AGILE!
The good,
The hype &
The ugly

Bertrand Meyer

Material copyright
Bertrand Meyer, 2012
Topics

Part 1: Overview
Part 2: The “enemy”: Big Upfront Everything
Part 3: What is “agile”?  
  ➢ 3.1 Agile principles  
  ➢ 3.2 Agile roles  
  ➢ 3.3 Agile 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  
  - 3.3 Agile 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...

![Diagram showing cost and time relationship with 25% nominal cost & time]
Assertions

Revolutionary

Better

Everyone else is doing it wrong

All or nothing
Not everyone is ecstatic...

Source: Stephens 03
Overview
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
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

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.

Source: Agile manifesto
Twelve principles

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.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
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.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
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.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Source: Agile Manifesto
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

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.

*Source: Meyer 1997*
The “lean” view

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*)

(Source: Poppendieck)
Not everyone is ecstatic...

Source: Stephens 03

The C3 Project Life Cycle

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

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 "right" from the "fragile"
- Proposes better ways of achieving XP’s agile goals that are applicable to a much wider range of projects
- 2 -

The “enemy”:
Big Upfront
Everything
Big Upfront Anything
The revolt of the cubicles
All you need is code

Code is all you need
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
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
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.

Figure 1. Implementation steps to deliver a small computer program for internal operations.
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.
Figure 3. Hopefully, the iterative interaction between the various phases is confined to successive steps.
Figure 4. Unfortunately, for the process illustrated, the design iterations are never confined to the successive steps.
The waterfall model of the lifecycle

1. Feasibility study
2. Requirements
3. Specification
4. Global design
5. Detailed design
6. Implementation
7. V & V
8. Distribution
Arguments for the waterfall

(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

1. Feasibility study
2. Requirements
3. Specification
4. Global design
5. Detailed design
6. Implementation
7. V & V (Verification and Validation)
8. Distribution
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.

(Pre-1970 cartoon; origin unknown)
A modern variant

1. How the customer explained it
2. How the Project Leader understood it
3. How the Analyst designed it
4. How the Programmer wrote it
5. How the Business Consultant described it
6. How the project was documented
7. What operations installed
8. How the customer was billed
9. How it was supported
10. What the customer really needed
The spiral model

- Iteration 1
- Iteration 2
- Iteration 3
The spiral model

Apply a waterfall-like approach to successive prototypes

1. Determine objectives
2. Identify and resolve risks
3. Development and Test
4. Plan the next iteration

Key stages:
-_cumulative cost
- progress
- review
- implementation
- release
- prototype 1
- prototype 2
- operational prototype
- detailed design
- code
- integration
- test
- development
- verification & validation
- requirements
- concept of requirements
- concept of operation
- requirements plan
- development plan
- test plan
- draft
CMMI background

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

1. Process unpredictable, poorly controlled and reactive

2. Process characterized for projects and is often reactive

3. Process characterized for the organization and is proactive

4. Process measured and controlled

5. Focus on process improvement
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

For 120 projects in Boeing Information Systems

- Without Historical Data: Variance: + 20% to -145% (Mostly Level 1 & 2)
- With Historical Data: Variance: - 20% to + 20% (Level 3)

## Generic goals and practices

<table>
<thead>
<tr>
<th>Level</th>
<th>Capability</th>
<th>Generic Goals</th>
<th>Generic Practices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Achieve Specific Goals</strong></td>
<td></td>
<td>GP 1.1 Perform Base Practices</td>
</tr>
</tbody>
</table>
| 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
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

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

Source for PSP material: Software Engineering Institute
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

Requirements

Planning
Design
Code
Compile
Test
Postmortem

Scripts

Logs

Finished product

Project summary

Project and process data summary report
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 behavior

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

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

Source: Meyer 08
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_D).
- a better error component will be used instead of current clumsy system
- and it should benefit redesigns related to debuggers.

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://tinynurl.com/5886rm
* DBG_EXPRESSION_EVALUATOR: http://tinynurl.com/6989cd
* DBG_EXPRESSION_EVALUATOR_B: http://tinynurl.com/695sk
* DBG_EVALUATOR: http://tinynurl.com/37ind
* AST_DEBUGGER_EXPRESSION_CHECKER_GENERATOR: http://tinynurl.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 some of 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 simplify the interface, and the design, and this would not 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 EXP_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 7/1/08 3:47 PM

Larry 1.1 I don’t know why {DBG_EXPRESSION_EVALUATOR_B} and {DBG_EXPRESSION_EVALUATOR_B} using inheritance since {DBG_EXPRESSION_EVALUATOR} only have one benefit is? Maybe just use delegation is simpler and clearer… Similar 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 far I did not both implement want to break the debugger, and I did want to use Swing 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 new code file, 2) a new kind of evaluator which would send the expression bytecode to the debugger in order to be evaluated on the debugger. 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
Part 1: Overview

Part 2: The “enemy”: Big Upfront Everything

Part 3: What is “agile”?
  3.1 Agile principles
  3.2 Agile roles
  3.3 Agile 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
What is agile?
3 What is agile?

- 3.1 - Agile principles
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

© 2001, the Java authors. This document may be freely copied in any form, but only as it appears throughout this notice.
Twelve principles

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.

Source: Agile manifesto
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
“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

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.

Source: Cohn (slightly abridged)
Eliminate waste

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

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

Source: Poppendieck

<table>
<thead>
<tr>
<th>The Seven Wastes of Manufacturing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overproduction</td>
</tr>
<tr>
<td>Inventory</td>
</tr>
<tr>
<td>Extra Processing Steps</td>
</tr>
<tr>
<td>Motion</td>
</tr>
<tr>
<td>Defects</td>
</tr>
<tr>
<td>Waiting</td>
</tr>
<tr>
<td>Transportation</td>
</tr>
</tbody>
</table>
Minimize artifacts (inventory)

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

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.

Source: Poppendieck
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

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.

Source: Poppendieck
Deliver as fast as possible

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

Source: Poppendieck
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).
Another key idea from Toyota 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

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
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”
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

See “sustainable pace”
No overtime

See “sustainable pace”

Yourdon, “Death March” (1999)
Encourage free expression of ideas
Do not ridicule anyone because of a question or suggestion
Build trust within the team
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

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

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 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)
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

Keep the team together and stable

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

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
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

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

*Code that does not pass tests is waste*
“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.”
“A bug is not an error in logic, it is a test you forgot to write”
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.

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

Source: Wells
Only one pair integrates code at a time

Collective code ownership
Development proceeds in parallel

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

<table>
<thead>
<tr>
<th>Project</th>
<th>Metaphor</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrist camera</td>
<td>Portrait studio</td>
<td>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.</td>
</tr>
<tr>
<td>Wrist camera</td>
<td>Cities and Towns</td>
<td>(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.</td>
</tr>
<tr>
<td>Variable transparency window</td>
<td>Chameleon</td>
<td>The window will use nanotechnology to vary opaqueness depending on sensor readings, e.g., temperature. It can also generate decorative patterns. This is the architecture project.</td>
</tr>
<tr>
<td>Financial planner</td>
<td>Human financial planner</td>
<td>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.</td>
</tr>
<tr>
<td>Department of Transportation (DOT) web site</td>
<td>TurboTax</td>
<td>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.</td>
</tr>
<tr>
<td>Ford Motor Company software architecture tool</td>
<td>C compiler</td>
<td>A tool is being developed to combine architectures of components into one major architectural artifact.</td>
</tr>
</tbody>
</table>

*Source: Tomayko 03*
Incremental design

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.”

Source: Shore 08
Incremental deployment

Deploy functionality gradually

“Big Bang” deployment is risky
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

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

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

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

Source: Wake
Ten-minute build

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 functional tests by unit tests.

Source: Shore 08
"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
Maintain a single code base: avoid branching, even if permitted by configuration management system
Access to automated tests, configuration management, frequent integration, code repository
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)**
Maintain only code and tests as permanent artifacts
Pay-per-use

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
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
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
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

Source: Sutherland
Manager

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 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

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.
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

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

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 What is agile?

- 3.3 -

Agile artifacts
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

<table>
<thead>
<tr>
<th>Name</th>
<th>UC-8: Search and Replace</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
<td>All occurrences of a search term are replaced with replacement text. 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.</td>
</tr>
<tr>
<td><strong>Rationale</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Users</strong></td>
<td>All users</td>
</tr>
<tr>
<td><strong>Preconditions</strong></td>
<td>A document is loaded and being edited. 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.</td>
</tr>
<tr>
<td><strong>Basic Course of Events</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Alternative Paths</strong></td>
<td>All occurrences of the search term have been replaced with the replacement text.</td>
</tr>
</tbody>
</table>

**Source:** Steelman & Greene
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.
“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 ...”
"As a <user_or_role>
I want <business_functionality>
so that <business_justification>"
Example user story

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

Source: Waters
Story card

From the original C3 project:

DATE: 3/19/98

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

STORY NUMBER: 1275

PRIORITY: USER: ____ TECH: ____

PRIOR REFERENCE: __________

RISK: __________

TECH ESTIMATE: __________

TASK DESCRIPTION:

SPLIT COLA: When the COLA rate chgs in the middle of the BlW 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.

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 adjustment. Create RM Boundary and Place in DEEntExRes COLA

TASK TRACKING:

<table>
<thead>
<tr>
<th>Date</th>
<th>Status</th>
<th>To Do</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Task card

From the original C3 project:

**Engineering Task Card**

**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:**

<table>
<thead>
<tr>
<th>Date</th>
<th>Done</th>
<th>To Do</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Use cases vs user stories

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

"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

<table>
<thead>
<tr>
<th>X</th>
<th>f (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Product backlog

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”.
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

<table>
<thead>
<tr>
<th>Benefits</th>
<th>Story</th>
<th>To Do</th>
<th>In Process</th>
<th>To Verify</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>As a user, I... 8 points</td>
<td>Code the...9</td>
<td>Test the...8</td>
<td>Code the...DC 4</td>
<td>Test the...SC 6</td>
</tr>
<tr>
<td></td>
<td>Code the...2</td>
<td>Code the...8</td>
<td>Test the...4</td>
<td>Test the...SC 8</td>
<td>SC 6</td>
</tr>
<tr>
<td></td>
<td>Test the...8</td>
<td>Test the...4</td>
<td></td>
<td>Test the...SC 6</td>
<td>SC 6</td>
</tr>
<tr>
<td></td>
<td>As a user, I... 5 points</td>
<td>Code the...8</td>
<td>Test the...8</td>
<td>Code the...DC 8</td>
<td>Test the...SC 6</td>
</tr>
<tr>
<td></td>
<td>Code the...4</td>
<td>Code the...6</td>
<td></td>
<td>Test the...SC 6</td>
<td>SC 6</td>
</tr>
</tbody>
</table>

*Source: Cohn, Anand*
Story board

Source: Cohn
Burndown chart

Publicly displayed chart, updated every day, showing

- Remaining work
- Progress

in the Sprint backlog
Bullpen

Single, open room
(See “Informative workspace” principle)
3 What is agile?

- 3.4 -

Agile practices
Onsite customer

Seen under Principles
Pair programming

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

“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

Source: Fowler
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
- 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

Source: Cohn
Test-Driven Development

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

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

- 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.)
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.

Source: Jeffries
Osmotic communication

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

The combination of frequent releases with relentless testing

Keep system fully integrated at all times
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

See “shared code” principle
Simple design

Produce the simplest design that works

Refactor as needed
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

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

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

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

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

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

Emphasizes management rather than specific software techniques

Has been used in conjunction with CMMI
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

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**

**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.

*Source: Sutherland Scrum*
Overall Scrum process

Inputs from Customers, Team, Managers, Execs

Product Owner

The Team

Sprint Backlog

Sprint Planning Meeting

Team selects starting at top as much as it can commit to deliver by end of Sprint

Task Breakout

Sprint Backlog

1-4 Week Sprint

Sprint end date and team deliverable do not change

Daily Standup Meeting

Scrum Master

Sprint Review

Finished Work

Sprint Retrospective

Source: Sutherland
Extreme Programming (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

Unfinished Features
Most Important Features
Iterative Planning
A Project Heartbeat
Working Software
Team Empowerment
Honest Plans
Daily Communication

Source: Wells
XP criteria

- 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

Source: Wells
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

- 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

Source: Cockburn

- 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

Colors:
- Clear
- Yellow
- Orange
- Red
- Maroon
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 Critical analysis

- 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"
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);
}
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 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
<table>
<thead>
<tr>
<th>$x$</th>
<th>$f(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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
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

Cluster 1

Cluster 2
Extremes

“Trickle”

Cluster 1

Cluster 2

“Clusterfall”

Cluster 1

Cluster 1

Cluster 2

Cluster 2
Dynamic rearrangement
Bottom-up order of cluster development

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 Critical analysis

- 4.2 -

Empirical evidence
## Technique adoption

*(Multiple answers)*

<table>
<thead>
<tr>
<th>Technique</th>
<th>Adoption Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Stakeholder Participation</td>
<td>938</td>
</tr>
<tr>
<td>AMDD</td>
<td>260</td>
</tr>
<tr>
<td>Code Refactoring</td>
<td>1467</td>
</tr>
<tr>
<td>Code Regression Testing</td>
<td>1383</td>
</tr>
<tr>
<td>Co-location</td>
<td>447</td>
</tr>
<tr>
<td>Common coding guidelines</td>
<td>1595</td>
</tr>
<tr>
<td>Continuous integration</td>
<td>1113</td>
</tr>
<tr>
<td>Database refactoring</td>
<td>416</td>
</tr>
<tr>
<td>Database regression testing</td>
<td>407</td>
</tr>
<tr>
<td>Pair programming</td>
<td>587</td>
</tr>
<tr>
<td>Single sourcing information</td>
<td>241</td>
</tr>
<tr>
<td>TDD</td>
<td>959</td>
</tr>
</tbody>
</table>

*Source: Ambler 2006*
Agile adoption

Have you adopted an agile methodology?

- Yes: 41%
- No: 59%

Source: Ambler 2006
Have you adopted any agile technique?

- Yes: 65%
- No: 35%

Source: Ambler 2006
Effect on quality

Source: Ambler 2006
Effect on cost

Source: Ambler 2006
Effect on productivity

Source: Ambler 2006

- Much Lower: 32%
- Somewhat Lower: 4%
- No Change: 23%
- Somewhat Higher: 8%
- Much Higher: 33%
- Don't Know: 0%
Effect on customer satisfaction

- Much Lower: 11%
- Somewhat Lower: 1%
- No Change: 27%
- Somewhat Higher: 25%
- Much Higher: 35%
- Don't Know: 1%

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.

Source: Ambler 2006
Pair programming

Speedup Ratio:

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

Effort Overhead:

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

Source: Madeyski 2010
## Pair programming: results

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

Source: Madeyski 2010
<table>
<thead>
<tr>
<th>Study</th>
<th>N</th>
<th>Findings</th>
</tr>
</thead>
</table>
| Ynchausti [268]              | 5  | - 38–267% increase in the quality test pass rate  
  - 60–187% longer development time      |
| Williams et al. [255]        | 9  | - reduced defect rate by 40% [255]-50% [174]  
  - minimal [174] or no difference [255] in LOC / person-month |
| Maximilien and Williams [174]|    | - 16% longer development*  
  - 18% more functional tests passed*     |
| George and Williams [87, 88] | 24 | - little or no difference in developer productivity  
  - required more time per assertion, more overall  
  and average development time ☺           |
| Geras et al. [89]            | 14 | - no evidence of more assertions or more assertions per method                              |
| Canfora et al. [40]          | 28 | - 15%(project B)–35%(project A) longer development time  
  - decreased defects/KLOC by 62%(project A)–76%  
  (project B)                                  |
| Bhat and Nagappan [28]       | 6 (A) | - 5–30% decrease in fault-slip-through rate*  
  - 60% decrease in avoidable fault costs*  
  - total project cost became less by 5–6%*  
  - the ratio of faults decreased by from 60–70%  
  (release 5) to 0–20% (release 7)*  
  - cost savings in maintenance are up to 25% of the  
  development cost*                           |
| Damm and Lundberg [56, 57]   | 100| - it took on average 15% or more* of overall time to write unit tests (“moderate perceived productivity losses”)  
  - reduced internal defect rate  
  - decreased defects rate by 40–90%  
  - 15–35% longer development time  
  - mean defect density reduced by 36%  
  - mean change density reduced by 76%       |
| Sanchez et al. [217]         | 9–17|                                                                                              |
| Nagappan et al. [192]        | 9.6, 5–8.7| -                                                                                         |
| Slyngstad [232]              |    |                                                                                              |
### Test-first programming studies (academic)

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

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

Source: Müller 05
- 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
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
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

For every complex problem there is an answer that is clear, simple, and wrong
H.L. Mencken

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

ANDROMAQUE: I do not understand abstractions.
CASSANDRA: As you like. Let us resort to metaphors.
Jean Giraudoux, The Trojan War Will Not Happen
Bibliography (1/4)

Agile Manifesto, at agilemanifesto.org/.


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


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


Ron Jeffries: *Xprogramming* site at [xprogramming.com](http://xprogramming.com).


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.


Mary &Tom Poppendieck: *Lean Software Development* site, [www.poppendieck.com](http://www.poppendieck.com).


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