# COM3005J: Agile Processes
## Chapter 1 : **Context**
## **Agile Principles**

Dr. Anca Jurcut
E-mail: `anca.jurcut@ucd.ie`

School of Computer Science and Informatics
University College Dublin

Beijing-Dublin International College

## 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
  - 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

## Lifecycle Models

Origin: Royce, 1970, Waterfall model

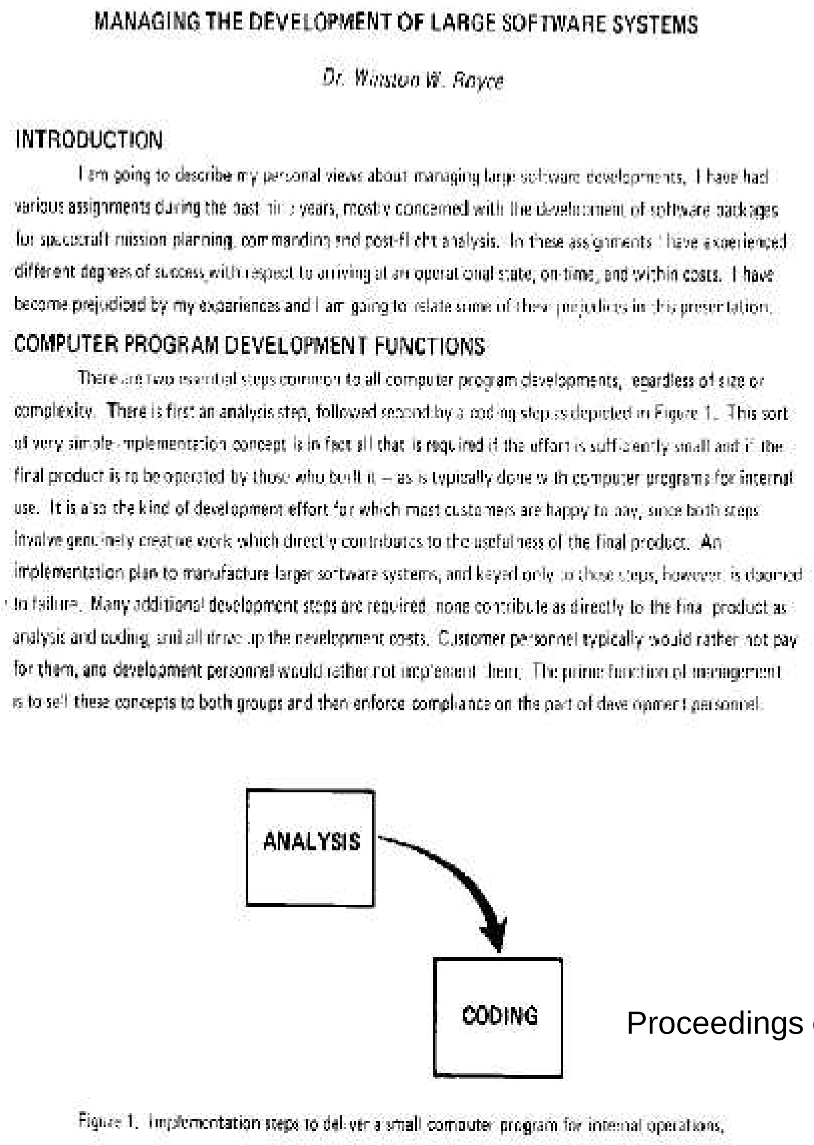
Scope: describe the set of processes involved in the production of software systems, and their sequencing

“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.

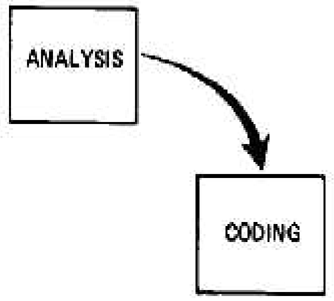


Figure 1. Implementation steps to deliver a small computer program for internal operations.

Proceedings of IEEE WESCON, pages 1-9, 1970

## Waterfall (continued)

Source: Royce 1970

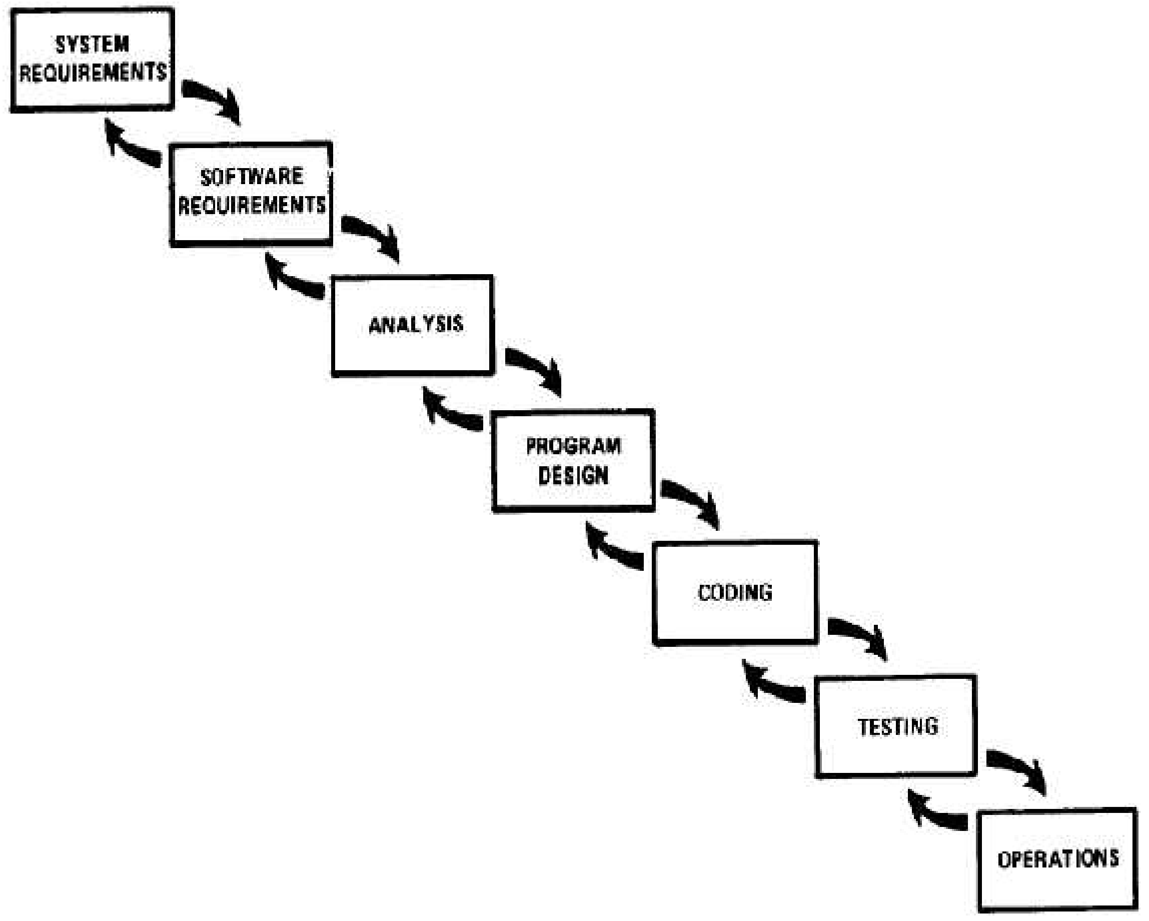
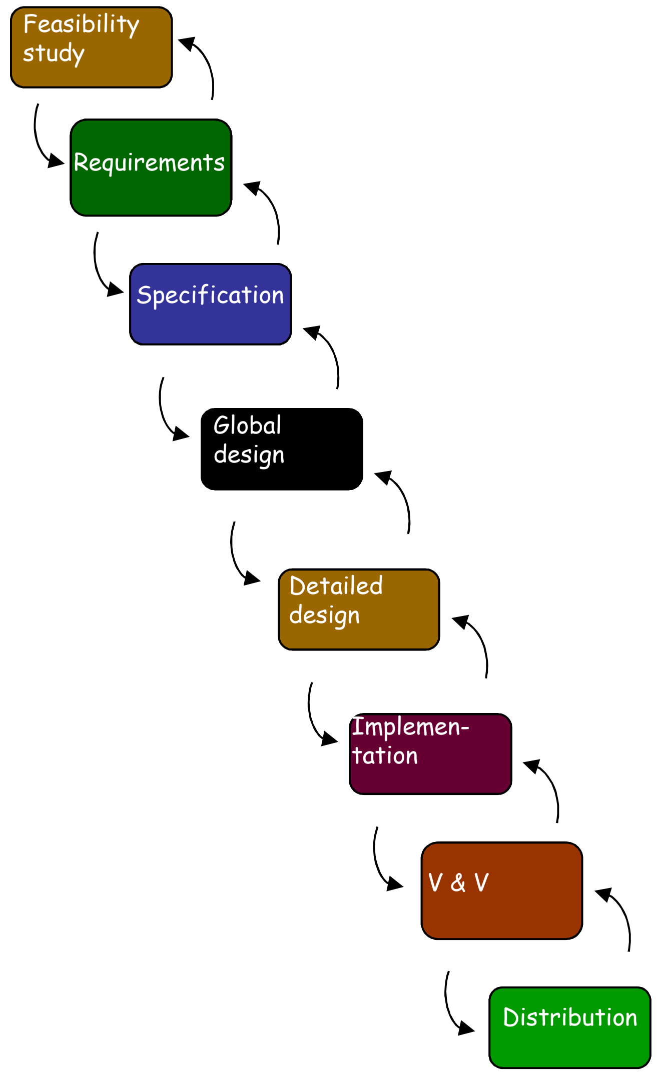


Figure 3. Hopefully, the iterative interaction between the various phases is confined to successive steps.

# The Waterfall Model of the Lifecycle



## Arguments for the Waterfall

Source: Boehm 81

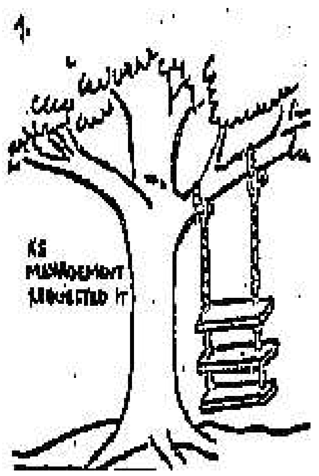
(After B.W. Boehm: *Software engineering economics*)

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

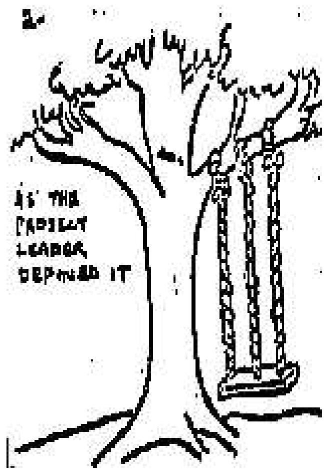
## Problems with the Waterfall

- 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

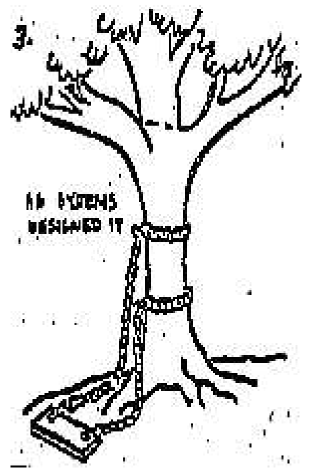
# Impedance Mismatches



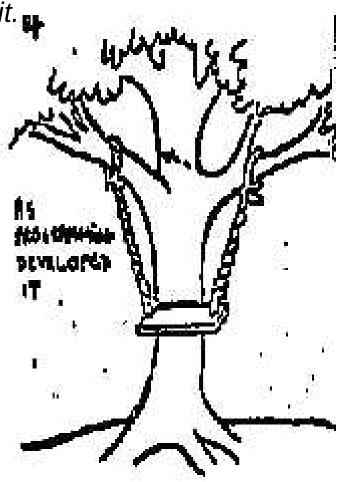
*As Management requested it.*



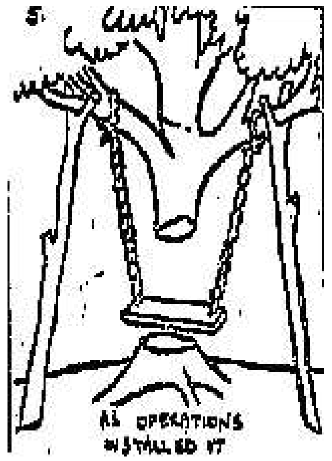
*As the Project Leader defined it.*



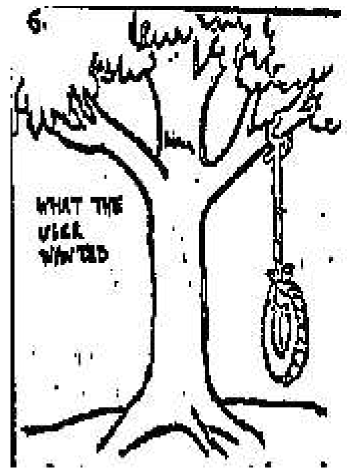
*As Systems designed it.*



*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*

## 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

# 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

Lean XP Scrum

Source: Poppendieck

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

"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

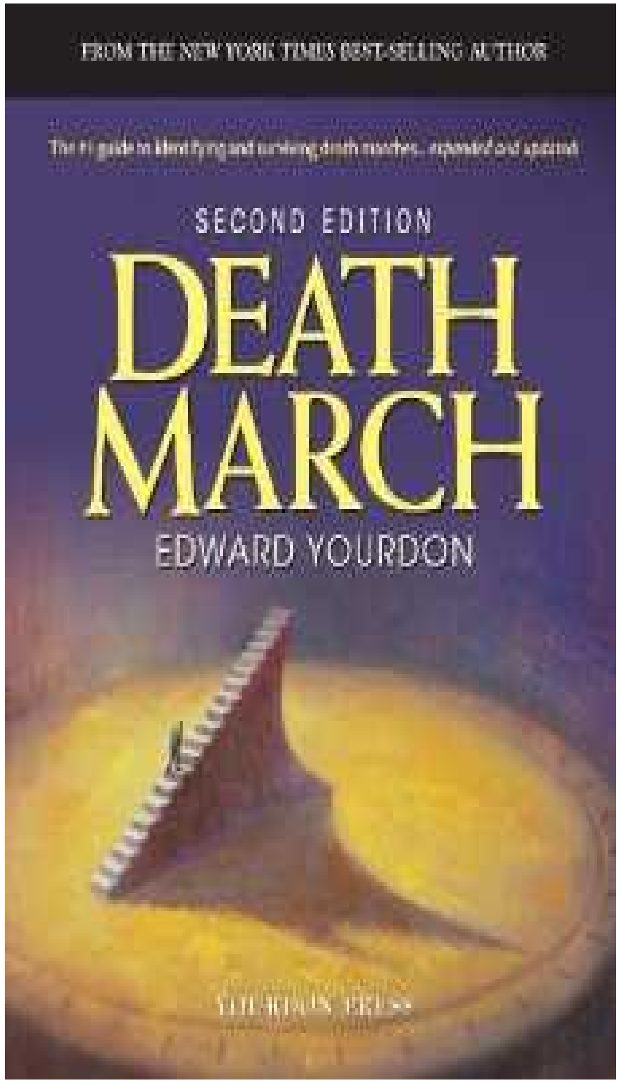
## Self-organizing team: I Musici (2)

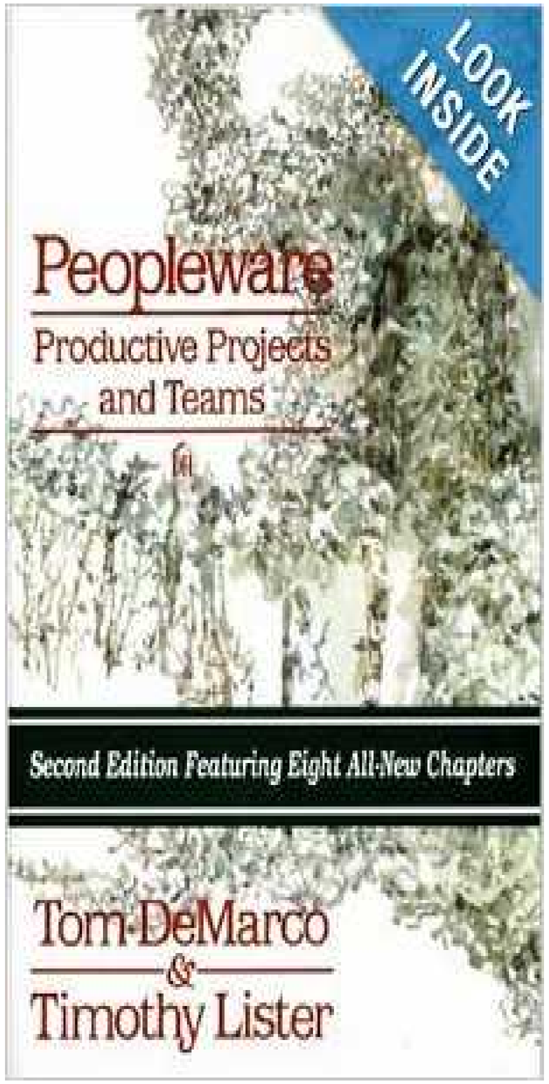
## Let the team self-organize (3)

(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




LOOK INSIDE
Peopleware
Productive Projects and Teams
Second Edition Featuring Eight All-New Chapters
Tom DeMarco & Timothy Lister

## Maintaining a sustainable pace (2)

- 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

## Principles

1. ..
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…

# 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: *Y**ou 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.*

## Principles

1. ..
2. ..
3. More Organizational Principles

**What we have seen:**

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