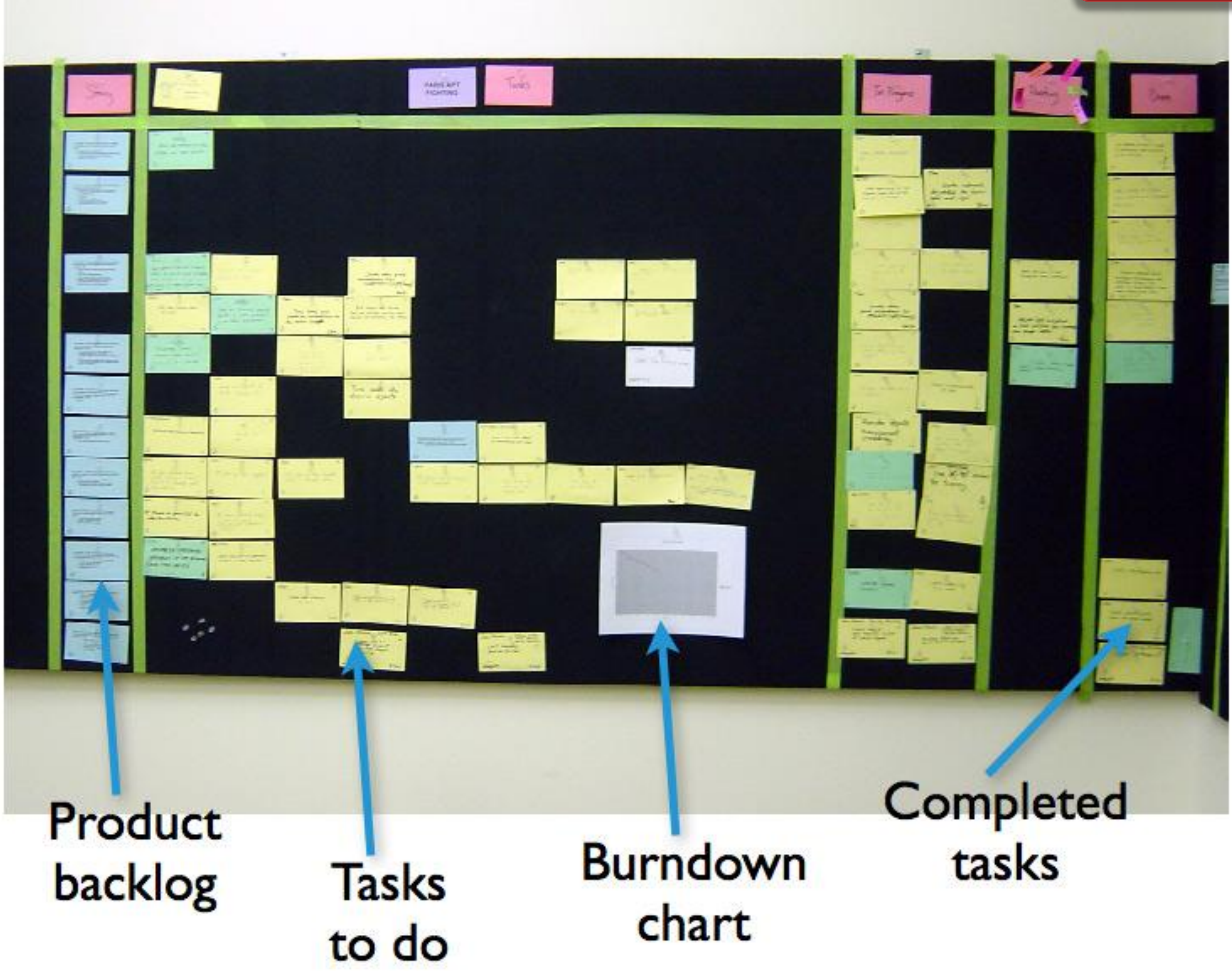
- Transparency
- Collaboration
- Prioritization
- Focus
- Self-organization
- Empiricism.
- Humility



Story	To Do	In Process	To Verify	Done
As a user, I... 8 points	Code the... 9 Test the... 8 Code the... 2 Code the... 8 Test the... 8 Test the... 4	Code the... DC 4 Test the... SC 8	Test the... SC 6	Code the... D Test the... SC 8 Test the... SC Test the... SC Test the... SC 6
As a user, I... 5 points	Code the... 8 Test the... 8 Code the... 4 Code the... 6	Code the... DC 8		Test the... SC Test the... SC Test the... SC 6

Story board

Source: Cohn



Product backlog

Tasks to do

Burndown chart

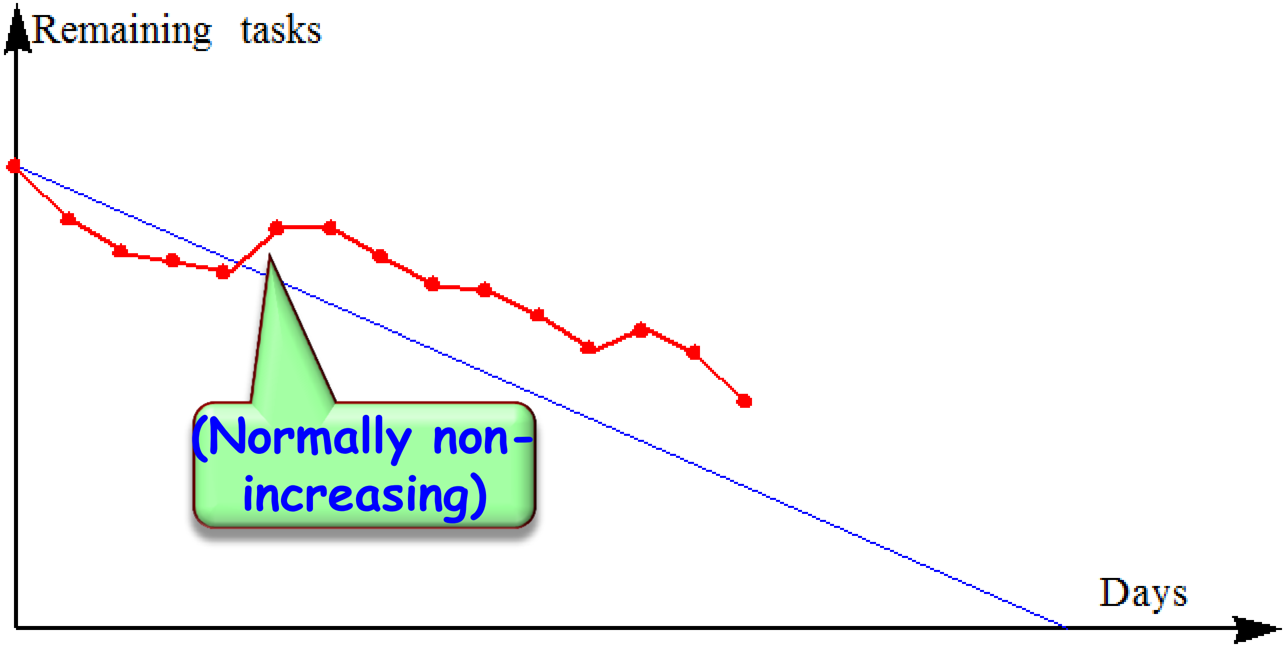
Completed tasks

Burndown chart Scrum

Publicly displayed chart, updated every day, showing

- Remaining work
- Progress

in the Sprint backlog



(Normally non-increasing)

Bullpen

Single, open room

(See “ Informative workspace” principle)

3 What is agile?

- 3.4 -

Agile practices

Onsite customer

XP

Seen under Principles

Pair programming

XP

Source: Beck 2005

Two programmers sitting at one machine

Dialog between two people, with shared keyboard & mouse

Goals:

- Keep each other on task
- Brainstorm refinements to system
- Clarify ideas
- Take initiative when other stuck, lowering frustration
- Hold each other accountable to team practices

“Avoid strong colognes” and “cover your mouth when you cough”, “avoid sexual arousal”

Refactoring

XP

Source: Fowler

"Disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior"

Example techniques:

Techniques that allow for more abstraction

- Encapsulate Field
- Replace conditional with polymorphism
- Extract Method
- Extract Class
- Move Method or Field
- Rename Method or Field
- Pull Up
- Push Down

Used in agile methods as a substitute for upfront design

Process review

Crystal

Scrum

Crystal: **Reflection workshop** held every two weeks to

- Identify processes that are and are not working well
- Help team to modify them to develop a satisfactory strategy

Scrum: **Sprint review** after a sprint

Source: Cohn

- During the sprint review the project is assessed against the sprint goal determined during the Sprint planning meeting
- Ideally the team has completed each product backlog item brought into the sprint, but it is more important that they achieve the overall goal of the sprint

Test-Driven Development XP

Standard cycle:

- Add a test
- Run all tests and see if the new one fails
- Write some code
- Run the automated tests and see them succeed
- Refactor code

Expected benefits:

- Catch bugs early
- Write more tests
- Drive the design of the program
- Replace specifications by tests

- Use debugger less
- More modular code
- Better coverage
- Improve overall productivity

Planning game XP

Meeting that occurs once per iteration

Purpose: guide the product into delivery

Instead of predicting exact delivery dates, planning game seeks to guide the project towards delivery

Two parts:

- Release Planning (with customers):
- Iteration Planning (developers only)

Each has three phases:

- Exploration Phase
- Commitment Phase
- Steering Phase:

Planning poker Scrum

- Present individual stories for estimation
- Discuss
- Each participant chooses from his deck the numbered card that represents estimate of work involved in story under discussion
- Deck has successive numbers (quasi-Fibonacci)
- Keep estimates private until each participant has chosen a card
- Reveal estimates
- Repeat until consensus

(Variant of Wideband Delphi technique.)

Whole team XP Scrum

Source: Jeffries

All contributors sit together as members of one team:

- Must include a business representative who provides the requirements, sets the priorities, and steers the project.
- Includes programmers
- May include testers,
- May include analysts, as helpers to the Customer, helping to define the requirements
- Often includes a coach
- May include a manager

None of these roles is necessarily the exclusive property of just one individual: Everyone on an XP team contributes in any way that they can. The best teams have no specialists, only general contributors with special skills.

Osmotic communication

Crystal

Team is together in a room and listen to each other

Information to flow around it

Developer must break concentration

Information flows quickly throughout the team

Questions answered rapidly

All team updated on what is happening

Reduce need for email and other non-direct communication

Facilitate taking over of others' tasks

Continuous integration

XP

The combination of frequent releases with relentless testing

Keep system fully integrated at all times

Small releases XP

Source: Jeffries

XP teams practice small releases in two important ways:

- Release running, tested software, delivering business value chosen by the Customer, every iteration. The Customer can use this software for any purpose, whether evaluation or even release to end users (highly recommended).

- Release to end users frequently as well. Web projects release as often as daily, in house projects monthly or more frequently. Even shrink-wrapped products are shipped as often as quarterly.

Coding standards XP

Project members all code to the same conventions

Collective code ownership XP

See “shared code” principle

Simple design XP

Produce the simplest design that works

Refactor as needed

Daily meeting Scrum

Goal: to set the day's work

Held every morning

Time-limited, usually 15 minutes

Involves all team members, with special role for those who are "committed" (over those just "involved")

Enables every team member to answer three questions:

- What did you do yesterday?
- What will you do today?
- Are there any impediments in your way?

Focus on commitments and on uncovering impediments (responsibility of the Scrum Master)

The resolution will take place outside of the meeting

Scrum of scrums Scrum

Each day normally after the daily scrum. These meetings allow clusters of teams to discuss their work, focusing especially on areas of overlap and integration

A designated person from each team attends

The agenda will be the same as the Daily Scrum, plus the following four questions:

- What has your team done since we last met?
- What will your team do before we meet again?
- Is anything slowing your team down or getting in their way?
- Are you about to put something in another team's way?

Planning meeting Scrum

Source: Wikipedia

At the beginning of the sprint cycle (every 7–30 days), a "Sprint Planning Meeting" is held. Select what work is to be done

Prepare the Sprint Backlog that details the time it will take to do that work, with the entire team

Identify and communicate how much of the work is likely to be done during the current sprint

Eight hour time limit

- (1st four hours) Product Owner + Team: dialog for prioritizing the Product Backlog
- (2nd four hours) Team only: hashing out a plan for the Sprint, resulting in the Sprint Backlog

At the end of a sprint cycle, two meetings are held: the "Sprint Review Meeting" and the "Sprint Retrospective"

Review meeting

Scrum

Review the work that was completed and not completed

Present the completed work to the stakeholders (a.k.a. "the demo")

Incomplete work cannot be demonstrated

Four hour time limit

Retrospective Scrum

All team members reflect on the past sprint

Make continuous process improvements

Two main questions are asked in the sprint retrospective:

- What went well during the sprint?
- What could be improved in the next sprint?

Three hour time limit

3 What is agile?

- 3.5 -

Agile methods

Scrum

Scrum

Developed in 1995 by Sutherland and Schwaber for software, based on ideas about developing commercial processes described byTakeuchi and Nonaka in an 1986 article

Emphasizes management rather than specific software techniques

Has been used in conjunction with CMMI

Lean software

Lean

Mary Poppendieck, late nineties

Inspired by techniques developed for production (starting with Deming)

7 key principles:

- Eliminate waste
- Amplify learning
- Decide as late as possible
- Deliver as fast as possible
- Empower the team
- Build integrity in
- See the whole

Scrum basics

Scrum

- Iterative, incremental process
- Emphasis on working product, fully tested and shippable
- Cross-functional team
- Basic work cycle: **sprint**
  - Typically 1-4 weeks in length, fixed duration, ending on specified date (even if work not complete)
- Prioritized list of requirements
  - At Sprint beginning, team selects from list and commits to completing them by end of Sprint
- Each work day: daily stand up meeting
  - Report to rest of team on progress
  - update visual representations of work remaining
- End of Sprint:
  - Team demonstrates what it has built
  - Gets feedback for next Sprint

Scrum terminology Scrum

Source: Sutherland

Impediment: Anything that prevents a team member from performing work as efficiently as possible

Sprint: Period, typically 2–4 weeks, in which development occurs on a set of backlog items that the Team has committed to

Definition of Done (DoD): exit criteria to determine whether a product backlog item is complete. Each team has its own DoD.

Abnormal Termination: Sprint cancellation by Product Owner

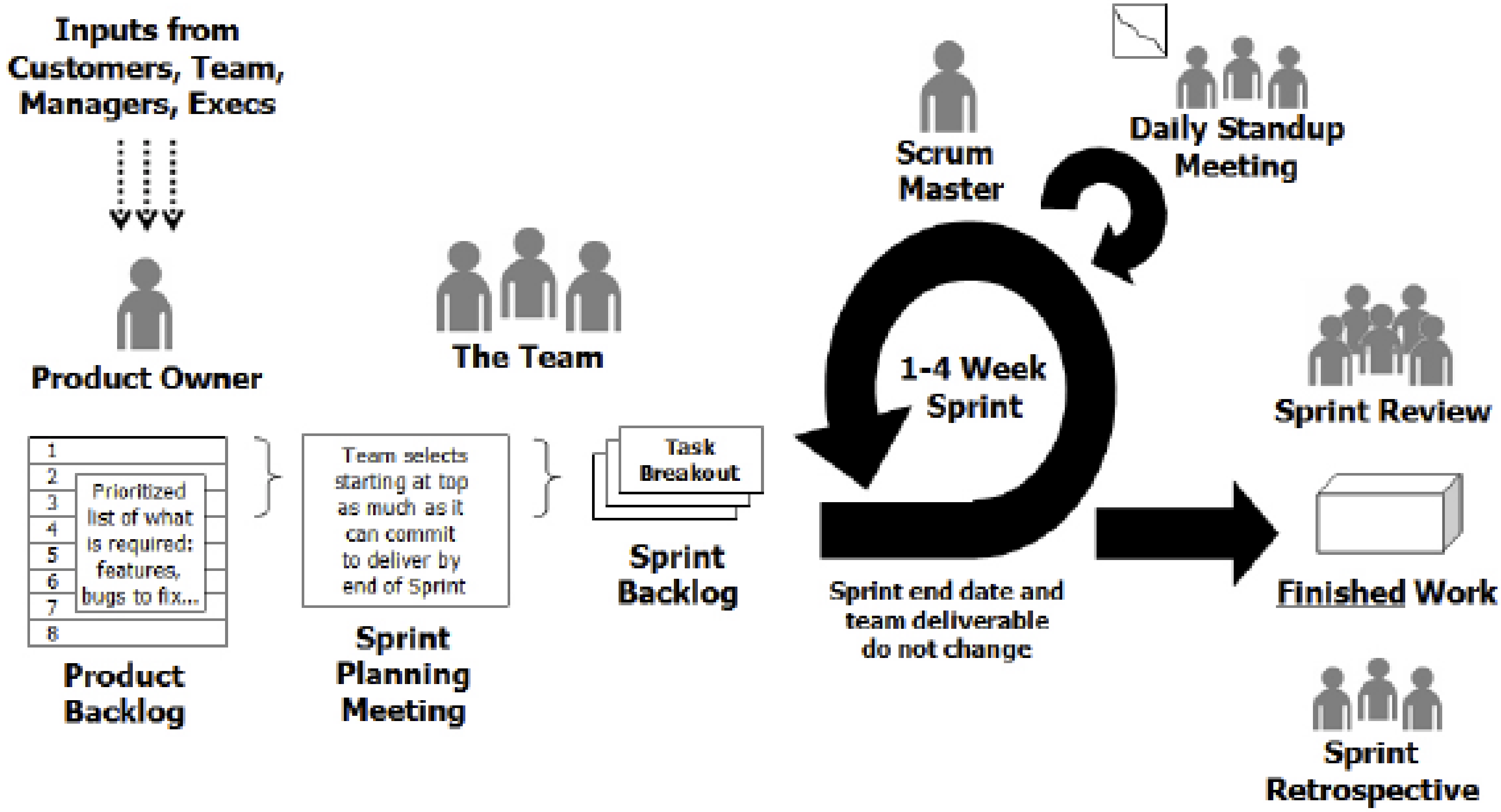
Planning Poker (studied earlier)

Point Scale: an abstract point system, used to discuss the difficulty of the story, without assigning actual hours. The most common scale used is a rounded Fibonacci sequence (1,2,3,5,8,13,20,40,100); also Clothes size (XS, S, M, L, XL)

Tasks: Added to the story at the beginning of a sprint and broken down into hours. Each task should not exceed 12 hours but it's common for teams to insist that a task take no more than a day to finish.

Overall Scrum process Scrum

Source: Sutherland


Inputs from Customers, Team, Managers, Execs
Product Owner
The Team
Scrum Master
Daily Standup Meeting
1
2
3
4
5
6
7
8
Prioritized list of what is required: features, bugs to fix...
Product Backlog
Team selects starting at top as much as it can commit to deliver by end of Sprint
Sprint Planning Meeting
Task Breakout
Sprint Backlog
1-4 Week Sprint
Sprint end date and team deliverable do not change
Sprint Review
Finished Work
Sprint Retrospective

Extreme Programming (XP) XP

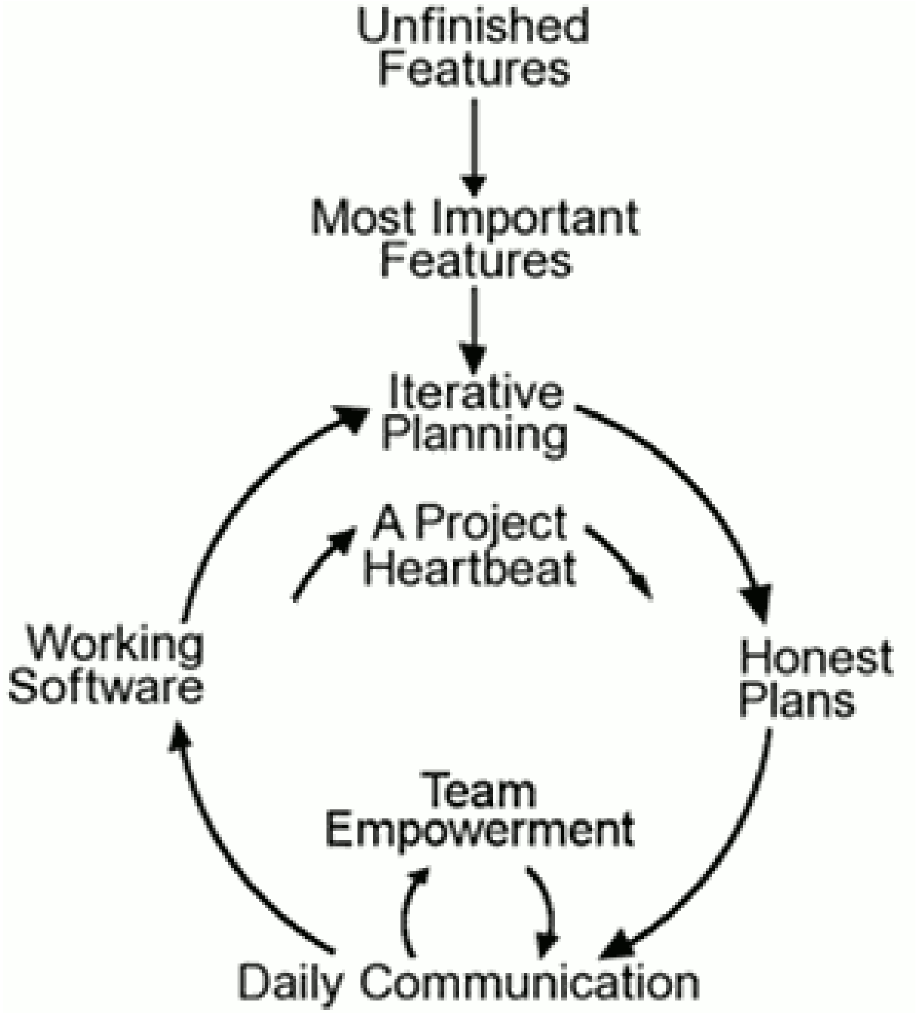
Created by Kent Beck during work on Chrysler Comprehensive Compensation System (C3) payroll project, written in Smalltalk

Actual outcome of project is highly controversial

XP is (Beck) a "software development discipline that organizes people to produce higher quality software more productively"

XP process XP

Source: Wells



XP criteria XP

Source: Wells

- Need to mitigate risk and produce working system
- Small team (2 to 12)
- Extended team, including manager and customer, "all working elbow to elbow"
- Testability: must be able to create and run automated unit and functional tests
- Timely delivery more important than productivity
- Produce the simplest design that works
- Refactor

Crystal

Created by Alistair Cockburn in mid-90s

Focused on:

- People
- Interaction
- Community
- Skills
- Talents
- Communications

Short description:

- "The lead designer and two to seven other developers … in a large room or adjacent rooms, … using such as whiteboards and flip charts, … having easy access to expert users, … distractions kept away, deliver running, tested, usable code to the users … every month or two (quarterly at worst), … reflecting and adjusting their working conventions periodically"

Crystal principles

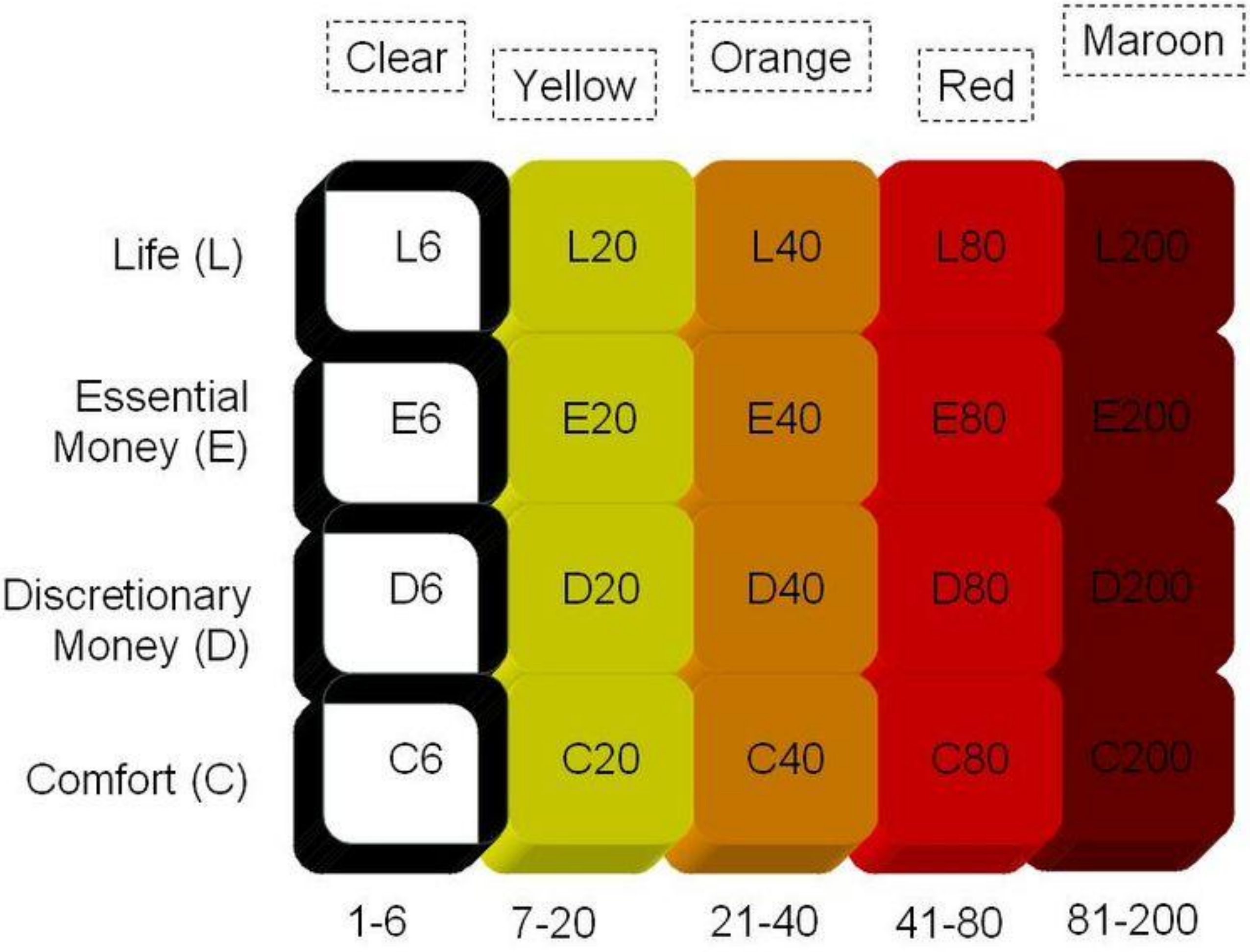
Crystal

- Frequent Delivery
- Reflective Improvement
- Osmotic Communication
- Personal Safety
- Focus
- Easy Access to Expert Users
- A Technical Environment with Automated Tests, Configuration Management, and Frequent Integration

Crystal family

Crystal

Source: Cockburn



	Clear	Yellow	Orange	Red	Maroon
Life (L)	L6	L20	L40	L80	L200
Essential Money (E)	E6	E20	E40	E80	E200
Discretionary Money (D)	D6	D20	D40	D80	D200
Comfort (C)	C6	C20	C40	C80	C200
	1-6	7-20	21-40	41-80	81-200

Example of deliverables Crystal

Crystal orange:

- Requirements Document
- Release Sequence (Schedule)
- Project Schedule
- Status Reports
- UI Design Document (if project has a UI)
- Object Model
- User Manual
- Test Cases

- 4.1 -

Conceptual analysis

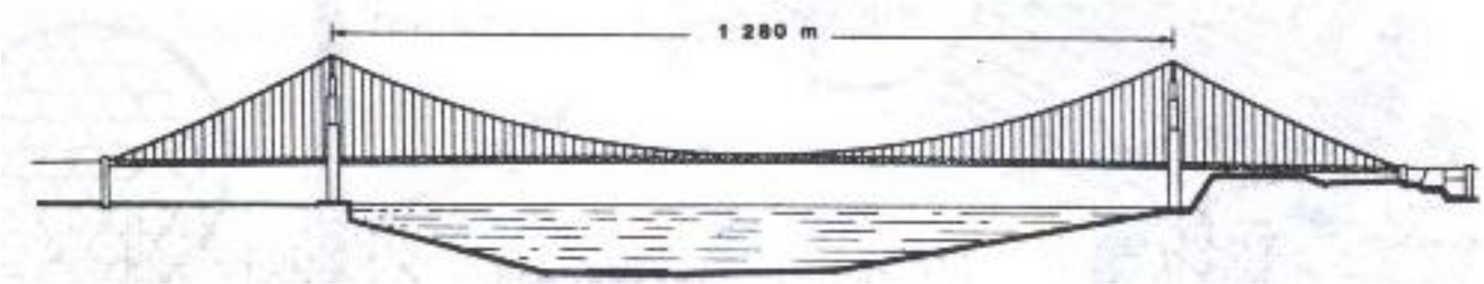
To what extent should we accept analogies?

Agile methods make considerable use of comparisons with engineering disciplines other than software

On the other hand, would you build a house using Scrum?

Description and implementation

A bridge



A drawing of a bridge

Description-Implementation Porosity

CECI N'EST PAS UNE PIPE
Magritte

A program text

```
private static boolean endsWith(String str, String suffix,
boolean ignoreCase) {
        if (str == null || suffix == null) {
             return (str == null && suffix == null);
        }
        if (suffix.length() > str.length()) {
             return false;
        }
        int strOffset = str.length() - suffix.length();
        return str.regionMatches(ignoreCase, strOffset);
  }
```

A ~~program text~~ specification (VDM)

```
AccNum = token;
CustNum = token;
Balance = int;
Overdraft = nat;
AccData :: owner : CustNum
          balance : Balance
state Bank of
          accountMap : map AccNum to AccData
          overdraftMap : map CustNum to Overdraft
inv mk_Bank(accountMap,overdraftMap) ==
          for all a in set rng accountMap & a.owner in set
                    dom overdraftMap and
                    a.balance >= -overdraftMap(a.owner)
```

Related idea: single-product principle in Eiffel

Single-Model Principle

All the information about a software system should be in the software text

Supported in EiffelStudio by Diagram Tool, multiple views of a class (contract, interface, inheritance…) & other techniques

Use cases and user stories

Use cases and user stories are only examples

The role of a requirements elicitation process is to go from individual examples to actual abstractions

My view

Use cases and user stories help requirement elicitation but not a fundamental requirement technique. They cannot define the requirements:

- Not abstract enough
- Too specific
- Describe current processes
- Do not support evolution

Use cases are to requirements what tests are to software specification and design

Major application: for validating requirements

User stories

x	f (x)
0	0
1	1
2	4
3	9
4	16
...	...

Use cases as requirements

What use cases (and user stories) are good for

They are ways to **validate** the user requirements

Use cases are to requirements (specifications) what tests are to programs

The task of requirements is to **abstract** from user stories

Test-Driven Development

The basic idea is sound…

… but not the replacement of specifications by test

Major benefit: keep an up-to-date collection of regression tests

Requirement that all tests pass can be unrealistic (tests degrade, a non-passing test can be a problem with the test and not with the software)

Basic TDD idea can be applied with specifications! See Contract-Driven Development

An alternative to waterfall, spiral etc.

The cluster model

Applied in the Eiffel context since 1990

Seamless, incremental development

Seamless development:

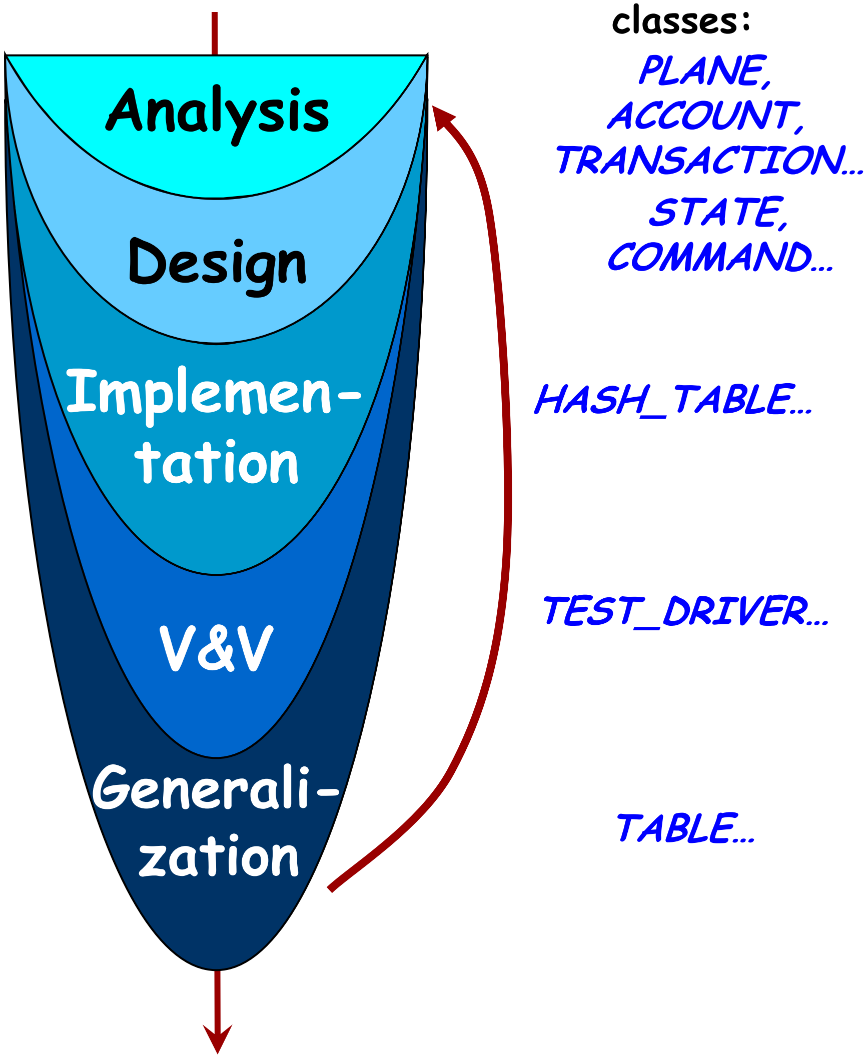
- Single set of notation, tools, concepts, principles throughout
- Continuous, incremental development
- Keep model, implementation and documentation consistent

Reversibility: can go back and forth

These are in particular some of the ideas behind the Eiffel method

Seamless development

- Single notation, tools, concepts, principles
- Continuous, incremental development
- Keep model, implementation and documentation consistent
- Reversibility: go back and forth

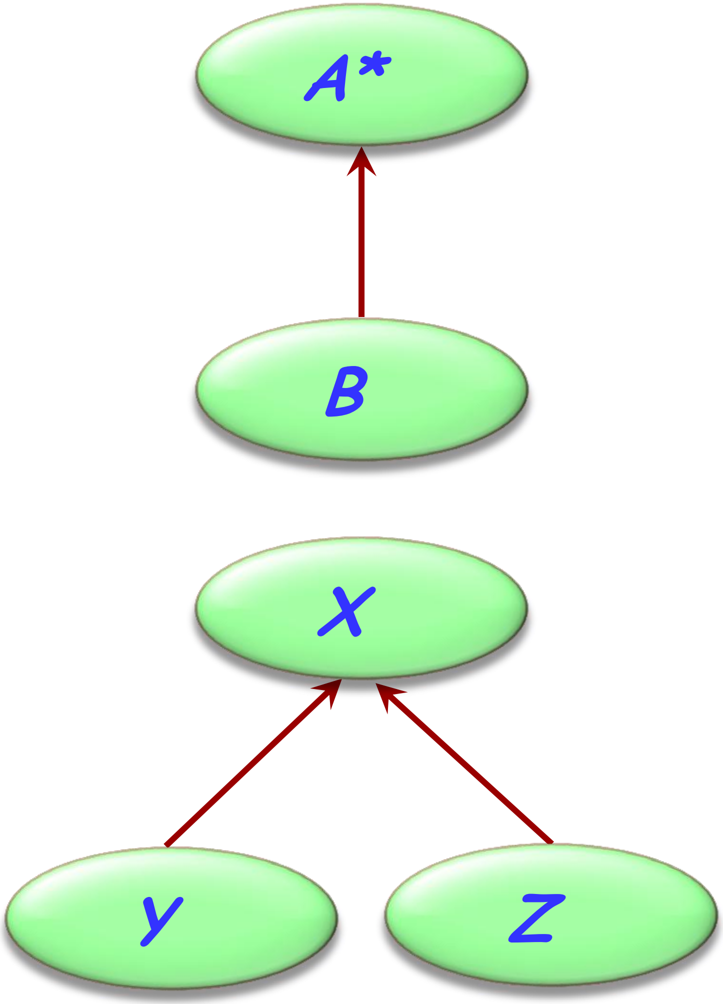


Generalization

Prepare for reuse. For example:

- Remove built-in limits
- Remove dependencies on specifics of project
- Improve documentation, contracts...
- Abstract
- Extract commonalities and revamp inheritance hierarchy

Few companies have the guts to provide the budget for this

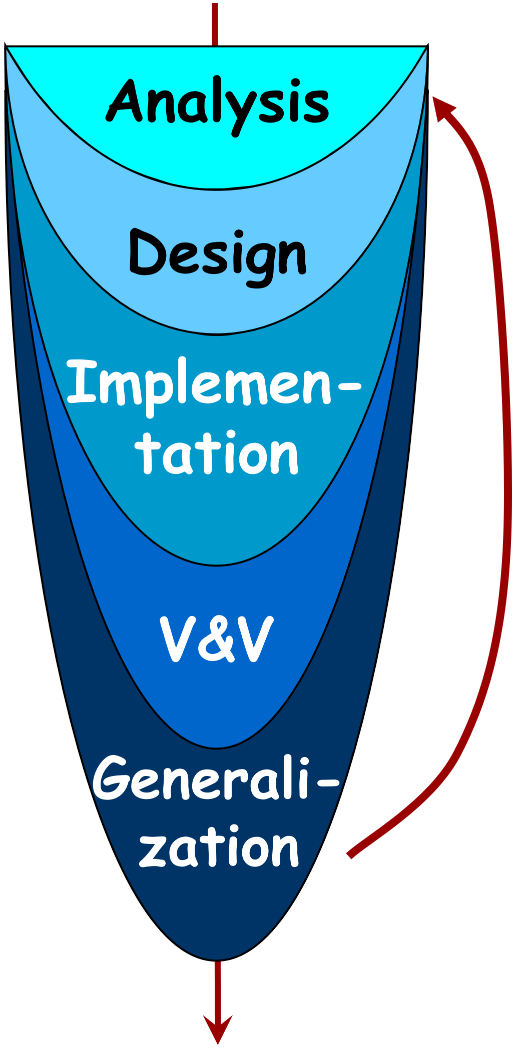


Finishing a design

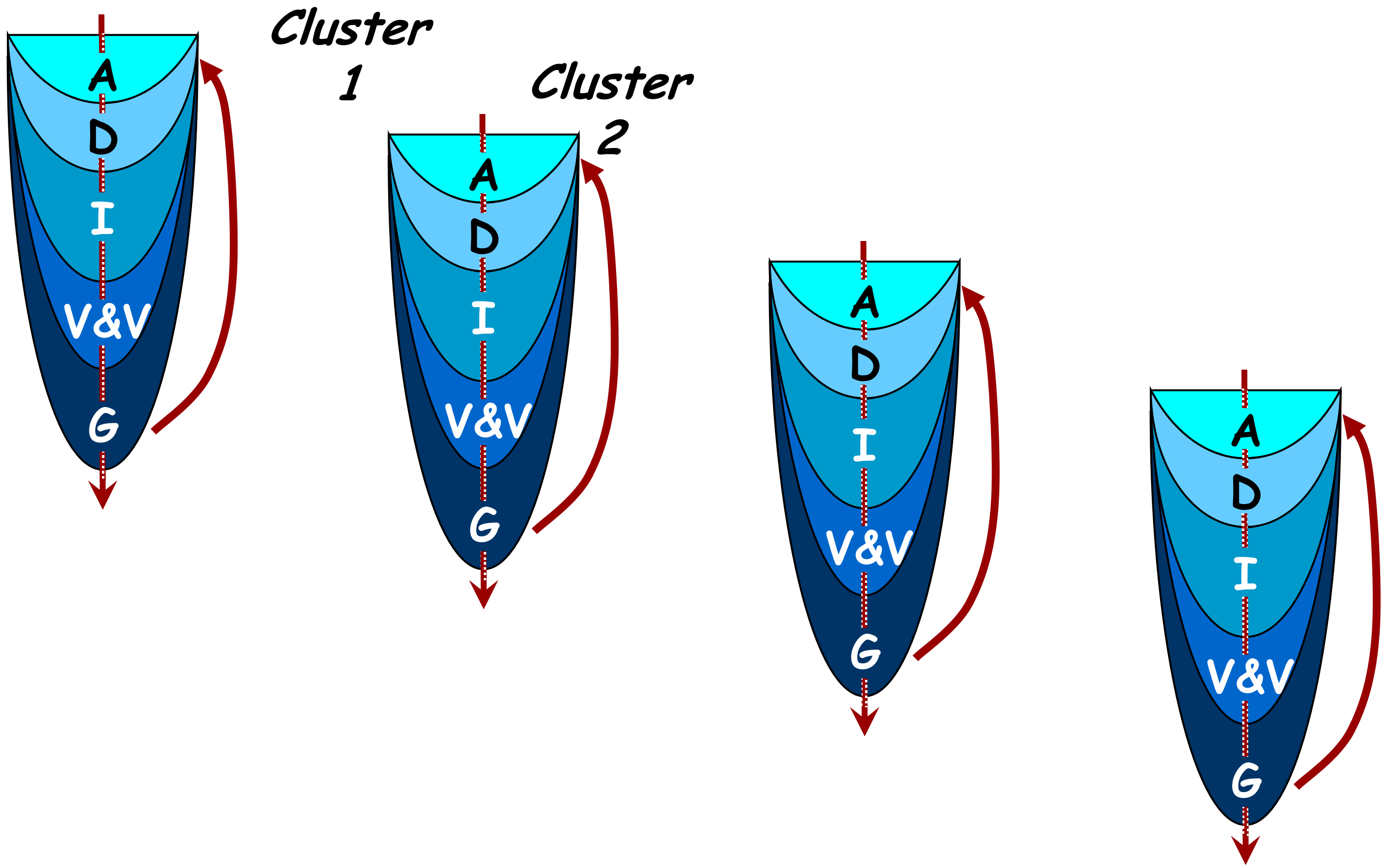
It seems that the sole purpose of the work of engineers, designers, and calculators is to polish and smooth out, lighten this seam, balance that wing until it is no longer noticed, until it is no longer a wing attached to a fuselage, but a form fully unfolded, finally freed from the ore, a sort of mysteriously joined whole, and of the same quality as that of a poem. It seems that perfection is reached, not when there is nothing more to add, but when there is no longer anything to remove.

(Antoine de Saint-Exupéry,
Terre des Hommes, 1937)

Reversibility

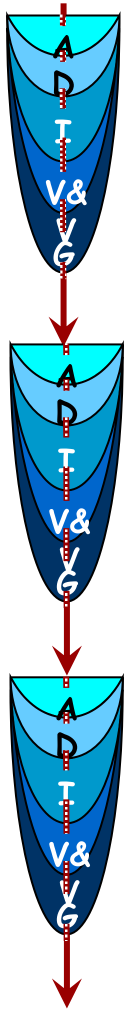


The cluster model



Extremes

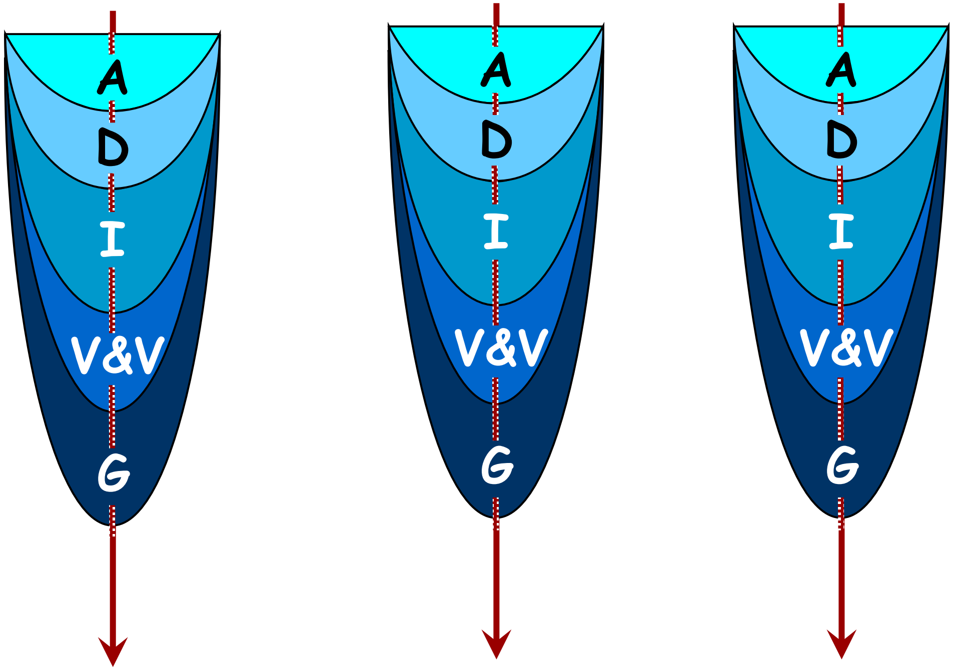
"Trickle"



Cluster 1

Cluster 2

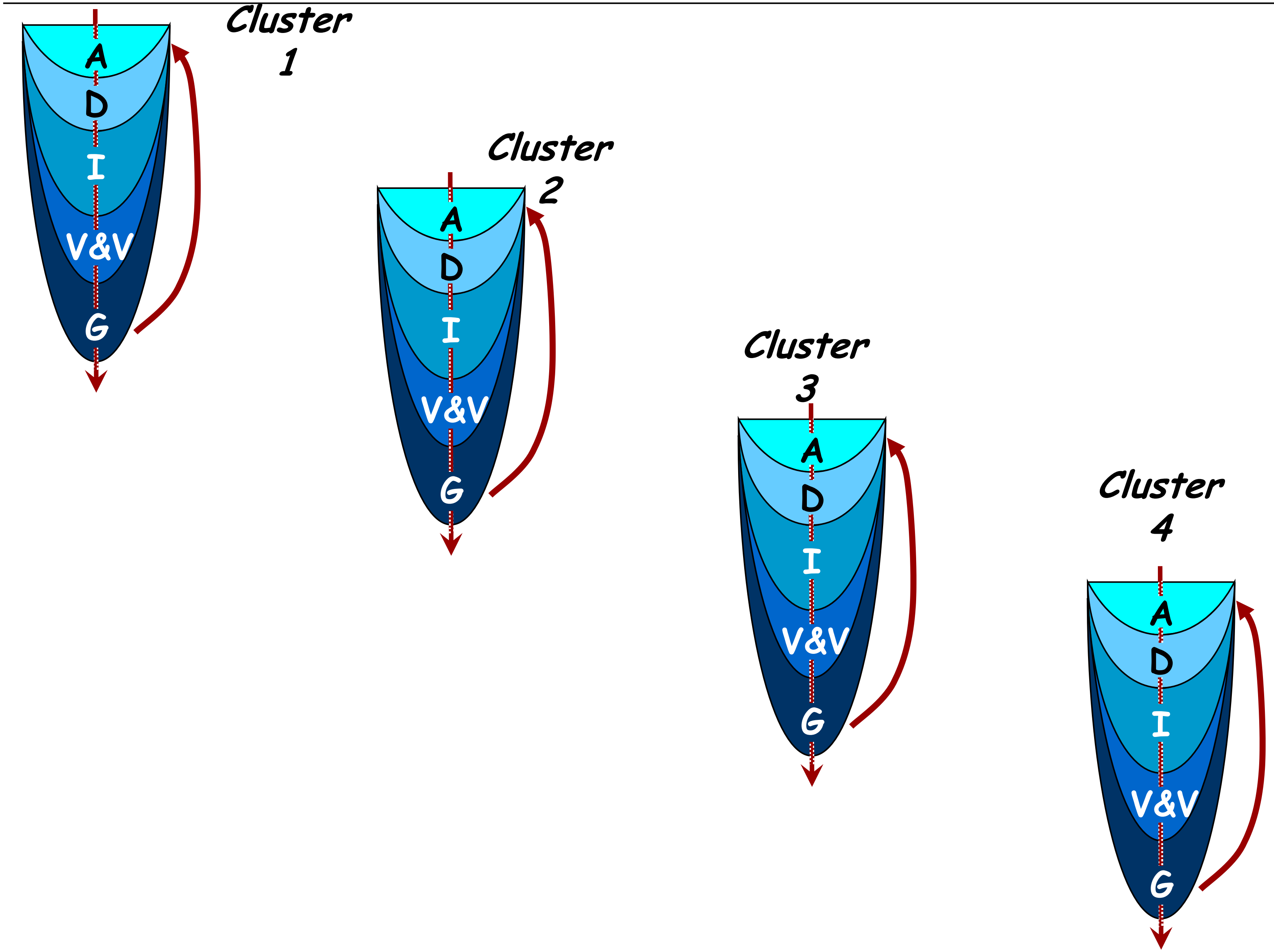
"Clusterfall"



Cluster 1

Cluster 2

Dynamic rearrangement



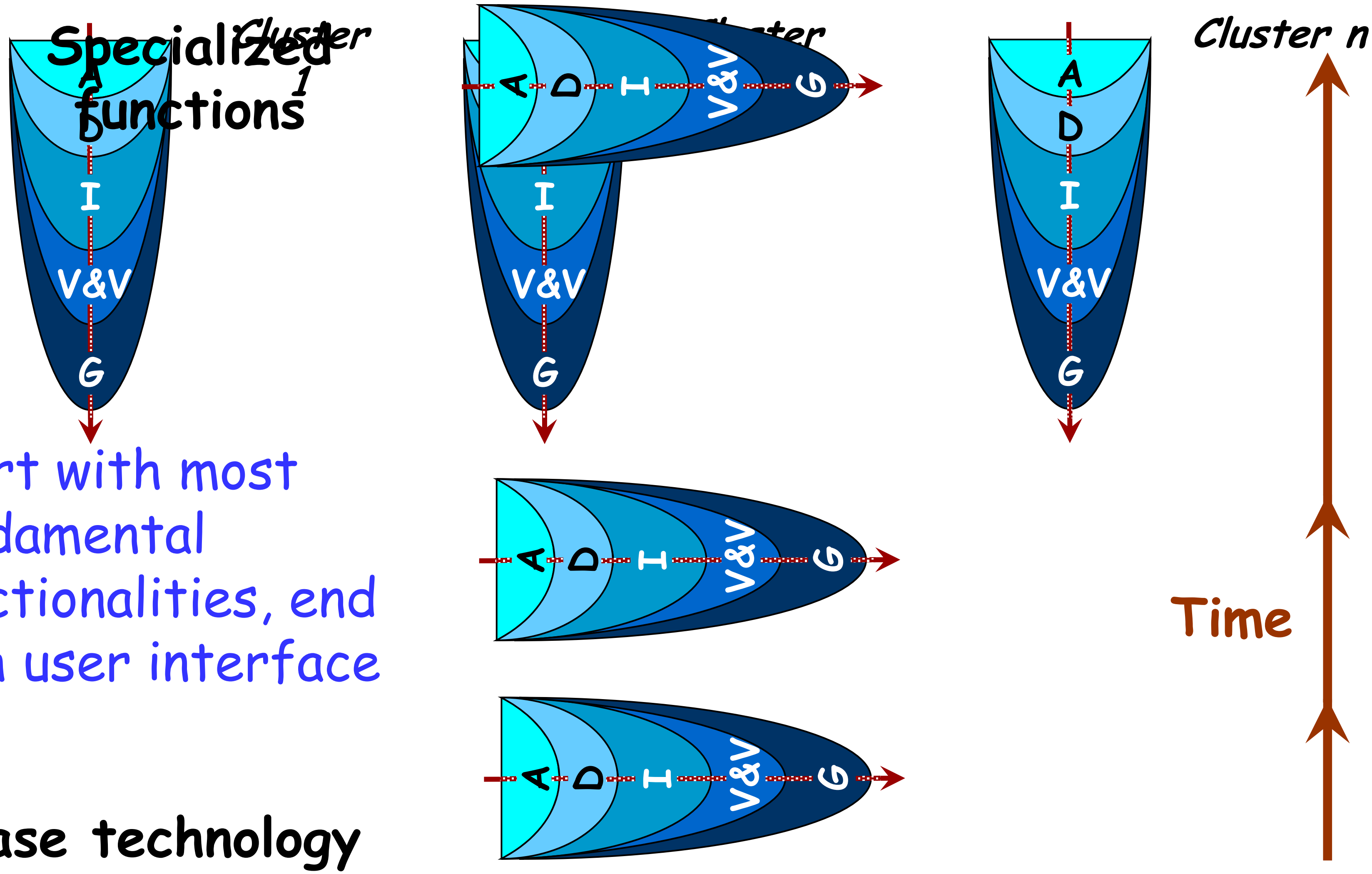
Cluster 1

Cluster 2

Cluster 3

Cluster 4

Bottom-up order of cluster development



Specialized functions

Start with most fundamental functionalities, end with user interface

Base technology

Seamless development with EiffelStudio

Diagram Tool

- System diagrams can be produced automatically from software text
- Works both ways: update diagrams or update text – other view immediately updated

No need for separate UML tool

Metrics Tool

Profiler Tool

Documentation generation tool

...

The Eiffel Software development process

Small group (8-12 developers)
Tightly knit group, have worked together for many years
Geographically distributed
All have commit rights
Experts in one particular area, but conversant with the rest of the technology

Full-fledged compiler & IDE with numerous libraries
2.5 million lines of code (all Eiffel except about 100,000 in C)
Open-source and commercial licenses
Highly portable, all major industry platforms
Incorporates numerous outside contributions

Timeboxed development: 2 releases a year (15 November and 15 May)

Cluster model

- 4.2 -

Empirical evidence

Technique adoption

Source: Ambler 2006

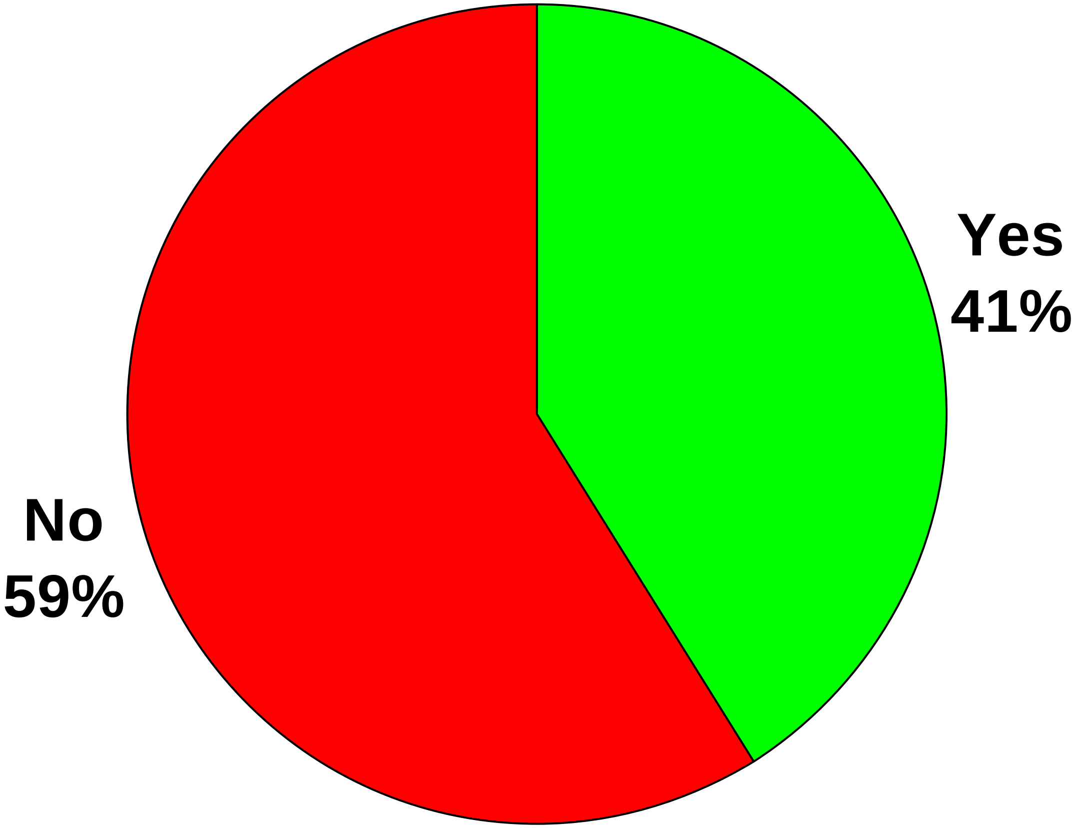
(Multiple answers)

Active Stakeholder Participation	938
AMDD	260
Code Refactoring	1467
Code Regression Testing	1383
Co-location	447
Common coding guidelines	1595
Continuous integration	1113
Database refactoring	416
Database regression testing	407
Pair programming	587
Single sourcing information	241
TDD	959

Agile adoption

Source: Ambler 2006

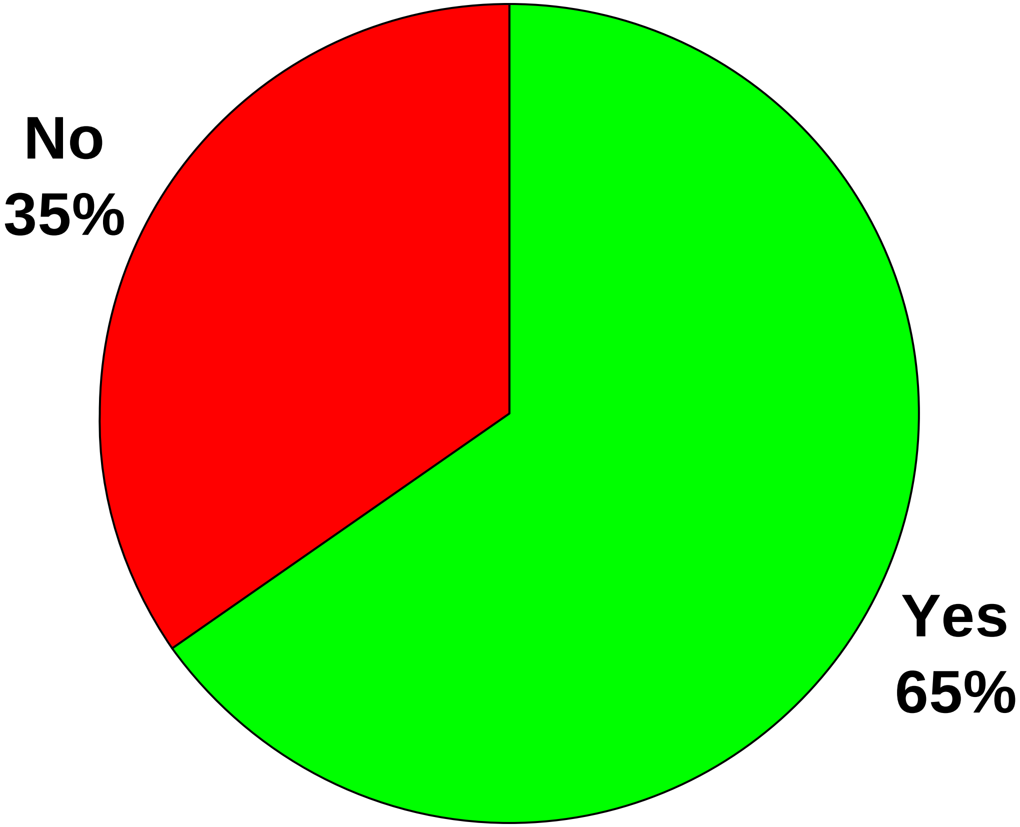
Have you adopted an agile methodology?



Agile adoption

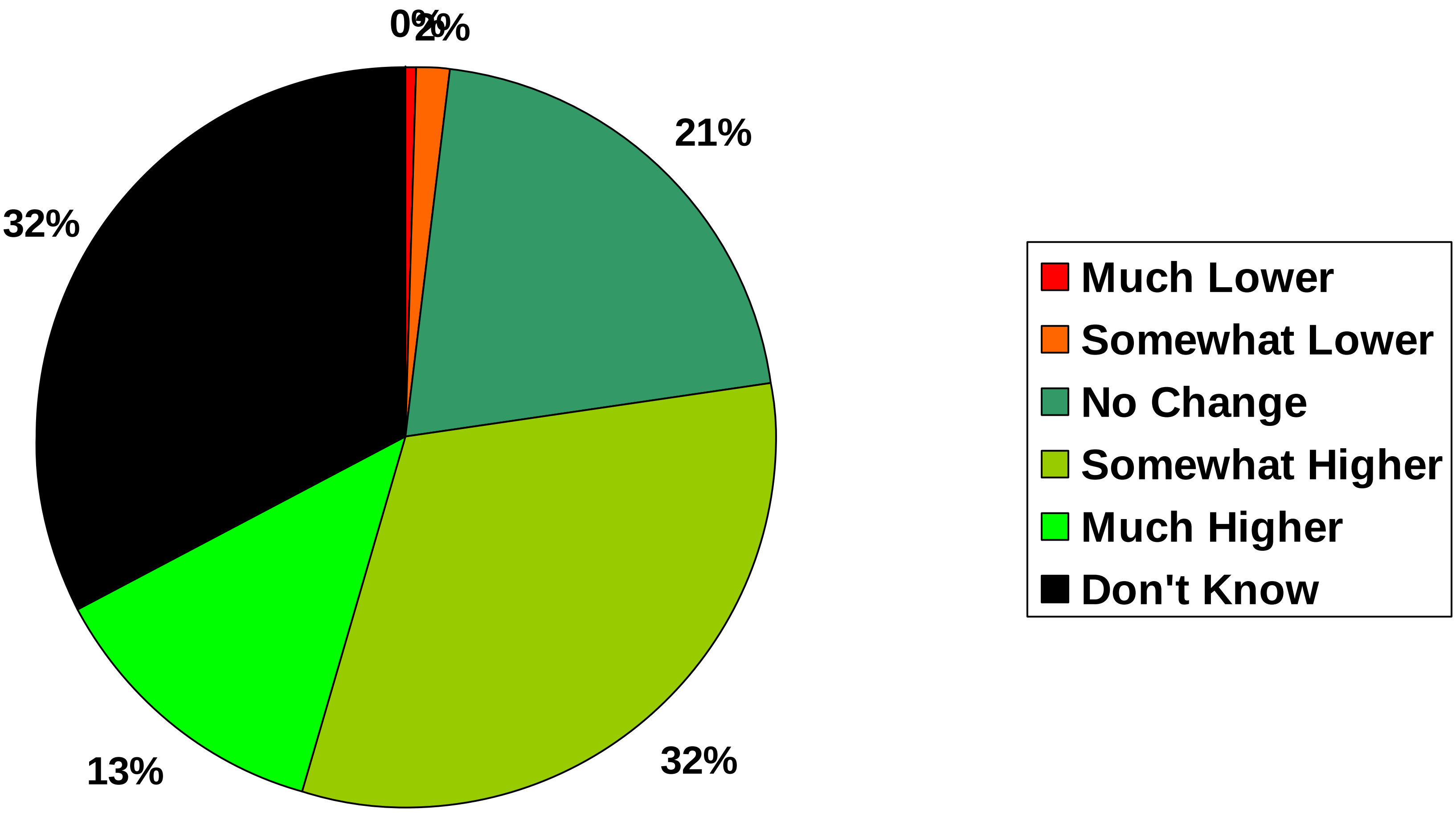
Source: Ambler 2006

Have you adopted any agile technique?



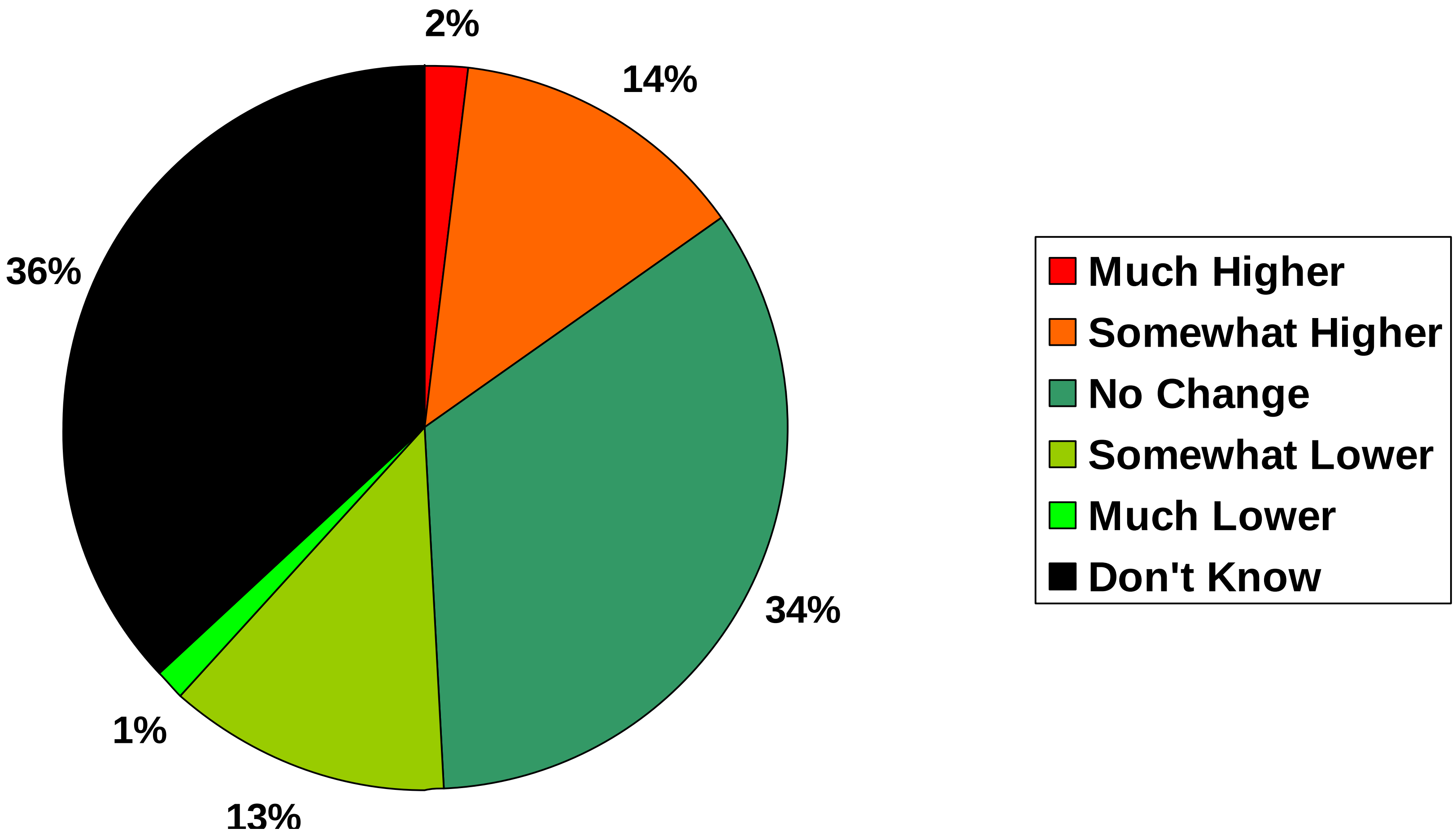
Effect on quality

Source: Ambler 2006



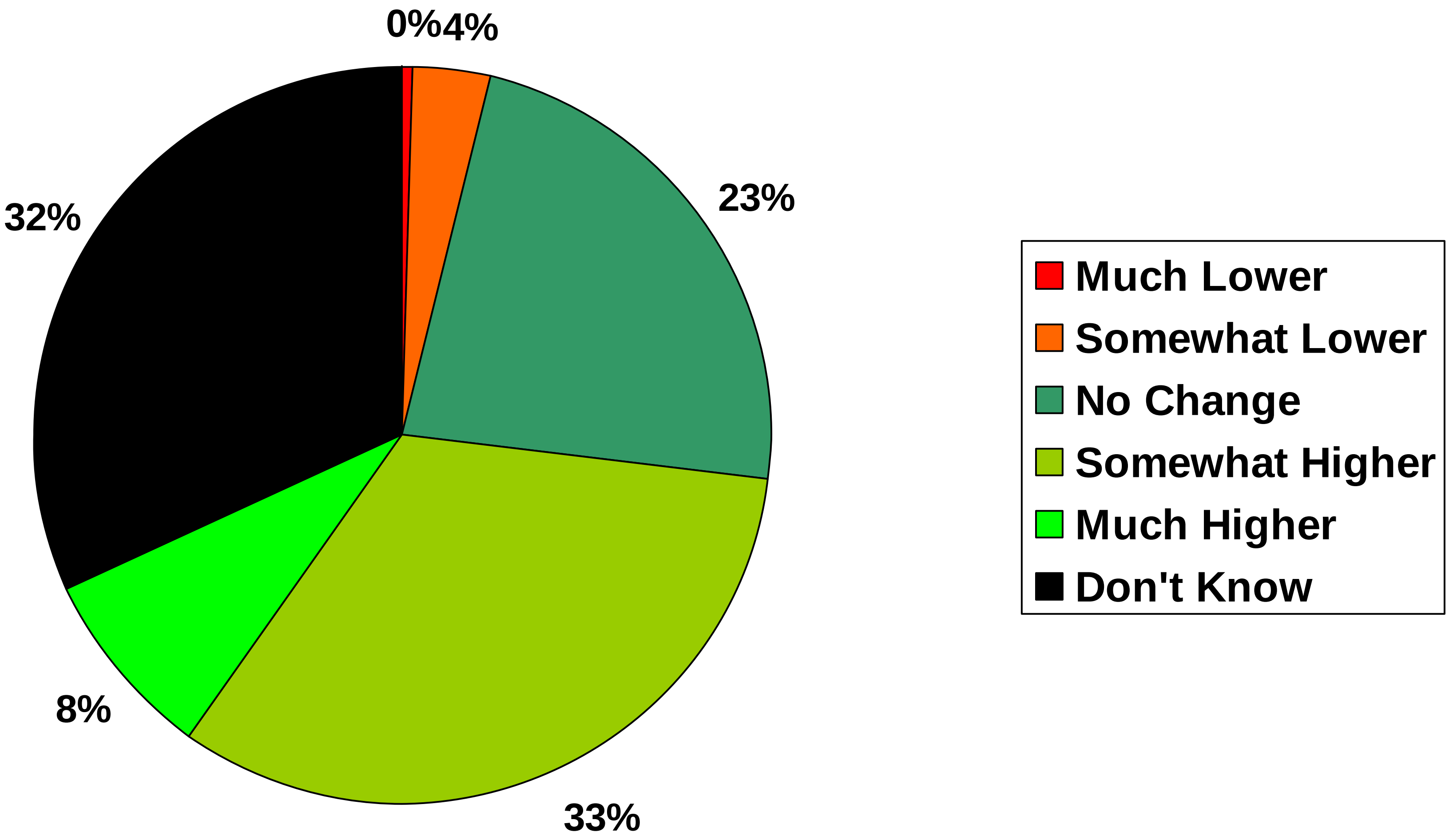
Effect on cost

Source: Ambler 2006



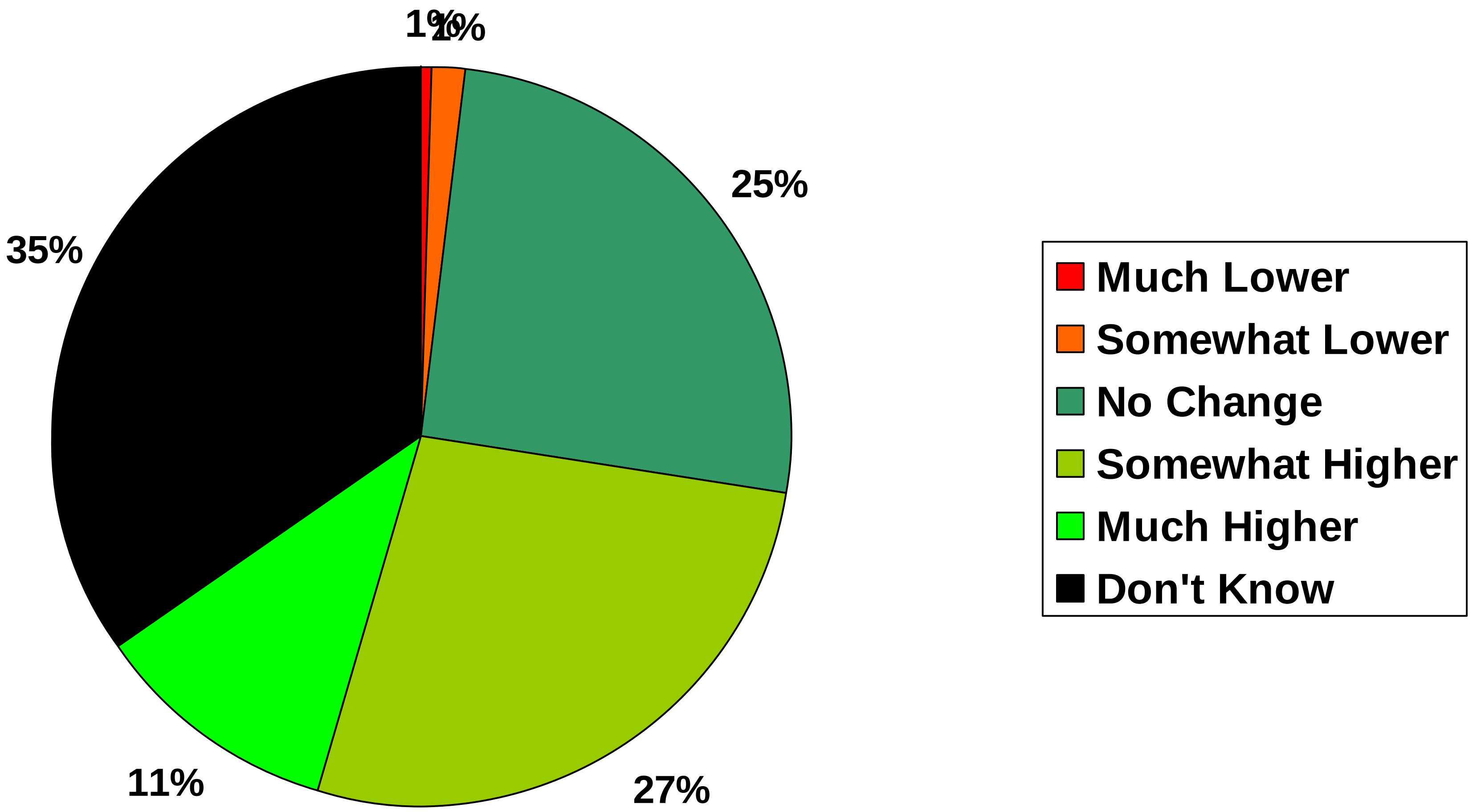
Effect on productivity

Source: Ambler 2006



Effect on customer satisfaction

Source: Ambler 2006



Bad experiences

Source: Ambler 2006

91 (2%) respondents had at least one really bad experience:

- 0. 5% had much lower productivity
- 0.5% had much lower quality
- 1.7% had much higher cost
- 0.5% had much lower business satisfaction

709 (17%) had some bad experience: above, plus

- 3% had somewhat lower productivity
- 1.5% had somewhat lower quality
- 13% had somewhat higher cost
- 1.4% had somewhat lower business satisfaction

There was a correlation between knowledge and results

- E.g. The people knowledgeable with agile approaches had better quality, stakeholder satisfaction, … than those who weren't knowledgeable

Pair programming

Source: Madeyski 2010

Speedup Ratio:

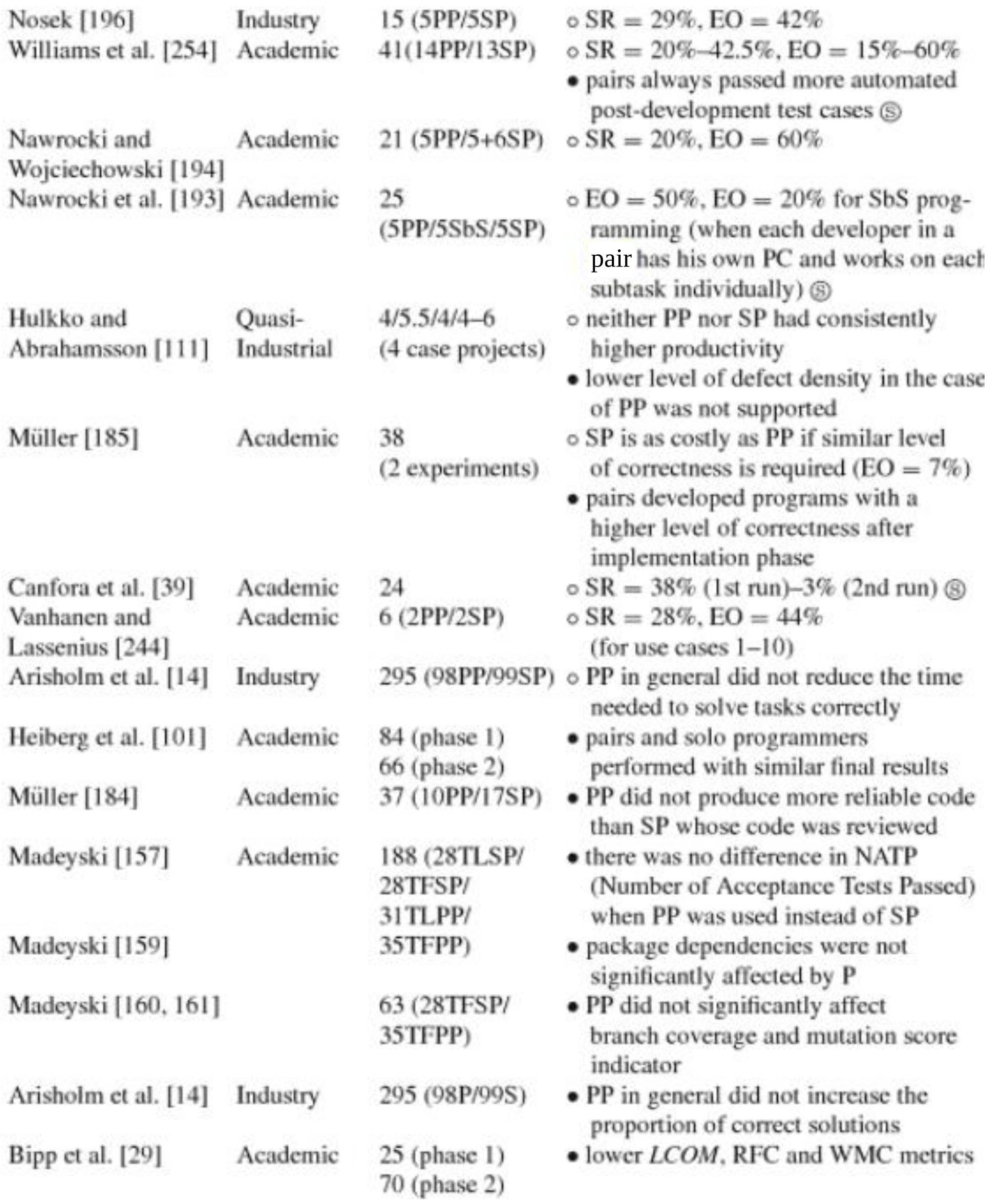
$$\mathrm{SR} = \frac{\text{Completion Time Of Solo Programmer} - \text{Completion Time Of Pair}}{\text{Completion Time Of Solo Programmer}} \times 100\%$$

Effort Overhead:

$$\mathrm{EO} = \frac{2 \times \text{Completion Time Of Pair} - \text{Completion Time Of Solo Programmer}}{\text{Completion Time Of Solo Programmer}} \times 100\%$$

Pair programming: results

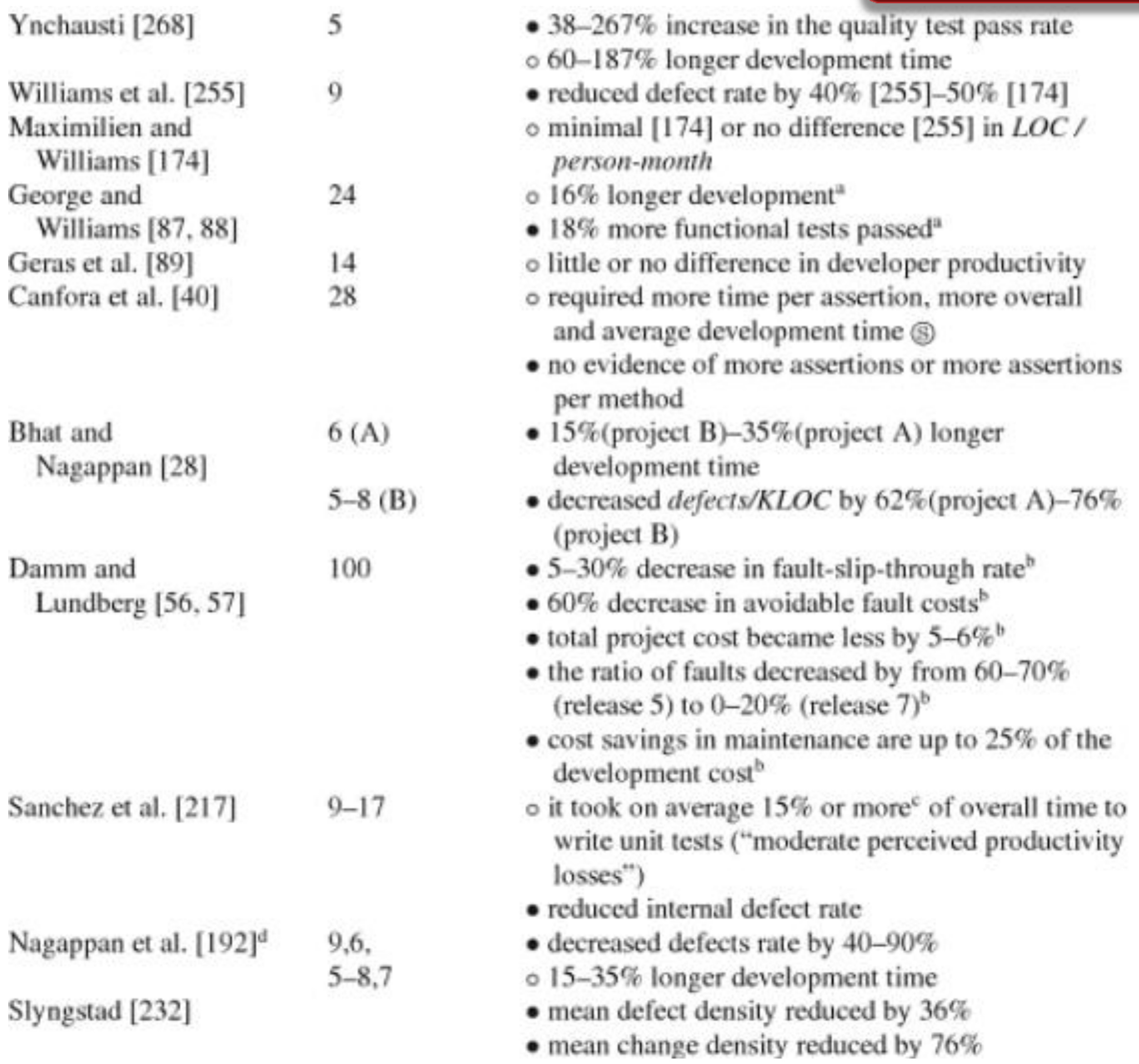
Source: Madeyski 2010



Nosek [196]	Industry	15 (5PP/5SP)	○ SR = 29%, EO = 42%
Williams et al. [254]	Academic	41(14PP/13SP)	○ SR = 20%–42.5%, EO = 15%–60% • pairs always passed more automated post-development test cases Ⓢ
Nawrocki and Wojciechowski [194]	Academic	21 (5PP/5+6SP)	○ SR = 20%, EO = 60%
Nawrocki et al. [193]	Academic	25 (5PP/5SbS/5SP)	○ EO = 50%, EO = 20% for SbS programming (when each developer in a pair has his own PC and works on each subtask individually) Ⓢ
Hulkko and Abrahamsson [111]	Quasi-Industrial	4/5.5/4/4–6 (4 case projects)	○ neither PP nor SP had consistently higher productivity • lower level of defect density in the case of PP was not supported
Müller [185]	Academic	38 (2 experiments)	○ SP is as costly as PP if similar level of correctness is required (EO = 7%) • pairs developed programs with a higher level of correctness after implementation phase
Canfora et al. [39]	Academic	24	○ SR = 38% (1st run)–3% (2nd run) Ⓢ
Vanhanen and Lassenius [244]	Academic	6 (2PP/2SP)	○ SR = 28%, EO = 44% (for use cases 1–10)
Arisholm et al. [14]	Industry	295 (98PP/99SP)	○ PP in general did not reduce the time needed to solve tasks correctly
Heiberg et al. [101]	Academic	84 (phase 1) 66 (phase 2)	• pairs and solo programmers performed with similar final results
Müller [184]	Academic	37 (10PP/17SP)	• PP did not produce more reliable code than SP whose code was reviewed
Madeyski [157]	Academic	188 (28TLSP/ 28TFSP/ 31TLPP/ 35TFPP)	• there was no difference in NATP (Number of Acceptance Tests Passed) when PP was used instead of SP
Madeyski [159]			• package dependencies were not significantly affected by P
Madeyski [160, 161]		63 (28TFSP/ 35TFPP)	• PP did not significantly affect branch coverage and mutation score indicator
Arisholm et al. [14]	Industry	295 (98P/99S)	• PP in general did not increase the proportion of correct solutions
Bipp et al. [29]	Academic	25 (phase 1) 70 (phase 2)	• lower *LCOM*, RFC and WMC metrics

Test-first programming studies (industrial)

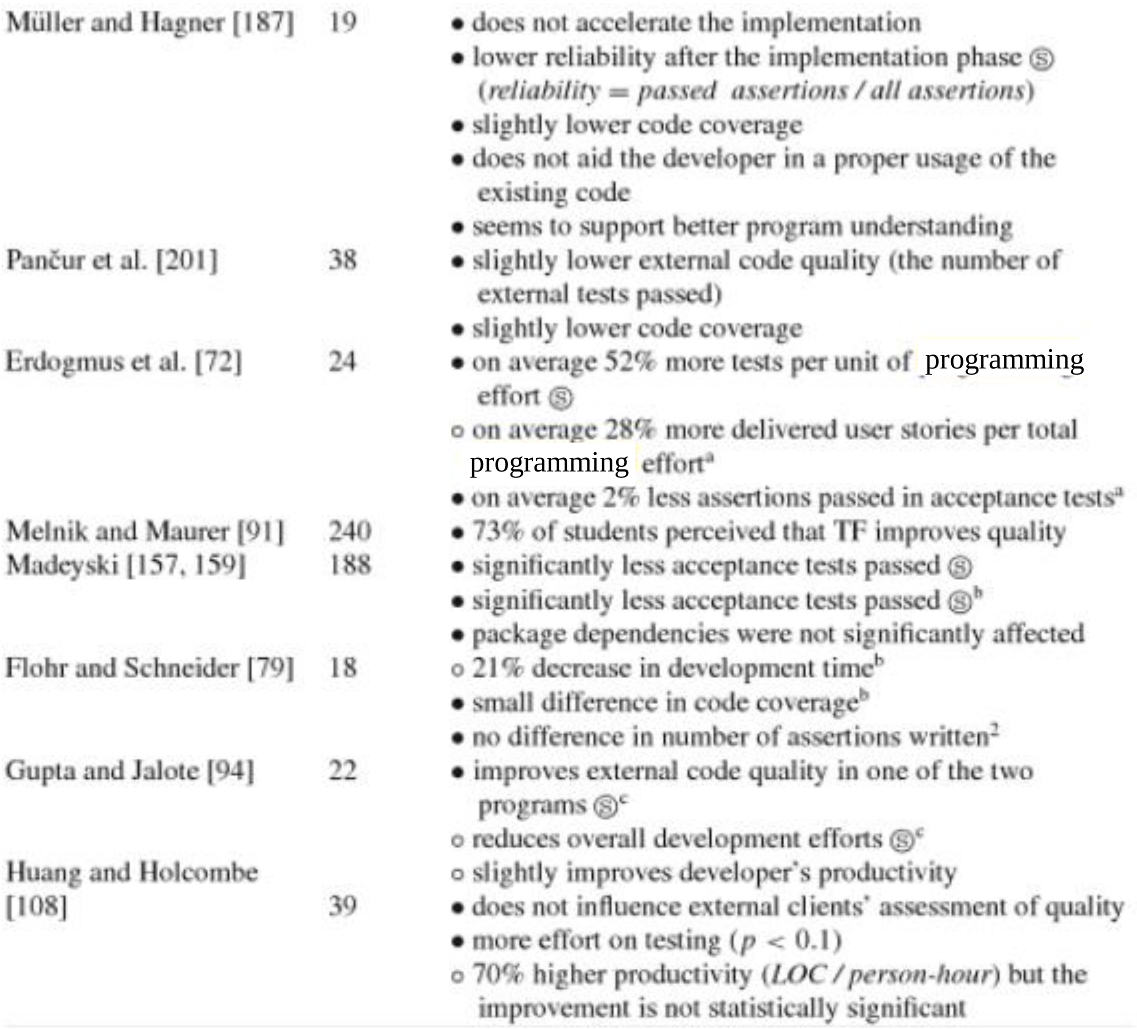
Source: Madeyski 2010



Ynchausti [268]	5	● 38–267% increase in the quality test pass rate ○ 60–187% longer development time
Williams et al. [255] Maximilien and Williams [174]	9	● reduced defect rate by 40% [255]–50% [174] ○ minimal [174] or no difference [255] in *LOC / person-month*
George and Williams [87, 88]	24	○ 16% longer development[a] ● 18% more functional tests passed[a]
Geras et al. [89]	14	○ little or no difference in developer productivity
Canfora et al. [40]	28	○ required more time per assertion, more overall and average development time Ⓢ ● no evidence of more assertions or more assertions per method
Bhat and Nagappan [28]	6 (A) 5–8 (B)	● 15%(project B)–35%(project A) longer development time ● decreased *defects/KLOC* by 62%(project A)–76% (project B)
Damm and Lundberg [56, 57]	100	● 5–30% decrease in fault-slip-through rate[b] ● 60% decrease in avoidable fault costs[b] ● total project cost became less by 5–6%[b] ● the ratio of faults decreased by from 60–70% (release 5) to 0–20% (release 7)[b] ● cost savings in maintenance are up to 25% of the development cost[b]
Sanchez et al. [217]	9–17	○ it took on average 15% or more[c] of overall time to write unit tests ("moderate perceived productivity losses") ● reduced internal defect rate
Nagappan et al. [192][d]	9,6, 5–8,7	● decreased defects rate by 40–90% ○ 15–35% longer development time
Slyngstad [232]		● mean defect density reduced by 36% ● mean change density reduced by 76%

Test-first programming studies (academic)

Source: Madeyski 10



Müller and Hagner [187]	19	• does not accelerate the implementation • lower reliability after the implementation phase Ⓢ (*reliability = passed assertions / all assertions*) • slightly lower code coverage • does not aid the developer in a proper usage of the existing code • seems to support better program understanding
Pančur et al. [201]	38	• slightly lower external code quality (the number of external tests passed) • slightly lower code coverage
Erdogmus et al. [72]	24	• on average 52% more tests per unit of programming effort Ⓢ ○ on average 28% more delivered user stories per total programming effort[a] • on average 2% less assertions passed in acceptance tests[a]
Melnik and Maurer [91]	240	• 73% of students perceived that TF improves quality
Madeyski [157, 159]	188	• significantly less acceptance tests passed Ⓢ • significantly less acceptance tests passed Ⓢ[b] • package dependencies were not significantly affected
Flohr and Schneider [79]	18	○ 21% decrease in development time[b] • small difference in code coverage[b] • no difference in number of assertions written[2]
Gupta and Jalote [94]	22	• improves external code quality in one of the two programs Ⓢ[c] ○ reduces overall development efforts Ⓢ[c] ○ slightly improves developer's productivity
Huang and Holcombe [108]	39	• does not influence external clients' assessment of quality • more effort on testing ($p < 0.1$) ○ 70% higher productivity (*LOC / person-hour*) but the improvement is not statistically significant

Pair programming

Source: Müller 05

Analysis of pair programming vs traditional code reviews

Results indicate that pairs and single programmers applying code reviews:

- Produce programs at a similar level of correctness
- Cost about the same

- 5 -

Conclusion

The good

- Acceptance of change
- Frequent iterations
- Emphasis on working code
- Tests as one of the key resources of the project
- Constant test regression analysis
- No branching
- Product (but not user stories!) burndown chart
- Daily meeting

The hype

- Pair programming
- Role of the manager
- Method keeper (e.g. ScrumMaster) as a separate role
- Planning poker
- Open offices

The ugly

- No upfront requirements
- Tests as a replacement for specifications
- User stories as a replacement for abstract requirements
- Rejection of auxiliary products
- Rejection of a priori concern for extendibility
- Rejection of a priori concern for reusability
- Rejection of a priori architecture work
- Rejection of non-shippable artifacts

Another classification

Your work, Sir, is both new and good, but what's new is not good and what's good is not new

Samuel Johnson

Good but not new

Iterative development

Role of change

Not new and not good

User stories as a substitute for requirements

New and not good

Rejection of up-front requirements

Test as a substitute for specifications

New and good!

Team empowerment (not entirely new, cf. TSP)

Daily meeting

Central role of tests, especially regression test suite

No development or bug fix without a test

Central role of code

Final observations

Your work, Sir, is both new and good, but what's new is not good and what's good is not new

Samuel Johnson

For every complex problem there is an answer that is clear, simple, and wrong

H.L. Mencken

ANDROMAQUE: I do not understand abstractions.

CASSANDRA: As you like. Let us resort to metaphors.

Jean Giraudoux, *The Trojan War Will Not Happen*

Software development is hard

Software quality is key

Lots of good ideas can help; there is no reason to reject those from any particular style of software engineering

Bibliography (1/4)

Agile Manifesto, at agilemanifesto.org/.

Scott Ambler: *Agile Adoption Rate Survey*, at www.ambysoft.com/surveys/agileMarch2006.html.

Bachan Anand: *Conscires* site at agile.conscires.com/.

Kent Beck and Cynthia Andres: *Extreme Programming Explained*, Addison-Wesley, 2nd edition, 2005.

Barry W. Boehm: *Software Engineering Economics*, Prentice Hall, 1981.

Barry W. Boehm & Richard Turner: *Balancing Agility and Discipline – A Guide for the Perplexed*, Addison-Wesley, 2004.

Fred Brooks: *The Mythical Man-Month*, Addison-Wesley, 1975.

Brad Cox: *Superdistribution: Objects as Property on the Electronic Frontier*, Addison Wesley. 1996.

Mike Cohn, *Succeeding With Agile*, Addison-Wesley, 2010.

Mike Cohn, *Succeeding With Agile* site, www.mountaingoatsoftware.com.

Chromatic: *Extreme Programming Pocket Guide*, O'Reilly, 2003.

Bibliography (2/4)

Martin Fowler: *Refactoring: Improving the design of existing code*, Addison Wesley, 1999.

Tom deMarco and Tim Lister: *Peopleware: Productive Projects and Teams* (Second Edition), Dorset House, 1999.

Ron Jeffries: *Xprogramming* site at xprogramming.com.

Lech Madeyski: *Test-Driven Development – An Empirical Development of Agile Practice*, Springer Verlag, 2010.

Bertrand Meyer: *Object Success: A Manager's Guide to Object Orientation, Its Impact on the Corporation and its Use for Reengineering the Software Process*, Prentice Hall, 1995.

Bertrand Meyer: *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.

Matthias Müller: *Two controlled experiments concerning the comparison of pair programming to peer review*, in *Journal of Systems and Software* 78, 2005, pages 166-179.

Bibliography (3/4)

Bertrand Meyer: *Touch of Class: Learning to Program Well, Using Objects and Contracts*, Springer-Verlag, 2009.

Bertrand Meyer: *Design and Code Reviews in the Age of the Internet*, in *Communications of the ACM*, vol. 51, no. 9, September 2008, pages 66-71.

Mary &Tom Poppendieck: *Lean Software Development* site, www.poppendieck.com.

Winston D. Royce: *Managing the Development of Large Software Systems*, in Proceedings of IEEE WESCON, 1970, pages 1-9.

James Shore & Shane Warden: *The Art of Agile Development*, O'Reilly. 2008.

Andrew Stellman & Jennifer Greene: *Building Better Software* site, www.stellman-greene.com/.

Matt Stephens & Doug Rosenberg: *Extreme Programming Refactored: The Case Against XP*, Apress, 2003.

Bibliography (4/4)

James Tomayko James Herbsleb: *How Useful Is the Metaphor Component of Agile Methods? A Preliminary Study*, Report CMU-CS-03-152, School of Computer Science, Carnegie-Mellon University, June 2003

Bill Wake: *Exploring Extreme Programming* site at xp123.com.

Doug Wallace, Isobel Raggett & Joel Aufgang: *Extreme Programming for Web Projects*, Addison-Wesley, 2002.

Kelly Waters: *All About Agile* site, www.allaboutagile.com/.

Don Wells: *Extreme Programming* site, www.extremeprogramming.org/.

Joel Wenzel, *In Point Form* site, joel.inpointform.net.