ETH zürich

Agile software development

Bertrand Meyer

Part B: agile principles

1: The enemy: Big Upfront Anything

2: Organizational principles

3: More organizational principles

4: Technical principles

5: A few method-specific principles

Understanding agile

Values

Principles

- Managerial
- Technical

Roles

Practices

- Managerial
- Technical

Artifacts

Agile principles

Organizational

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

Technical

- **6** Develop iteratively
  - 6.1 Produce frequent working iterations
  - 6.2 Freeze requirements during 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

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.

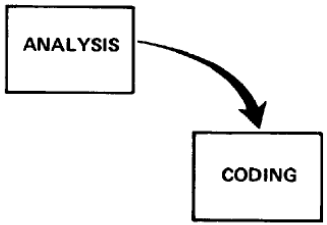


Figure 1. Implementation steps to deliver a small computer program for internal operations.

Proceedings of IEEE WESCON, pages 1-9, 1970

Waterfall (continued)

Source: Royce 1970

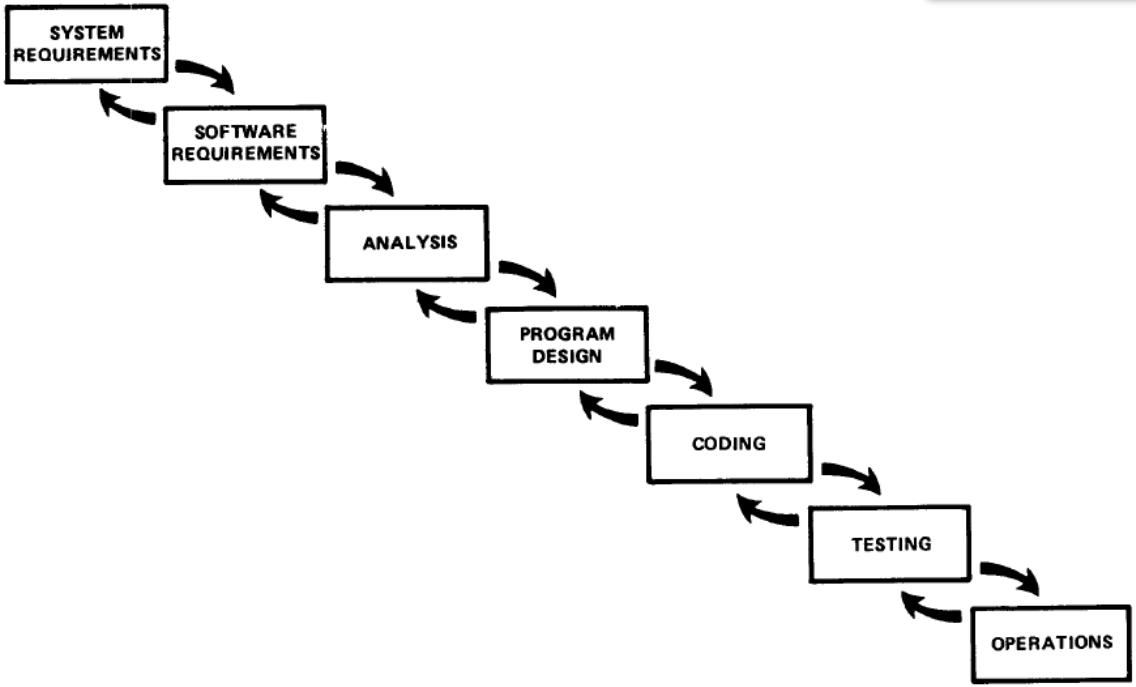
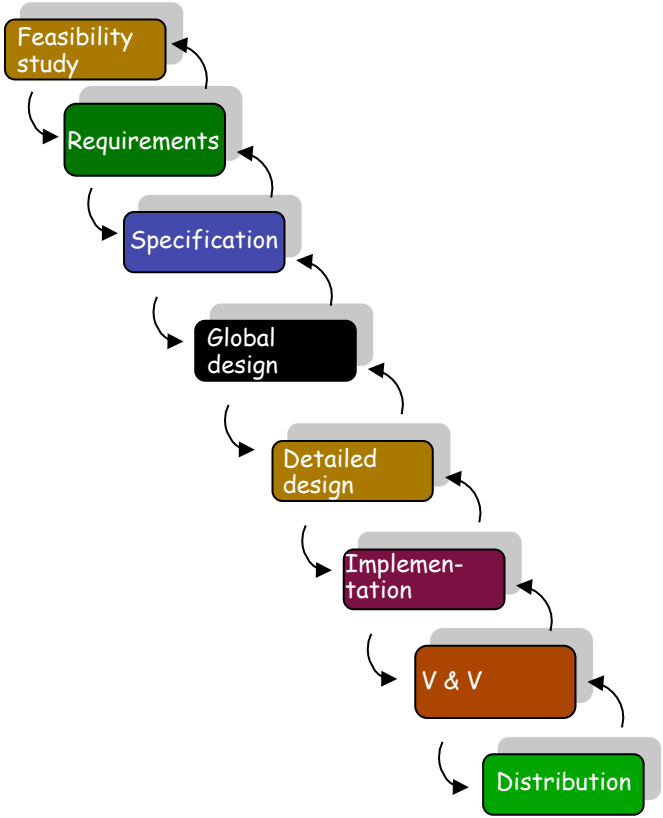


Figure 3. Hopefully, the iterative interaction between the various phases is confined to successive steps.

The waterfall model of the lifecycle



Arguments for the waterfall

Source: Boehm 81

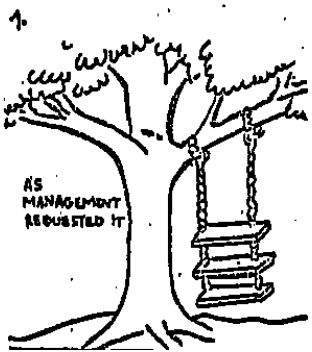
(After B.W. Boehm: *Software engineering economics*)

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

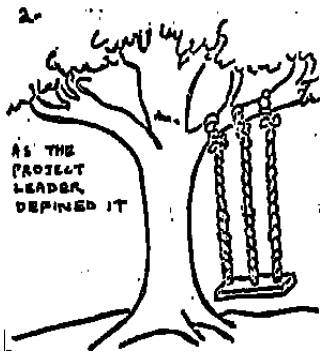
Problems with the waterfall

- 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

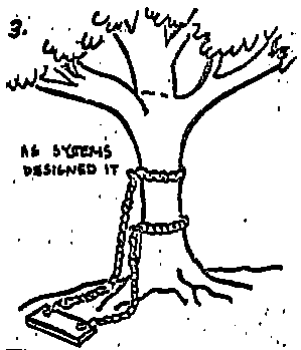
Impedance mismatches



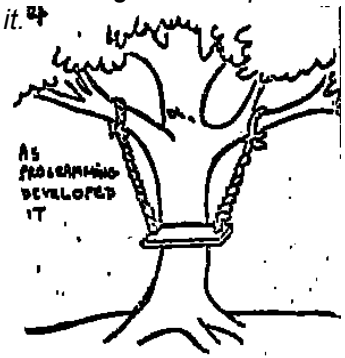
As Management requested it.



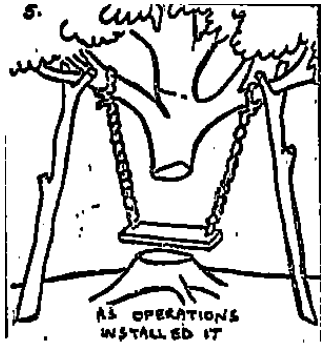
As the Project Leader defined it.



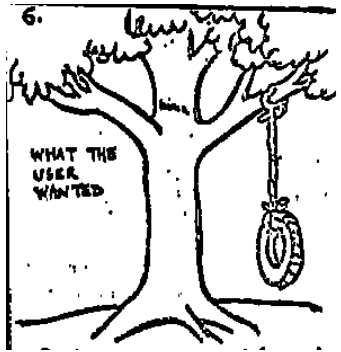
As Systems designed it.



As Programming developed it.



As Operations installed it.



What the user wanted.

(Pre-1970 cartoon; origin unknown)

Requirements

80% of interface fault and 20% of implementation faults due to requirements (Perry & Stieg, 1993)

48% to 67% of safety-related faults in NASA software systems due to misunderstood hardware interface specifications, of which 2/3rds are due to requirements (Lutz, 1993)

85% of defects due to requirements, of which: incorrect assumptions 49%, omitted requirements 29%, inconsistent requirements 13% (Young, 2001).

Numerous software bugs due to poor requirements, e.g. Mars Climate Orbiter

The two agile criticisms of requirements

Change criticism

Waste criticism

Beck: *Software development is full of the waste of overproduction, [such as] requirements documents that rapidly grow obsolete.*

Agile views of requirements

Beck (XP):

Requirements gathering isn't a phase that produces a static document, but an activity producing detail, just before it is needed, throughout development

Cohn (Scrum):

Scrum projects do not have an upfront analysis or design phase; all work occurs within the repeated cycle of sprints

Poppendieck (Lean):

And those things called requirements? They are really candidate solutions; separating requirements from implementation is just another form of handover

Agile Software Development

Bertrand Meyer

Part B: Principles

1: The Enemy: Big Upfront Anything

What we have seen:

The waterfall model: a useful foil

The role of requirements

The risk of applying extreme precepts literally

ETH zürich

Agile software development

Bertrand Meyer

Part B: agile principles

1: The enemy: Big Upfront Anything

2: Organizational principles

3: More organizational principles

4: Technical principles

5: A few method-specific principles

Agile principles

Organizational

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

Technical

- **6** Develop iteratively
  - 6.1 Produce frequent working iterations
  - 7.2 Freeze requirements during 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

1 Put the customer at the center

Beck: *You will get [better] results with real customers. They are who you are trying to please. No customer at all, or a "proxy" for a real customer, leads to waste as you develop features that aren't used, specify tests that don't reflect the real acceptance criteria, and lose the chance to build real relationships between the people with the most diverse perspective of the project.*

XP: embedded customer; Scrum: product owner

Can customer involvement replace requirements?

2 Accept change

Agile manifesto: “welcome” change

In standard software engineering, especially object-oriented: extendibility

Poppendieck: *While **in theory** OO development produces code that is easy to change, in practice OO systems can be as difficult to change as any other, especially when information hiding is not deeply understood and effectively used*

3 Let the team self-organize

Traditional view: managers tell workers to do their job
Agile view: managers listen to developers, explain possible actions, provide suggestions for improvements.

"The leader is there to:

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

Team chooses own commitments & has access to customers

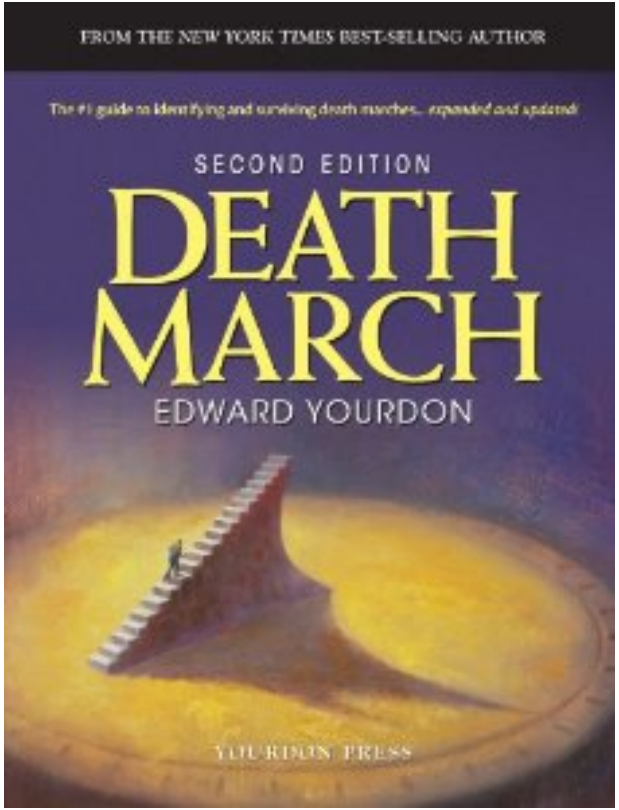
Self-organizing team: I Musici

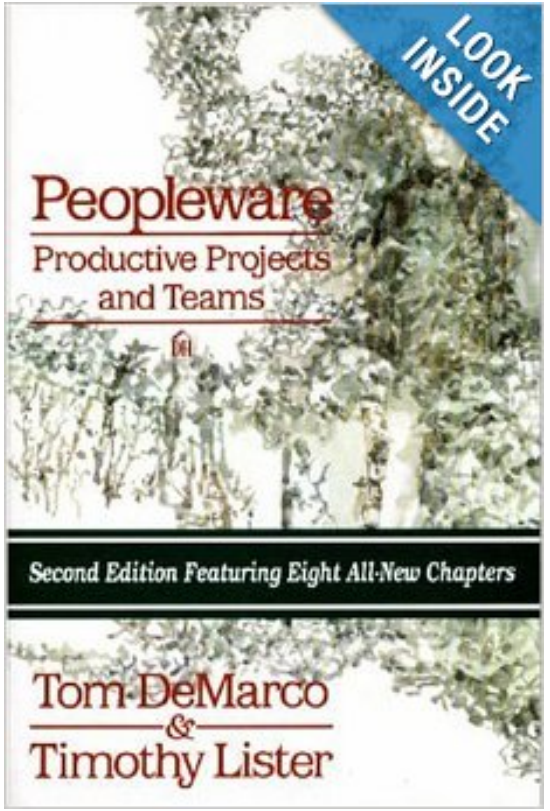
Let the team self-organize

(Blog poster) *The most important aspect of these methods is to put the management of the project squarely where it belongs: on the backs of the people doing the work. When the people actually doing the work have the final say in what gets done and when, then projects actually get done on time.*

But (Schwaber): *Control through peer pressure and "control by love" are the basis of subtle control. The dynamic flow of the team surfaces the tacit (unconscious) knowledge of the group and creates explicit knowledge in the form of software.*

4 Maintain a sustainable pace





Maintaining a sustainable pace

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

Agile Software Development

Bertrand Meyer

Part B: Principles
2: Organizational principles

What we have seen:

Principles:

Put the customer at the center

Accept change

Let the team self-organize

Maintain a sustainable pace

More to come…

ETH zürich

Agile software development

Bertrand Meyer

Part B: agile principles

1: The enemy: Big Upfront Anything

2: Organizational principles

3: More organizational principles

4: Technical principles

5: A few method-specific principles

Agile principles

Organizational

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

Technical

- **6** Develop iteratively
  - 6.1 Produce frequent working iterations
  - 7.2 Freeze requirements during 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

5 Develop minimal software

Minimalism:

- Minimal functionality
- Product only
- Only code and tests

5.1 Develop minimal software: functionality

YAGNI (Jeffries): [this principle] *reminds us always to work on the story we have, not something we think we're going to need. Even if we know we're going to need it.*

Poppendieck: *Our software systems contain far more features than are ever going to be used. Extra features increase the complexity of the code, driving up costs nonlinearly. If even half of our code is unnecessary — a conservative estimate — the cost is not just double; it's perhaps ten times more expensive than it needs to be.*

The “lean” view

Source: Poppendieck

Seven wastes of software development:

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

5.2 Develop minimal software: product only

Cunningham:

- *You are always taught to do as much as you can. Always put checks in. Always look for exceptions. Always handle the most general case. Always give the user the best advice. Always print a meaningful error message. Always this. Always that. You have so many things in the background that you're supposed to do, there's no room left to think. I say, forget all that and ask yourself,* ***What's the simplest thing that could possibly work?***

About reuse…

Jeffries: *Unless the projects are being done by the same team, reuse is quite difficult to do effectively: there is a big difference between some part of the project that I can reuse, and packaging that part well enough so that others can do so. I have to do packaging work that I wouldn't do for myself, to document it, to make it more bulletproof, removing issues that I just work around automatically, to support it, answer questions about it, train people in how to use it. If I do those things, it's expensive. If I don't, using my stuff is difficult for others and doesn't help them much.*

I build the abstractions I need. If I need an abstraction again, in a different context, I would improve it. But unless my project's purpose is to build stuff for other projects, I try not to waste any of my time and money building for other projects.

5.3 Develop minimal software: code & tests

Cockburn: *You get no credit for any item that does not result in **running, tested code**. Okay, you also get credit for **final deliverables** such as training materials and delivery documentation.*

Poppendieck: *The documents, diagrams, and models produced as part of a software development project are often consumables, aids used to produce the system, but not necessarily a part of the final product. Once a working system is delivered, the user may care little about the intermediate consumables. Lean principles suggest that every consumable is a candidate for scrutiny. The burden is on the artifact to prove not only that it adds value to the final product, but also that it is the most efficient way of achieving that value.*

Agile Software Development

Bertrand Meyer

Part B: Principles
1: The Enemy: Big Upfront Anything

What we have seen:

Minimality in various guises:
minimal functionality,
product only,
code and tests only

ETH zürich

Agile software development

Bertrand Meyer

Part B: agile principles

1: The enemy: Big Upfront Anything

2: Organizational principles

3: More organizational principles

4: Technical principles

5: A few method-specific principles

Agile principles

Organizational

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

Technical

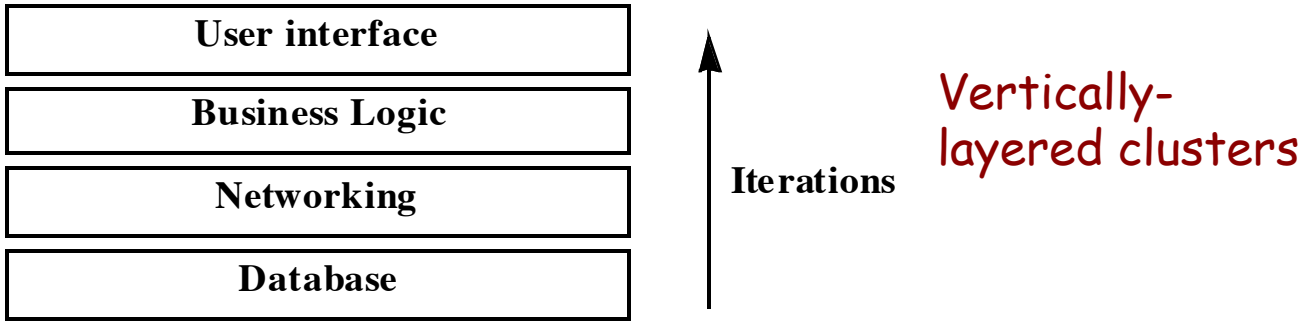
- **6** Develop iteratively
  - 6.1 Produce frequent working iterations
  - 7.2 Freeze requirements during iterations
- **7** Treat tests as a key resource:
  - 7.1 Do not start any new development until all tests pass
  - 7.2 Test first
- **8** Express requirements through scenarios

6 Develop iteratively

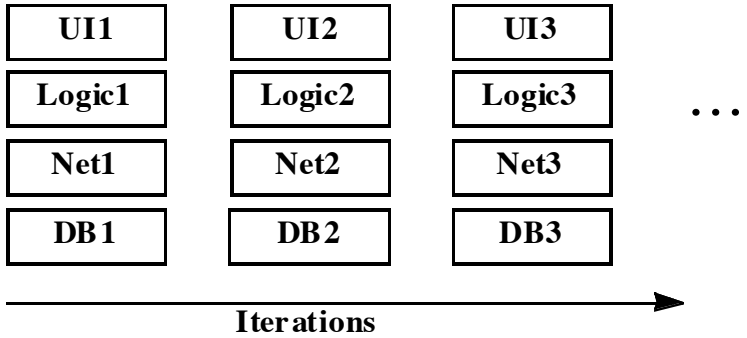
Iterativeness:

- 6.1 Frequent working iterations
- 6.2 Freeze requirements during iteration (Closed-Window Rule)

6.1 Iterativeness: frequent working iterations



Horizontally-layered clusters



6.2 Iterativeness: freeze requirements during iteration

The Closed-Window Rule:

During an iteration, no one may add functionality

(or: the sprint is cancelled)

Dual development

Early on: build infrastructure

Later: produce releases

7.1 Tests: do not move on until all tests pass

Excellent principle, to be reconciled with practical constraints

Need to classify severity

7.2 Test first

See discussion of practices

8 Express requirements through scenarios

User stories

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*>"

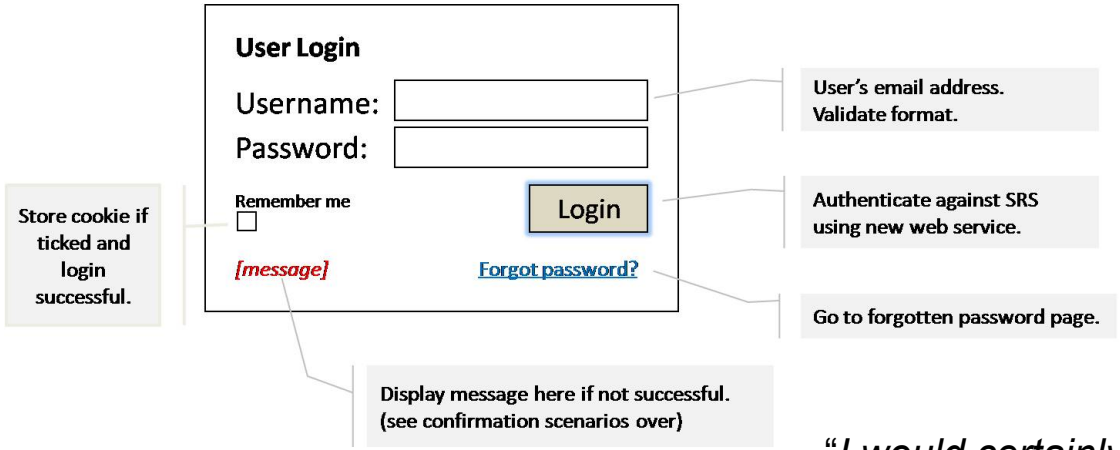
Example:

"***As a*** *customer,*
I want to *see a list of my recent orders,*
so that *I can track my purchases with a company."*

Example user story

#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.

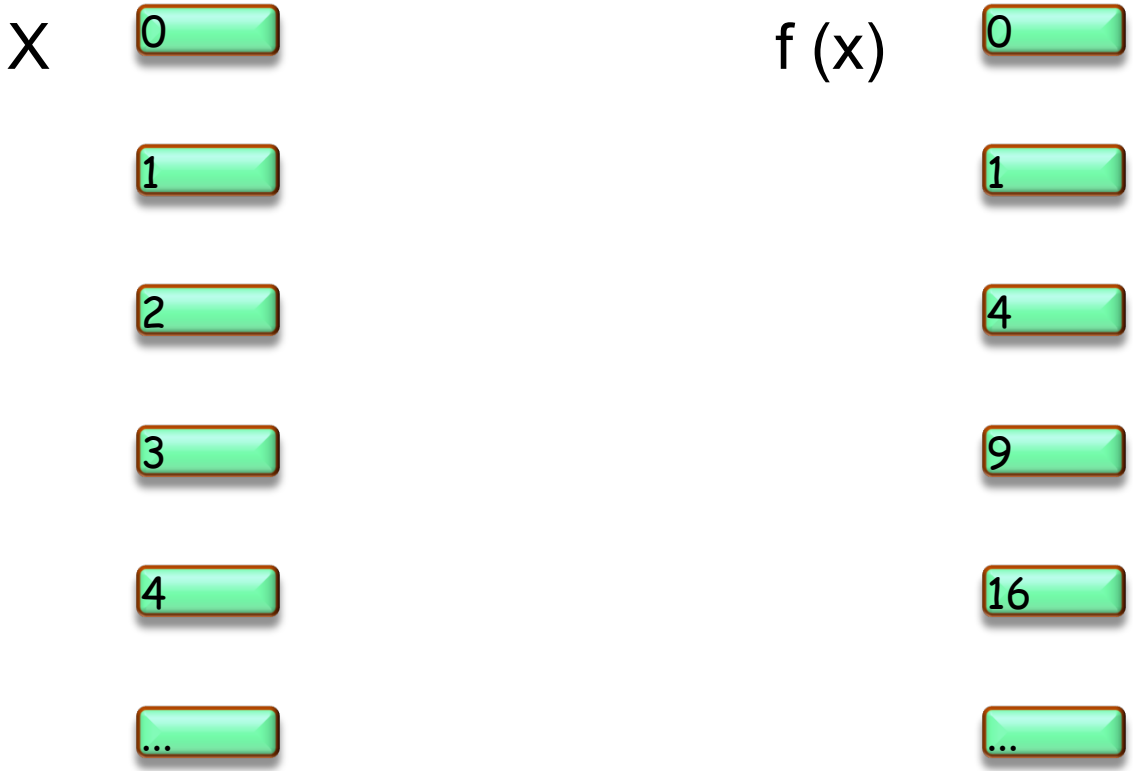


Further information is attached to this story on VSTS Product Backlog.

Source: Waters

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

User stories



See https://bertrandmeyer.com/2012/10/14/a-fundamental-duality-of-software-engineering/

or http://tinyurl.com/qzyxlal

Analysis

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

User stories are to requirements what tests are to software specification and design

Major application: for validating requirements

Agile Software Development

Bertrand Meyer

Part B: Principles
5: Technical principles

What we have seen:

Iterative development

The fundamental role of tests (more to come)

User stories as the source of requirements…

… and the limits of this approach

ETH zürich

Agile software development

Bertrand Meyer

Part B: agile principles

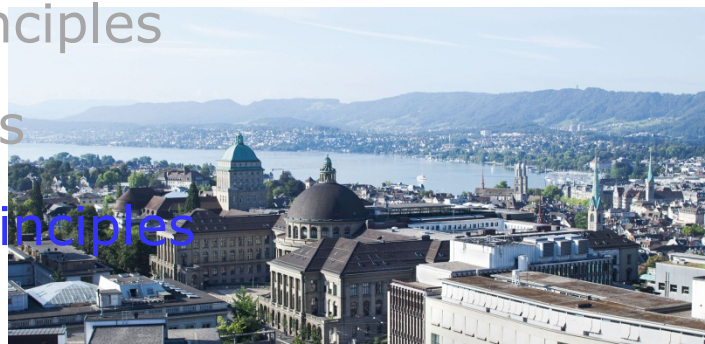
1: The enemy: Big Upfront Anything

2: Organizational principles

3: More organizational principles

4: Technical principles

5: A few method-specific principles



Agile principles

Organizational

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

Technical

- **6** Develop iteratively
  - 6.1 Produce frequent working iterations
  - 7.2 Freeze requirements during 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

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

Amplify learning

Lean

Source: Poppendieck

Software development is a continuous learning process

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

- 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

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

Focus

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

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

Multiple design

Lean

Source: Poppendieck

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

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)

Additive and multiplicative complexity

Adding features

Source: Zave*

In telecommunication software, feature interactions are a notorious source of runaway complexity, bugs, cost and schedule overruns, and unfortunate user experiences.

Bob has "call-forwarding" enabled and is forwarding calls to Carol. Carol has "do-not-disturb". Alice calls Bob, the call is forwarded to Carol, and Carol's phone rings, because "do-not-disturb" is not applied to a forwarded call.

Alice calls a sales group. A feature for the sales group selects Bob as a sales representative on duty, and forwards the call to Bob. Bob's cellphone is turned off, so his personal Voice Mail answers the call and offers to take a message.

[etc.]

*Abridged, see full citation in book

User stories (imagined)

(#1) **As an** executive, **I want** a redirection option **so that** if my phone is busy the call is redirected to my secretary

…

(#2) As a system configurator, I want to be able to specify various priorities for “busy” actions

..

(#3) As a salesperson, I want to make sure that if a prospect calls while I am in a conversation, the conversation is interrupted **so that** I can take the call immediately

…

(#4) As a considerate correspondent, I want to make sure that if a call comes while my phone is busy I get to the option of calling back as soon as the current call is over

Personal safety

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)

Agile Software Development

Bertrand Meyer

Part A: Context
6: A few method-specific principles

What we have seen:

Important ideas from Scrum, Lean, Crystal and XP
Minimize waste
Accumulate user stories (and the limits of that approach)
Amplify learning
Ensure personal safety
(and a few others)