# Robotics Programming Laboratory

Bertrand Meyer
Jiwon Shin

## Lecture 1:

## Introduction to robotics

## Introduction to software engineering

# Objectives

After completing this laboratory course, you will understand:

- Basic software engineering principles and methods
- Most common architectures in robotics
- Coordination and synchronization methods
- How software engineering applies to robotics

and have gained experience in programming a small robotics system

# Practical details

Lecturers

- Prof. Dr. Bertrand Meyer
- Dr. Jiwon Shin

Assistants

- Andrey Rusakov
- Ganesh Ramanathan

Course page

- http://se.inf.ethz.ch/courses/2015b_fall/rpl

Forum

- https://piazza.com/class/idbqs3jsxfn6zn

# Practical details

Schedule

- Monday, 16:15 – 18:00, WEH D 4
- Thursday, 15:15 – 17:00, WEH D 4

This is a hands-on laboratory class. You will develop software for your own robot. Lectures and exercise sessions will be much more interactive than in traditional courses.

Your fellow classmates are your best resources. We encourage you to talk to each other and help each other. For online communication, use the forum to post your questions and answer questions other have.

# Practical details

Laboratory space

- WEH D 4 is open exclusively to you.
- In a week, you can pick up keys to the building and to the room.
- Please **lock the room** when you leave and **close the main door** when you enter and leave.
- Please **keep the space tidy!**

Hardware

- Starting next Monday, you can get a robot, a sensor, and some cables to be used for the class.
- We ask you to deposit 50 CHF for the hardware. You will get the money back when you return the hardware.
- We expect you to have a laptop. If you do not have one, please contact us.

# Grading

The grade for this laboratory course is based **entirely on the project**.

Every assignment has an individual component (50%) and a group component (50%). For the group portion, you may work in a group of 2 to 3 people.

You must submit your work at every evaluation point and participate in the final competition to receive a grade for this class. You must pass both individual component and group component to pass this course.

- Assignment 1 (8 Oct/19 Oct): control and obstacle avoidance
- Assignment 2 (3 Nov/9 Nov): path planning
- Assignment 3 (23 Nov/30 Nov): object recognition
- Final competition (7 Dec/17 Dec): search and rescue

# Project grading

In-class Demonstration: 50%

- Precise evaluation criteria will be defined at the beginning of each phase

Software Quality: 50%

- Choice of abstractions and relations
- Correctness of implementation
- Extendibility and reusability
- Comments and documentation, including ”README”

# Topics

Control and obstacle avoidance

- ROS and Roboscoop, Modern software engineering tools SCOOP, Robot control and obstacle avoidance, Design patterns

Path planning

- Path planning

Object recognition

- Robot perception

Search and rescue

- Localization, Software architecture in robotics

# Lecture schedule

1. Introduction to robotics and software engineering
2. ROS and Roboscoop
3. Control / Modern software engineering tools
4. SCOOP
5. Obstacle avoidance
6. Design patterns

    Assignment 1: control and obstacle avoidance

7. Path planning
8. Robot perception

    Assignment 2: path planning

9. Localization
10. Software architecture for robotics

    Assignment 3: Object recognition

    Assignment 4: Search and rescue

# Recommended literature

## Software engineering

- *Object-Oriented Software Construction*, Meyer
- *Design Patterns*, Gamma, Helm, Johnson, Vlissides
- *Pattern-Oriented Software Architecture: Volume 2*, Schmidt, Stal, Rohnert, Buschmann

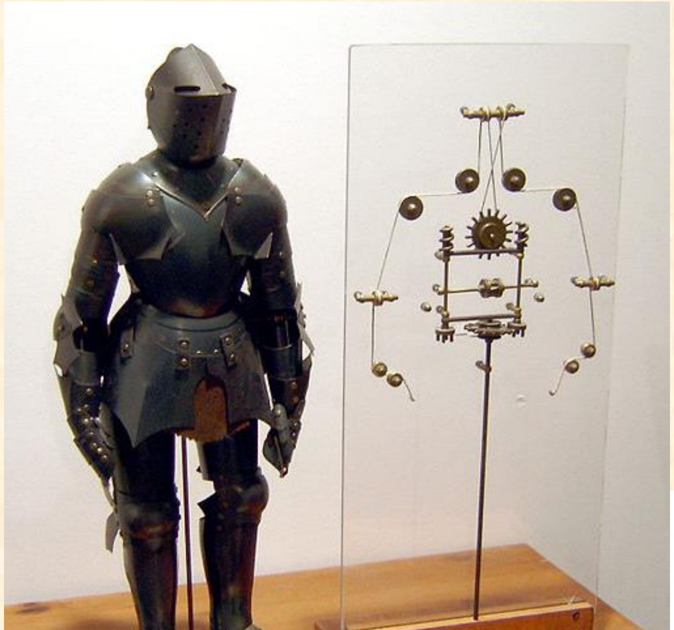
## Robotics

- *Probabilistic Robotics*, Thrun, Burgard, Fox
- *Introduction to Autonomous Mobile Robots*, by Siegwart, Nourbakhsh, Scaramuzza

## Programming language

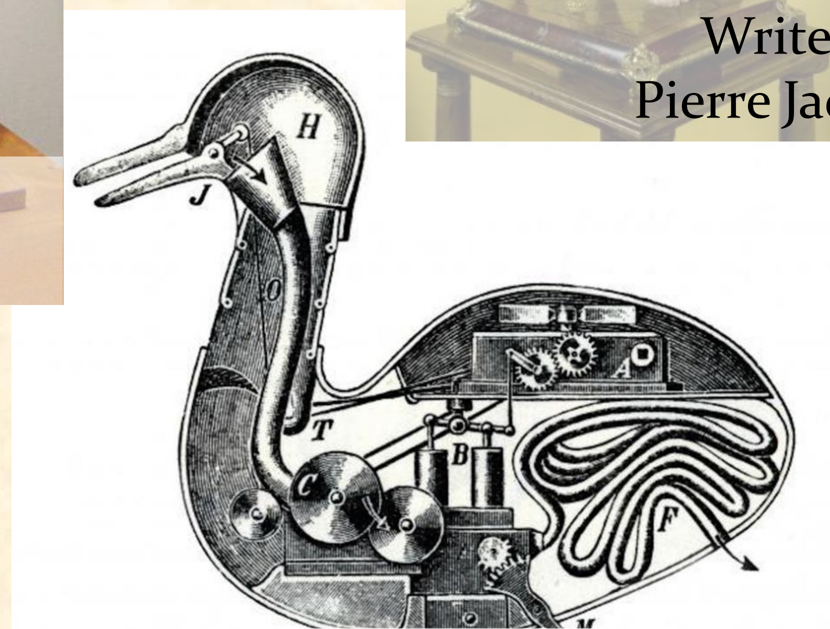
- *Touch of Class*, Meyer
- *The C++ Programming Language*, Stroustrup

# Robots as automata



Robot knight (1495)
Leonardo da Vinci

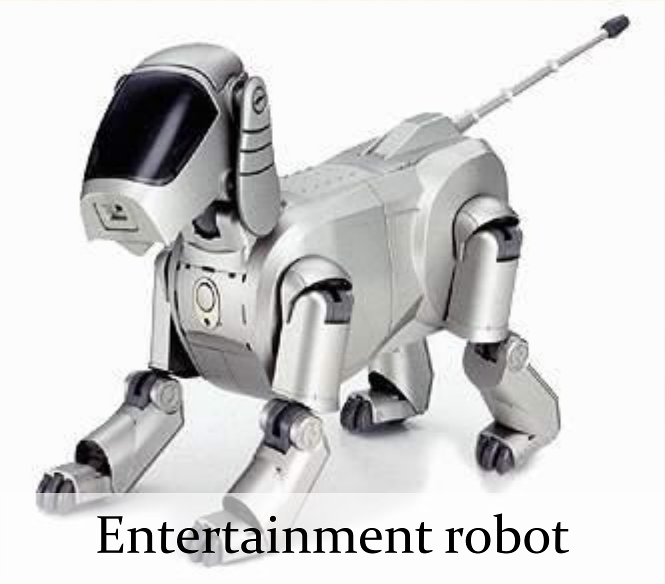
Writer (1774)
Pierre Jaquet-Droz

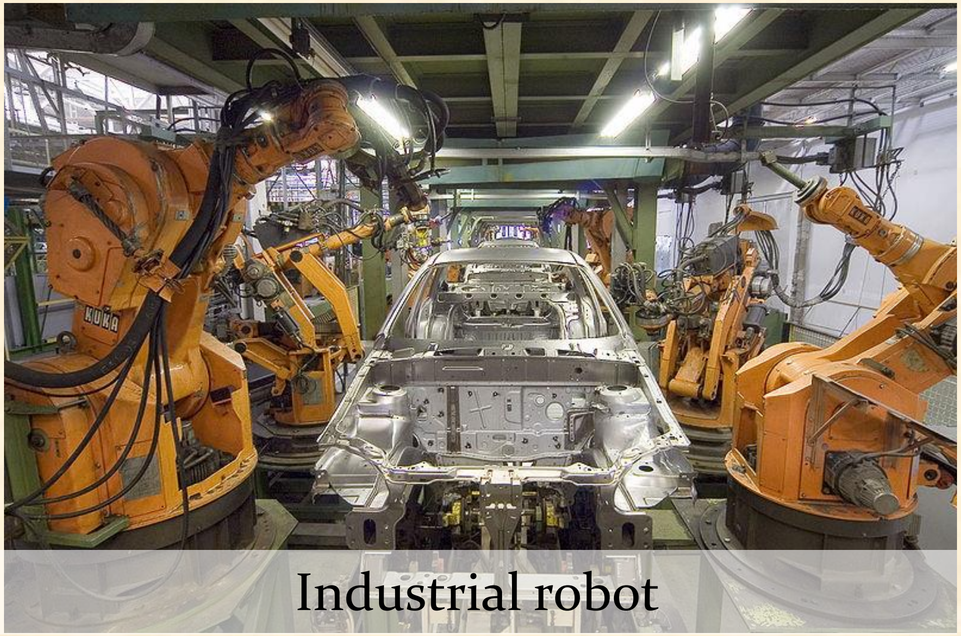


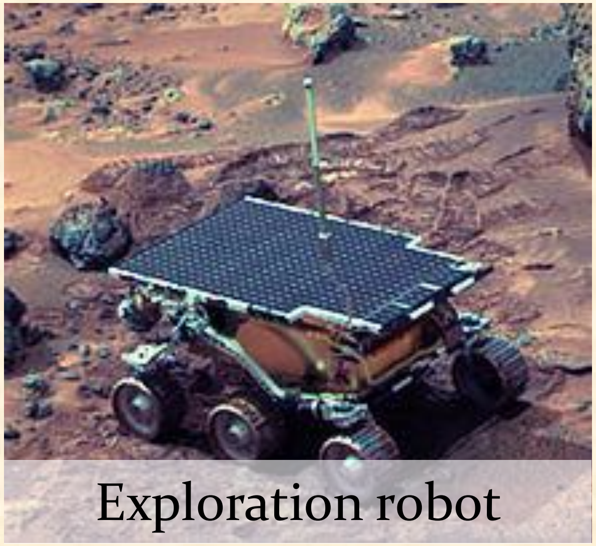

Digesting duck (1738)
Jacques de Vaucanson

# Robots of the 20th century


Surveillance robot


Entertainment robot


Industrial robot


Exploration robot

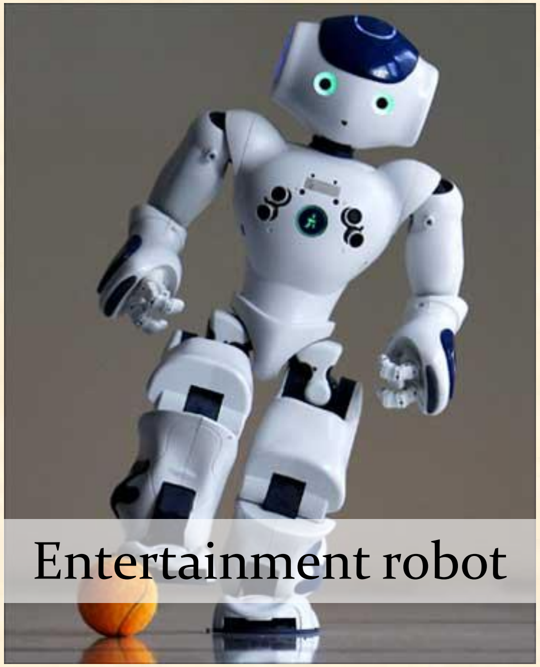
# Robots of today


Exploration robot


Autonomous vehicle


Industrial robot


Surveillance robot


Entertainment robot


Service robot

# Robotics

**Robot**: A machine capable of carrying out a complex series of actions automatically, especially one programmable by a computer

**Robotics**: The branch of technology that deals with the design, construction, operation, and application of robots – Oxford dictionary

**Components of robotics**

- Perception: vision, touch, range, sound
- Actuation: manipulation, locomotion
- Cognition: navigation, recognition, planning, interaction

# Challenges in robotics: uncertainty!

Solved challenges

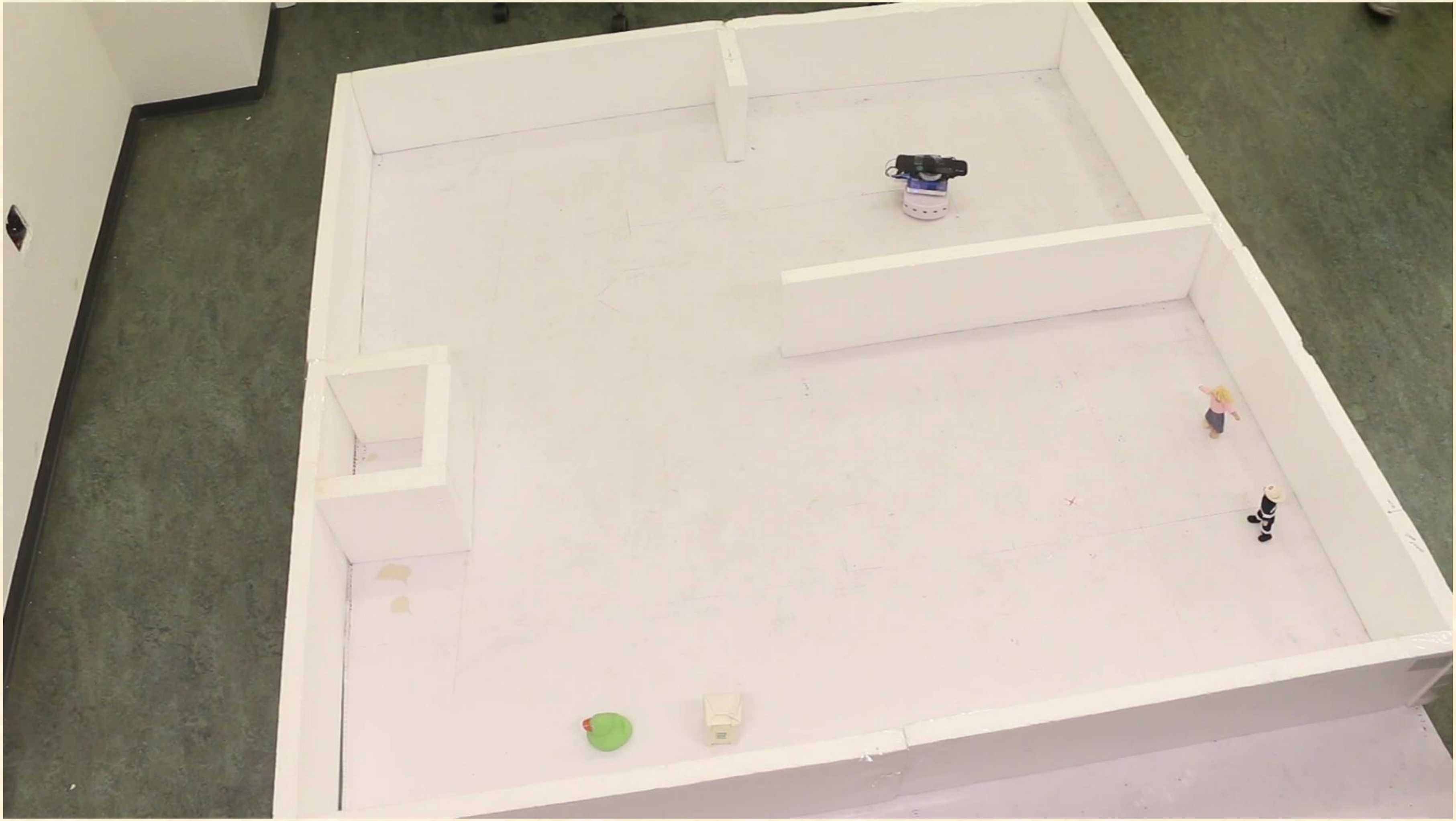
- Navigation in static environment – Clausiusstrasse
- Recognition of known objects – face, simple objects
- Manipulation of simple, rigid objects – beer fetching

Open challenges

- Navigation in dynamic environment – Bahnhofstrasse
- Scene understanding – a group of people at a party
- Manipulation of complex, deformable objects – laundry folding
- Learning over time and knowledge transfer

# Robot for the class

# What people did last year



# Introduction to software engineering

(and software architecture)

# Software engineering

"The application of engineering to software"

Engineering (Wikipedia): "the discipline, art and profession of acquiring and applying technical, scientific, and mathematical knowledge to design and implement materials, structures, machines, devices, systems, and processes that safely realize a desired objective or invention"

A simpler definition of engineering: the application of scientific principles to the construction of artifacts

# For this course

The application of engineering principles and techniques, based on mathematics, to the development and operation of possibly large software systems satisfying defined standards of quality

# Parnas's view

(Cited in Ghezzi et al.)

"The multi-person construction of multiversion software"

# “Large” software systems

What may be large: any or all of

- Source size (lines of code, LoC)
- Binary size
- Number of users
- Number of developers
- Life of the project (decades…)
- Number of changes, of versions

(Remember Parnas’s definition)

# Process and product

Software engineering affects both:

- Software **products**
- The **processes** used to obtain and operate them

Products are not limited to code. Other examples include requirements, design, documentation, test plans, test results, bug reports

Processes exists whether they are formalized or not

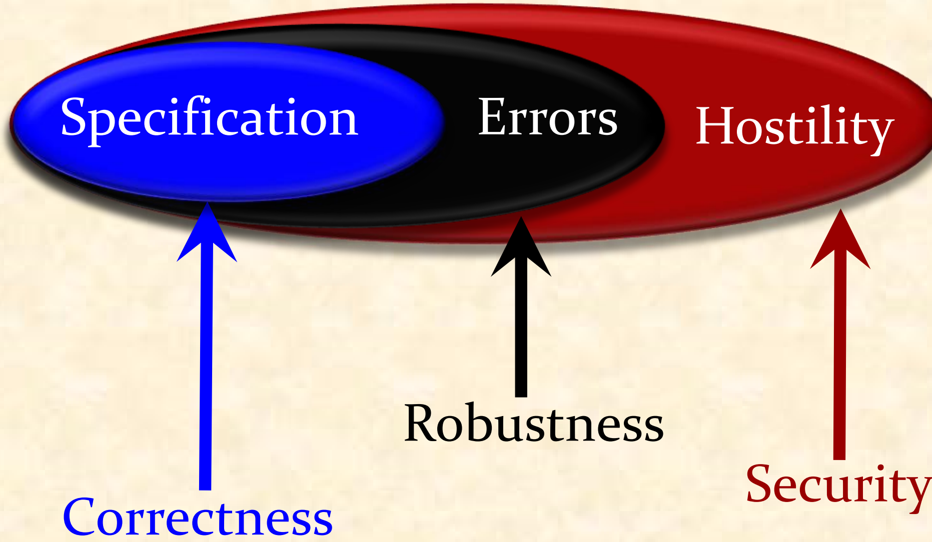
# Software quality factors

## Product

### Immediate

- Correctness
- Robustness
- Security
- Ease of use
- Ease of learning
- Efficiency

"Reliability"




### Long-term

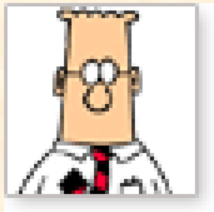
- Extendibility
- Reusability
- Portability

## Process

- Timeliness
- Cost-effectiveness
- Predictability
- Reproducibility
- Self-improvement

# Software engineering today

Three cultures:

- Process 
- Agile 
- Object

The first two are usually seen as exclusive, but all have major contributions to make.

# The process culture

Emphasize:

- Plans
- Schedules
- Documents
- Requirements
- Specifications
- Order of tasks
- Commitments

Examples: Rational Unified Process, CMMI, Waterfall…

# CMMI basic ideas

CMMI is a catalog of approved practices and goals

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

Focused on process, not technology

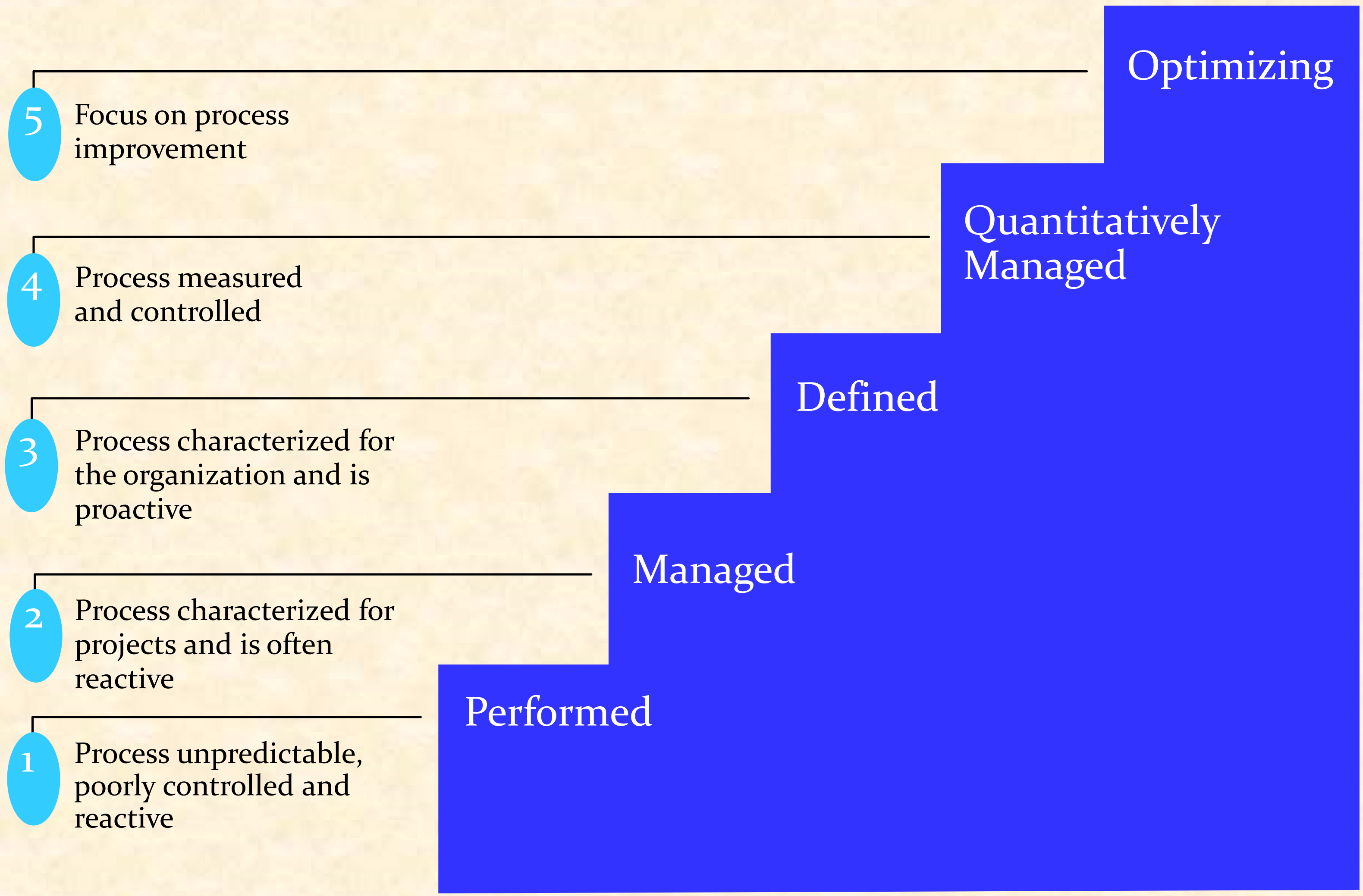
Emphasizes **reproducibility** of results

(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

# CMMI maturity levels



# Agile

Examples: Extreme Programming (XP), Scrum

Emphasizes:

- Short iterations
- Emphasis on working code
- Emphasis on testing
- De-emphasis of plans and documents
- De-emphasis of upfront specifications and design
- Communication: customer involvement
- Specific practices, e.g. Pair Programming

# 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 disciplined change
  - 6.1 Do not change requirements during an iteration
- **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

# Six task groups of software engineering

Describe — Requirements, documentation ...

Implement — Design, programming

Assess — V&V*, esp. testing

Manage — Plans, schedules, communication, reviews...

Operate — Deployment, installation

Notate — Languages for programming etc.

***Validation & Verification**

# Software lifecycle models

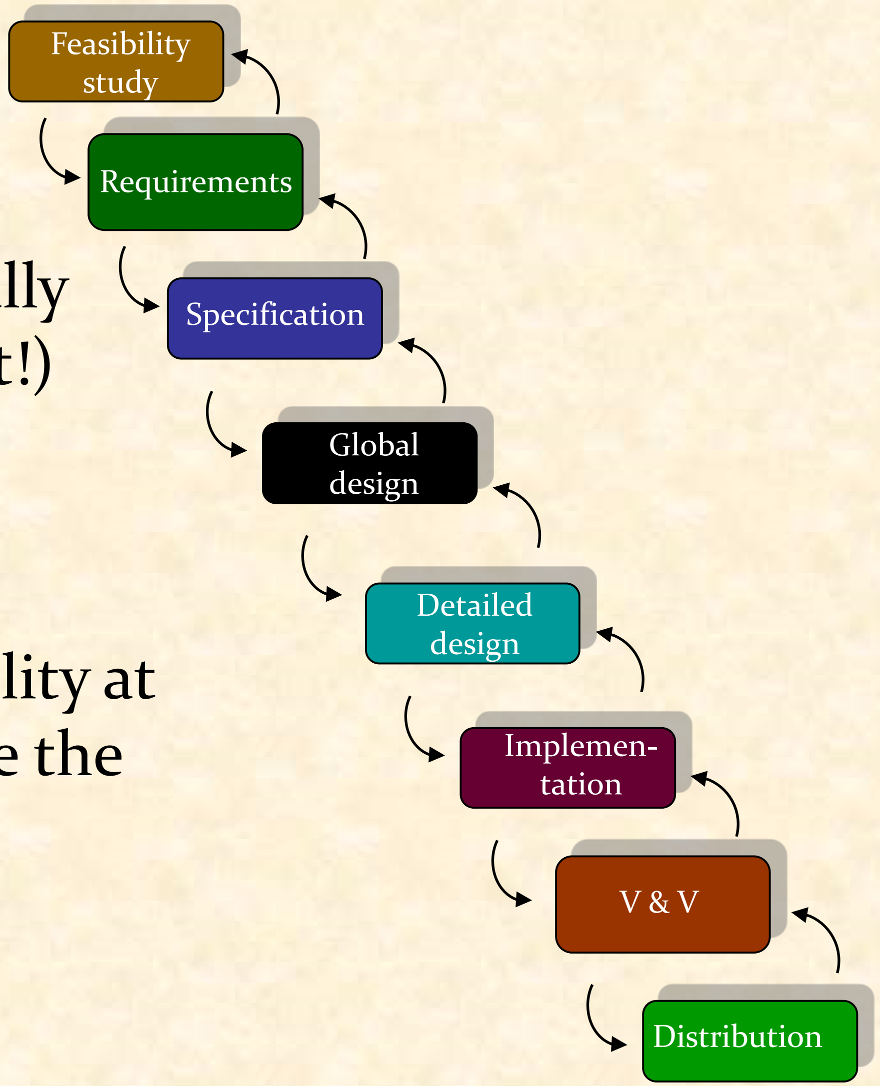
Describe an overall distribution of the software construction into tasks, and the ordering of these tasks

They are models in two ways:

- Provide an abstracted version of reality
- Describe an ideal scheme, not always followed in practice

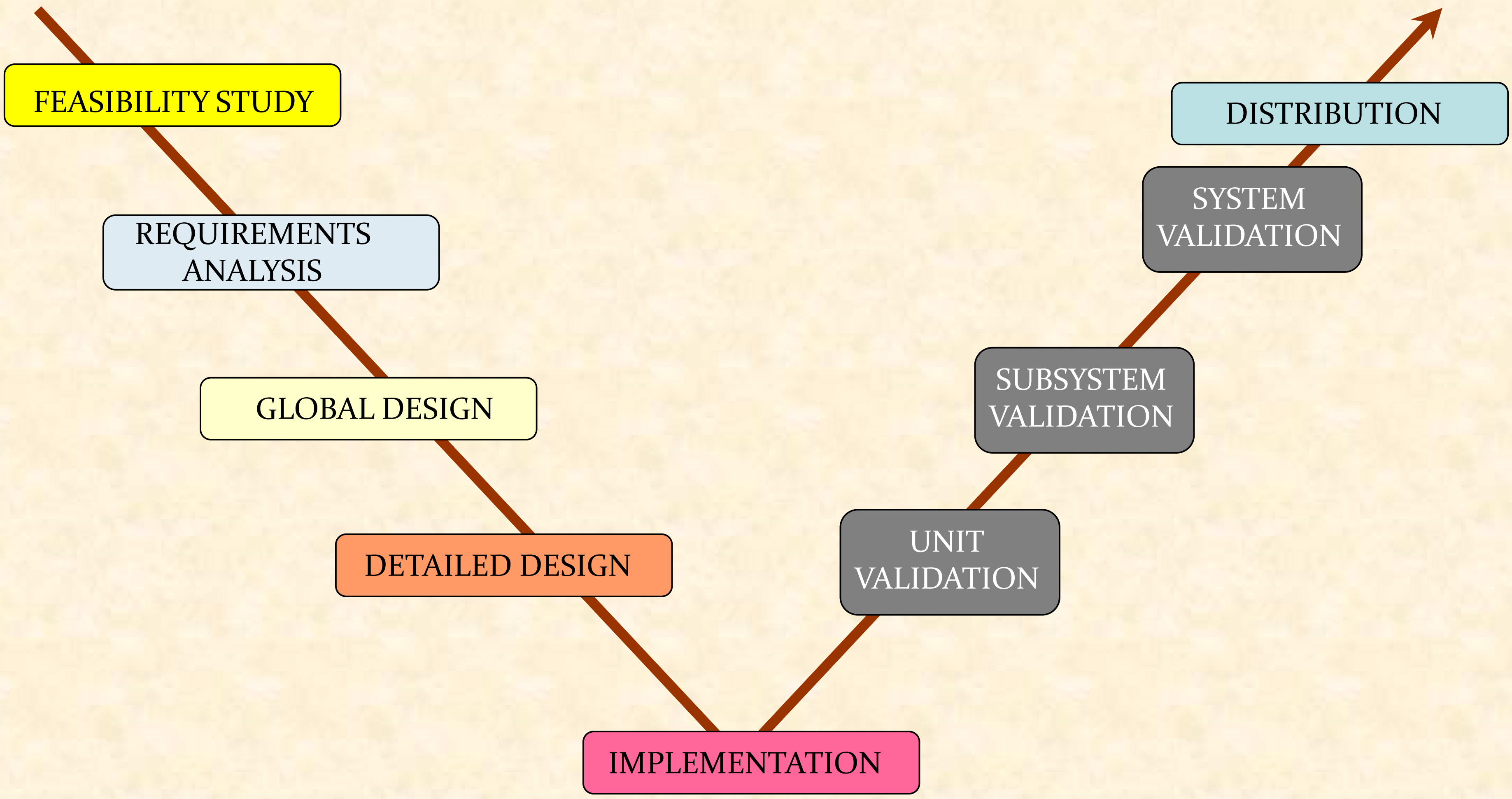
# Lifecycle: the waterfall model

Royce, 1970 (original article actually presented the model to *criticize* it!)




Succession of steps, with possibility at each step to question and update the results of the preceding step

# A V-shaped variant



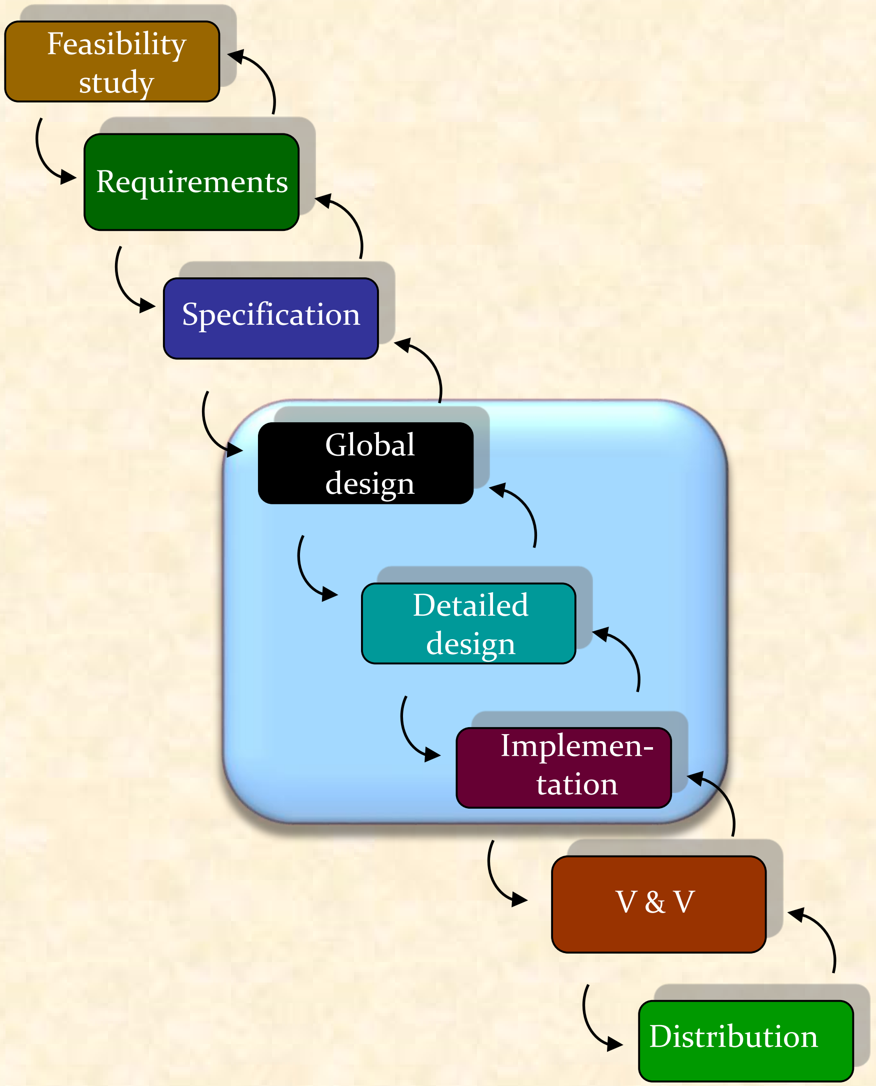
# Arguments for the waterfall

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

- The activities are necessary
  - (But: merging of middle activities)

- The order is the right one.

# Merging of middle activities



# Arguments for the waterfall

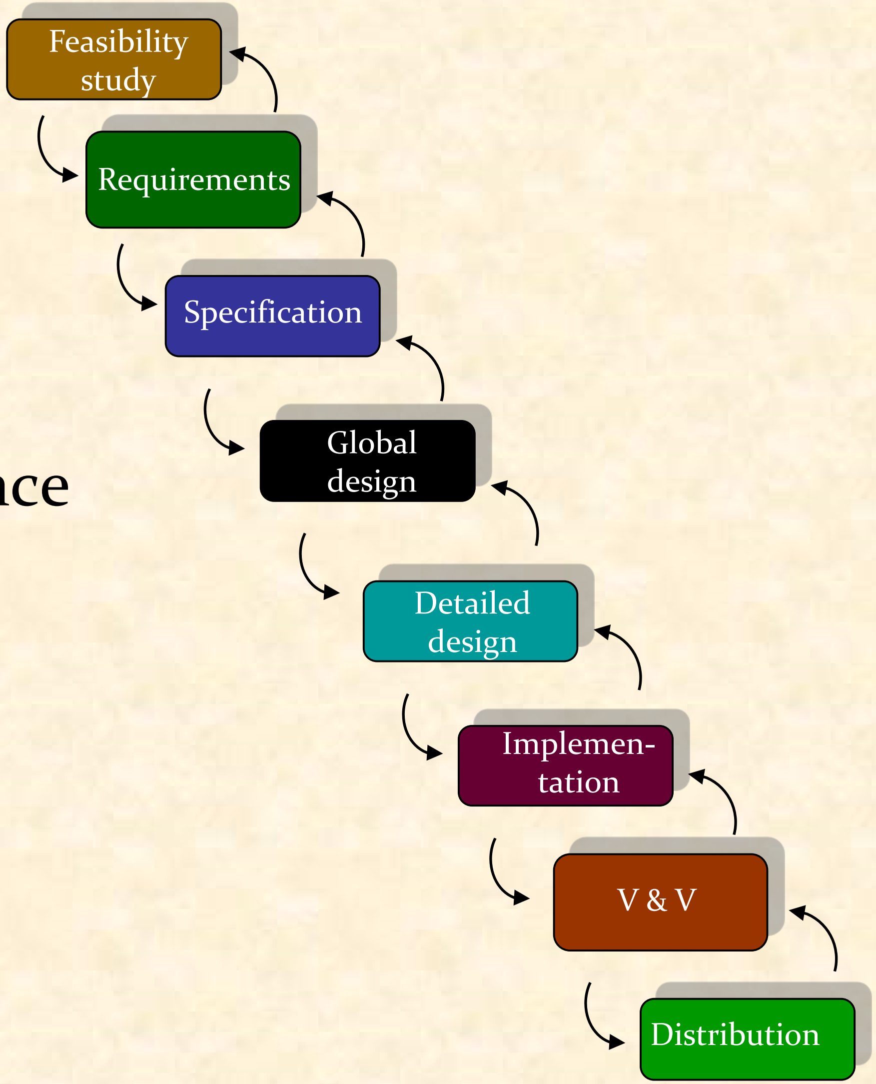
(After B.W. Boehm: *Software engineering economics*)

- The activities are necessary
  - (But: merging of middle activities)

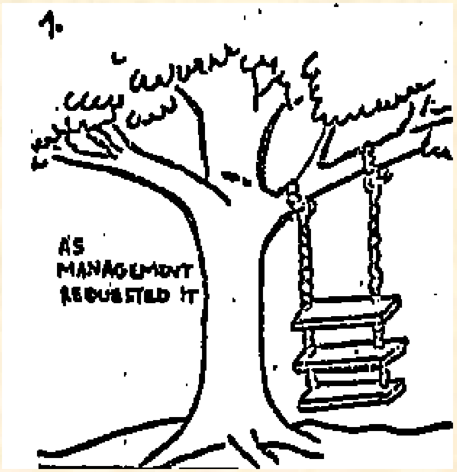
- The order is the right one.

# Problems with the waterfall

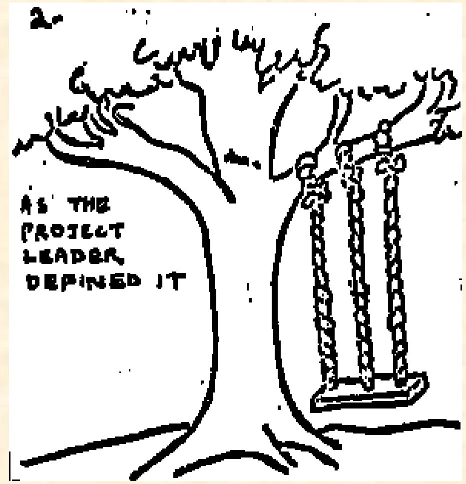
- Late appearance of actual code
- Lack of support for requirements change — and more generally for extendibility and reusability
- Lack of support for the maintenance activity (70% of software costs?)
- Division of labor hampering Total Quality Management
- Impedance mismatches
- Highly synchronous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



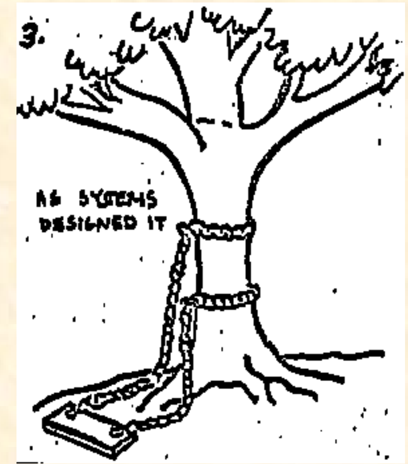
# Lifecycle: “impedance mismatches”



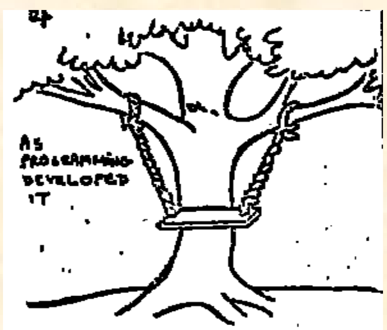

*As Management requested it*



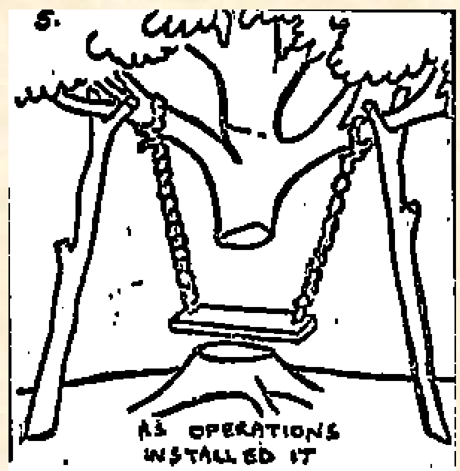

*As the Project Leader defined it*



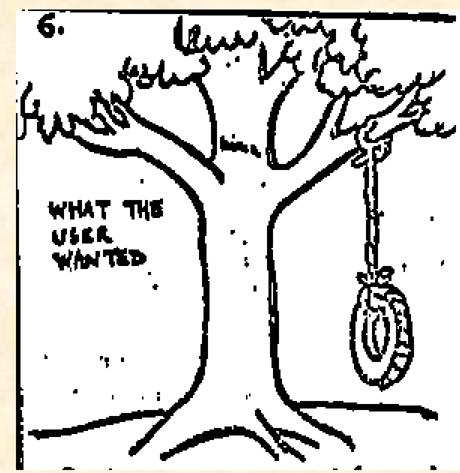

*As Systems designed it*




*As Programming developed it*



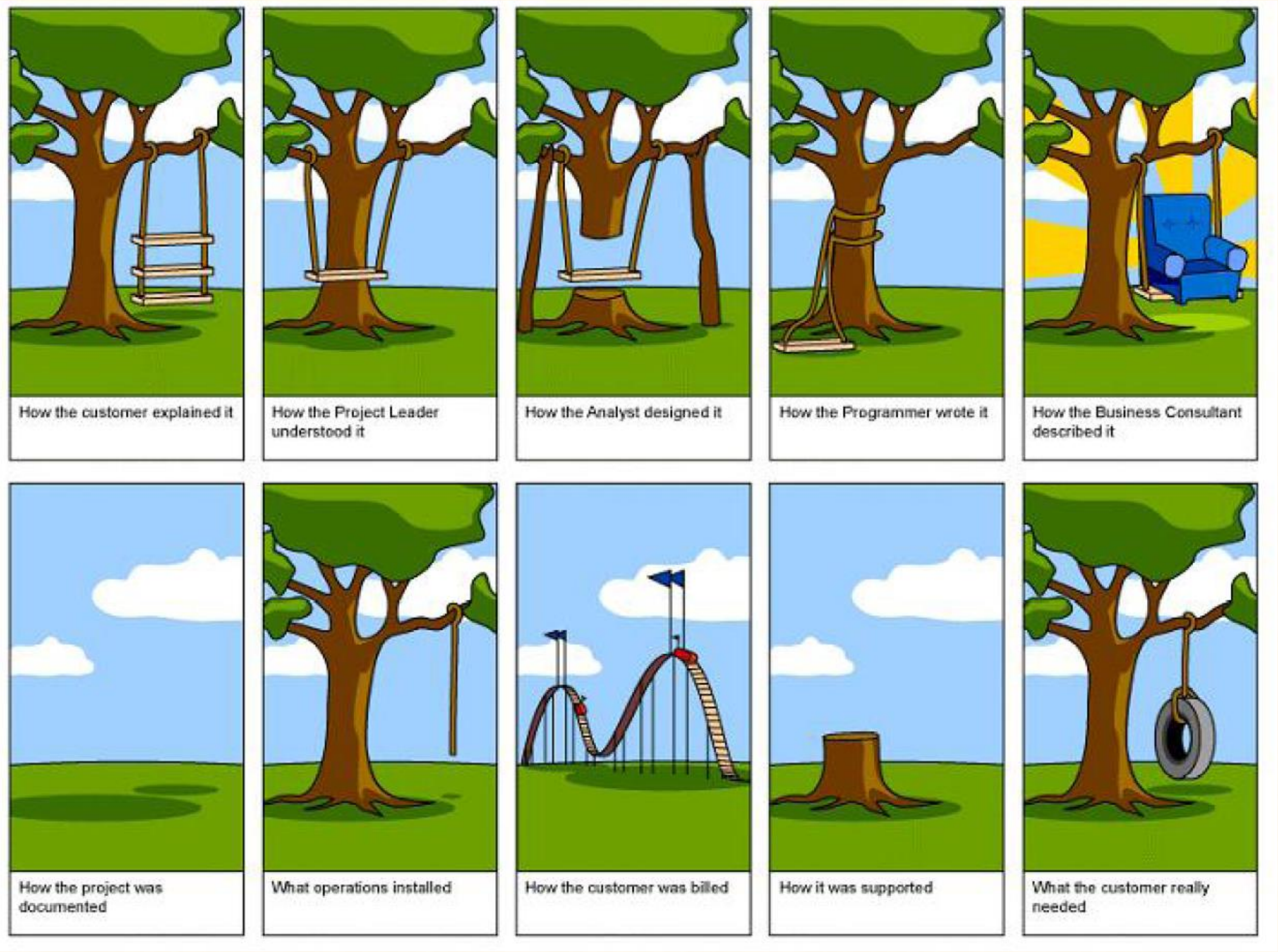

*As Operations installed it*




*What the user wanted*

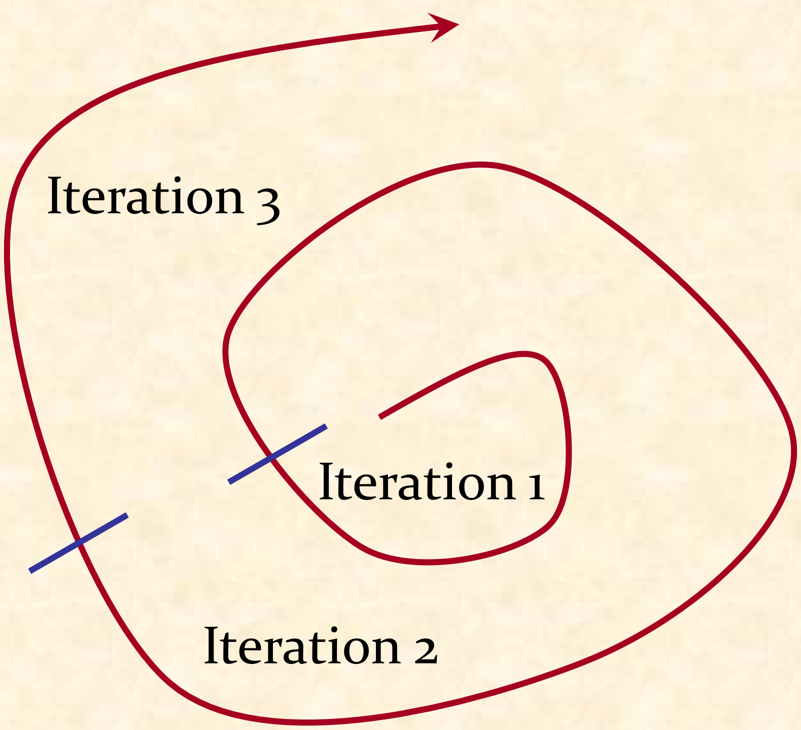
(Pre-1970 cartoon; origin unknown)

# A modern variant

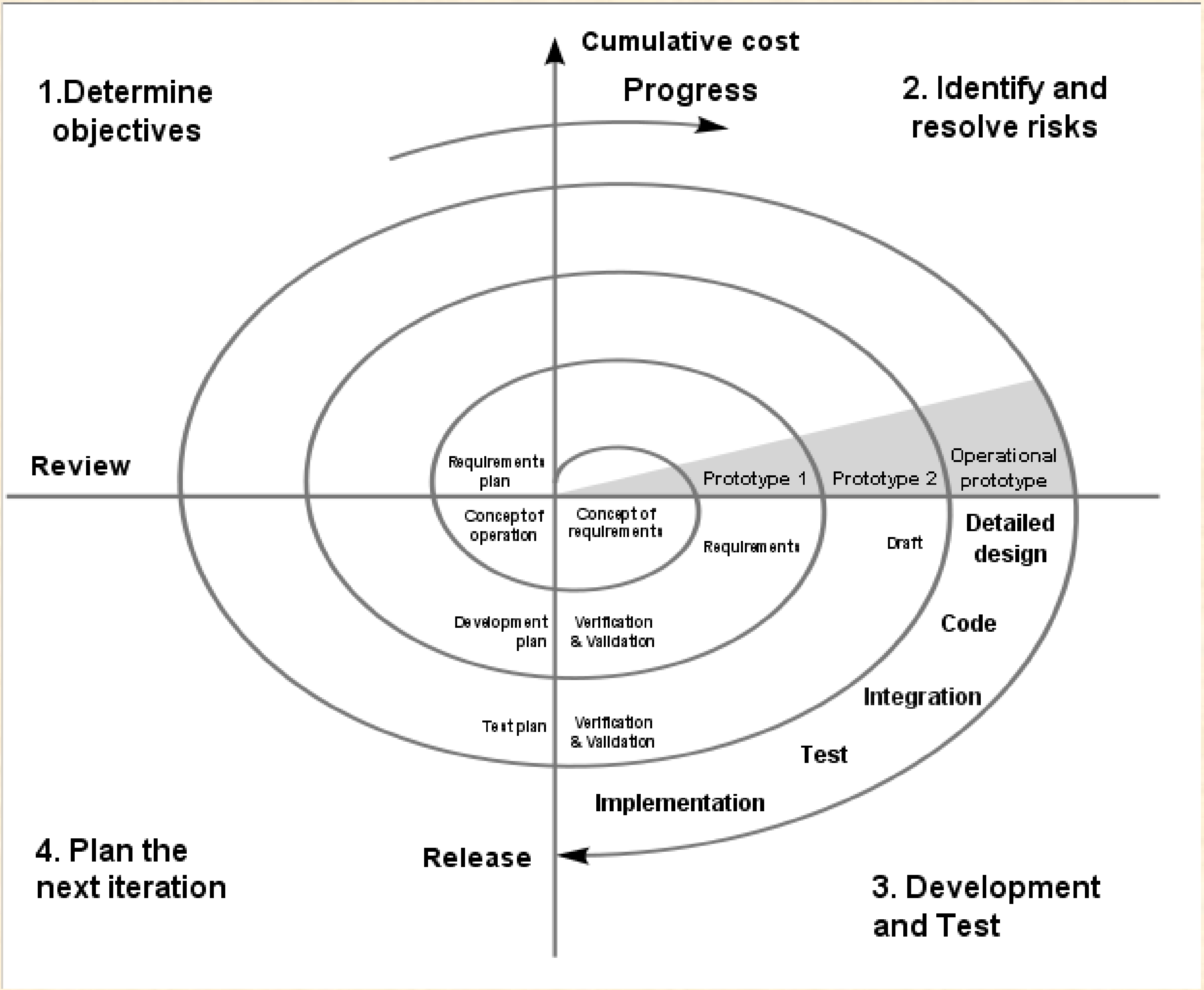


# The spiral model (Boehm)

Apply a waterfall-like approach to successive prototypes



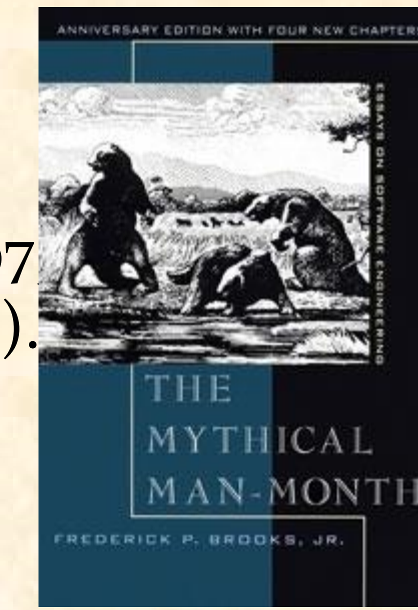
# The Spiral model



# "Prototyping" in software

The term is used in one of the following meanings:

- 1. Experimentation:
  - Requirements capture
  - Try specific techniques: GUI, implementation ("buying information")
- 2. Pilot project
- 3. Incremental development
- 4. Throw-away development
  (Fred Brooks, *The Mythical Man-Month*, 197
  "Plan to throw one away, you will anyhow").



# The problem with throw-away development

Software development is hard because of the need to reconcile conflicting criteria, e.g. portability and efficiency

A prototype typically sacrifices some of these criteria

Risk of shipping the prototype

In the 20$^{th}$-anniversary edition of his book (1995), Brooks admitted that “plan to throw one away” is bad advice

# The agile view

Iterative development

Short iterations (“sprints”), typically 1 month

Every iteration should produce a working system

# Seamless, incremental development

Seamless development:

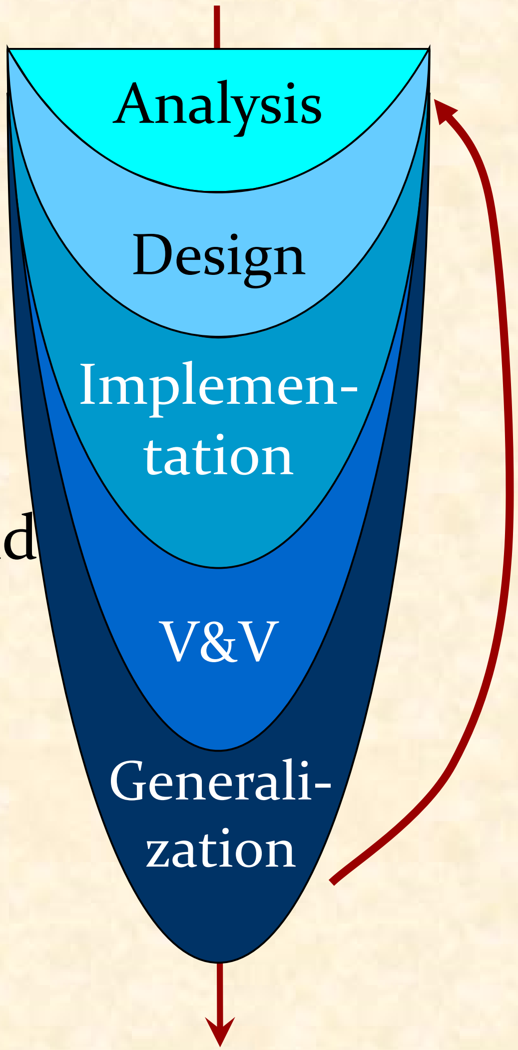
- Single set of notation, tools, concepts, principles throughout
- Continuous, incremental development
- Keep model, implementation and documentation consistent

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




Example classes:

*PLANE, ACCOUNT, TRANSACTION...*

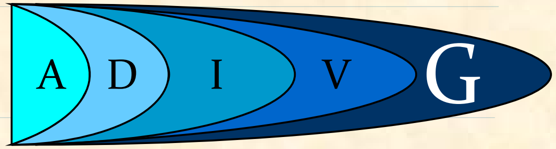
*STATE, COMMAND...*

*HASH_TABLE...*

*TEST_DRIVER...*

*TABLE...*

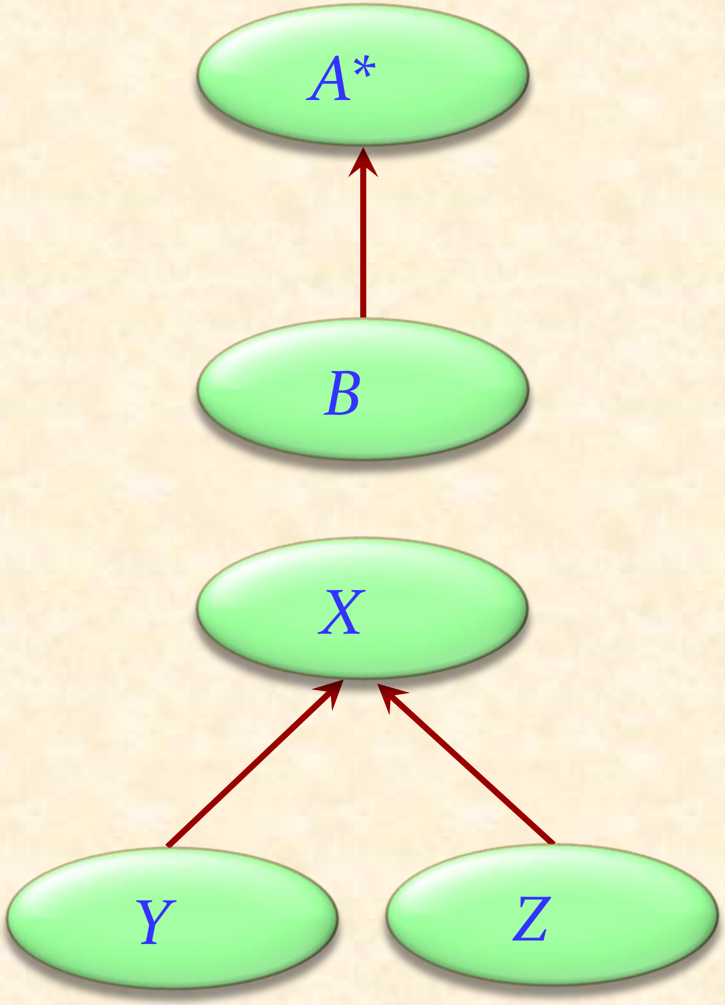
# Generalization




Prepare for reuse. For example:

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

Few companies have the guts to provide the budget for this



# Finishing a design

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

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

# Finishing a design

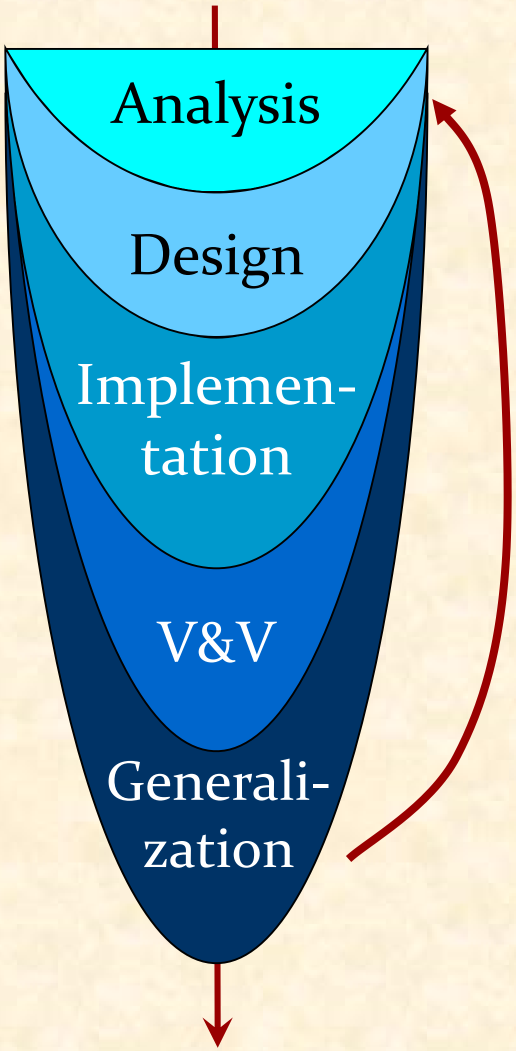
*Il semble que tout l'effort industriel de l'homme, tous ses calculs, toutes ses nuits de veille sur les épures, n'aboutissent […] qu'à la seule simplicité, comme s'il fallait l'expérience de plusieurs générations pour dégager peu à peu la courbe d'une colonne, d'une carène, ou d'un d'avion, jusqu'à leur rendre la pureté élémentaire de la courbe d'un sein ou d'une épaule. Il semble que le travail des ingénieurs, […] des calculateurs du bureau d'études ne soit ainsi, en apparence, que de polir et d'effacer, d'alléger […] Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.*

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

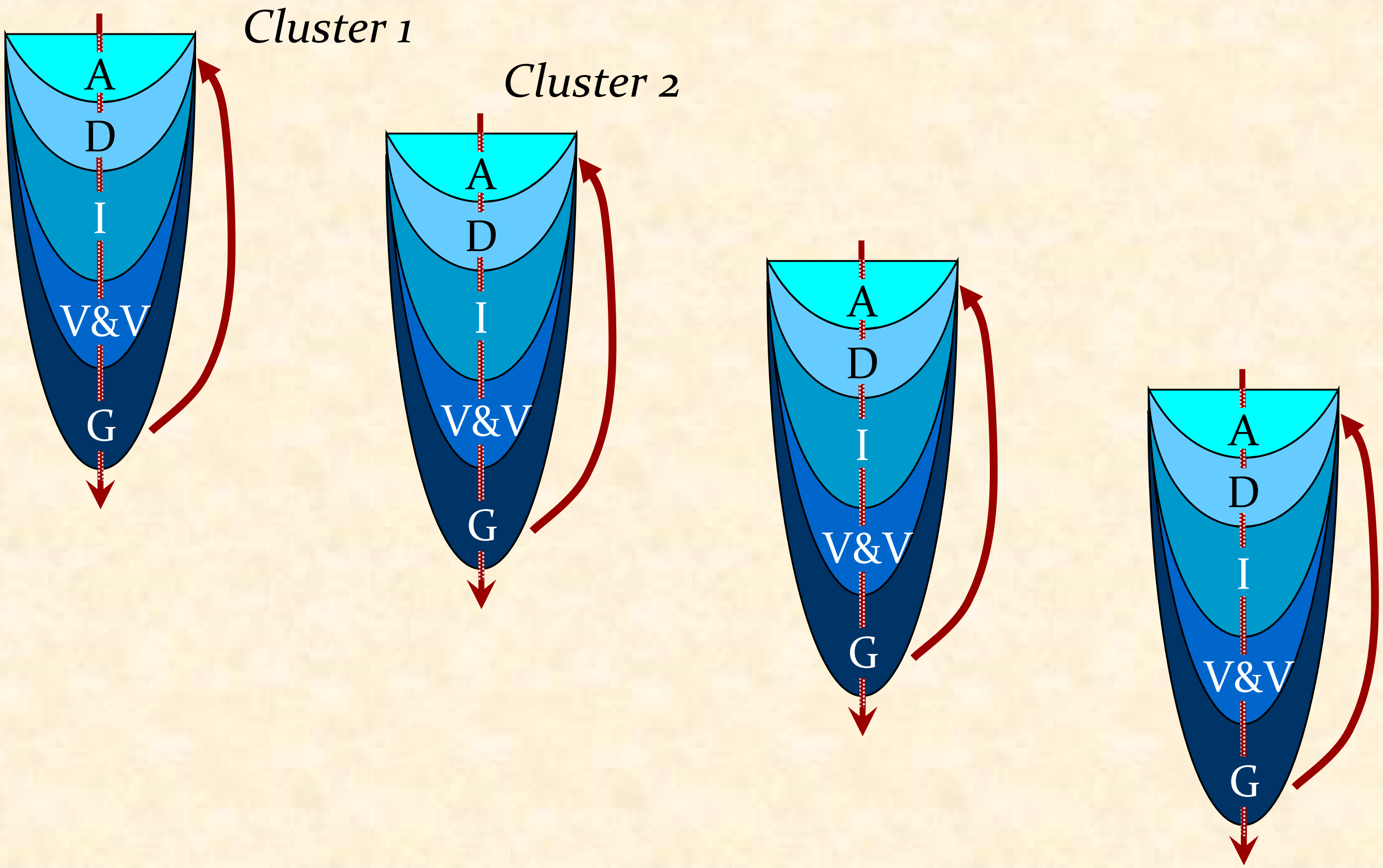
# Steve Jobs, 1998

That's been one of my mantras -- focus and simplicity. Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there, you can move mountains.

# Reversibility

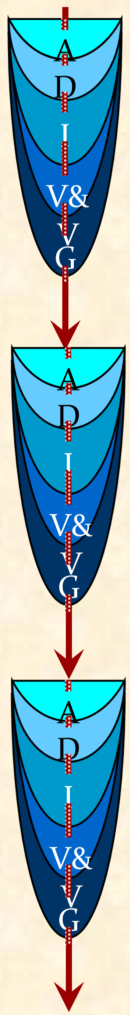


# The cluster model



# Extremes

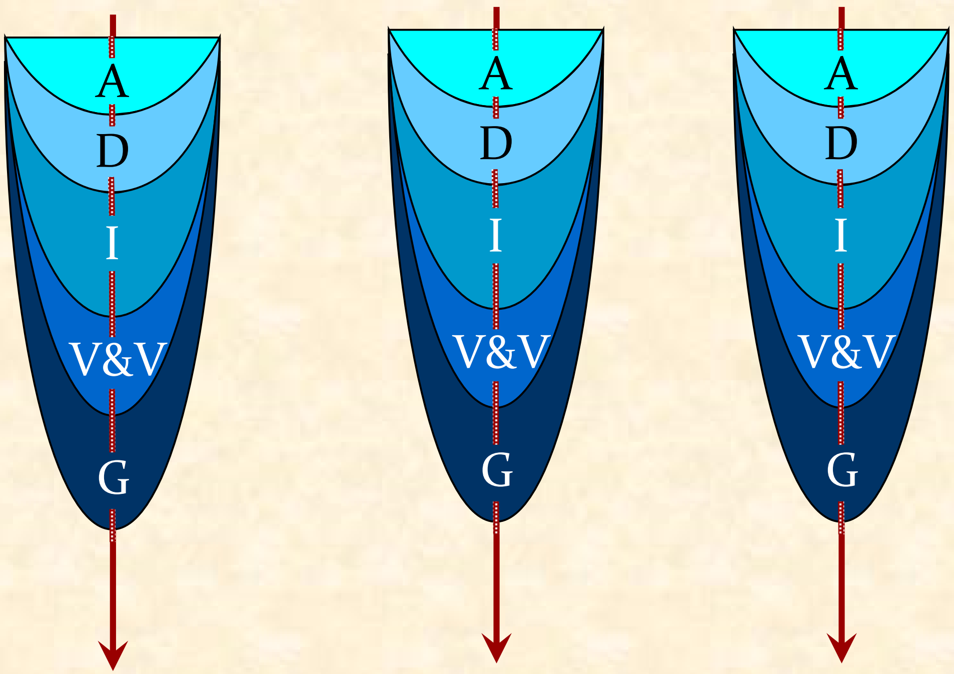
“Trickle”




*Cluster 1*

*Cluster 2*

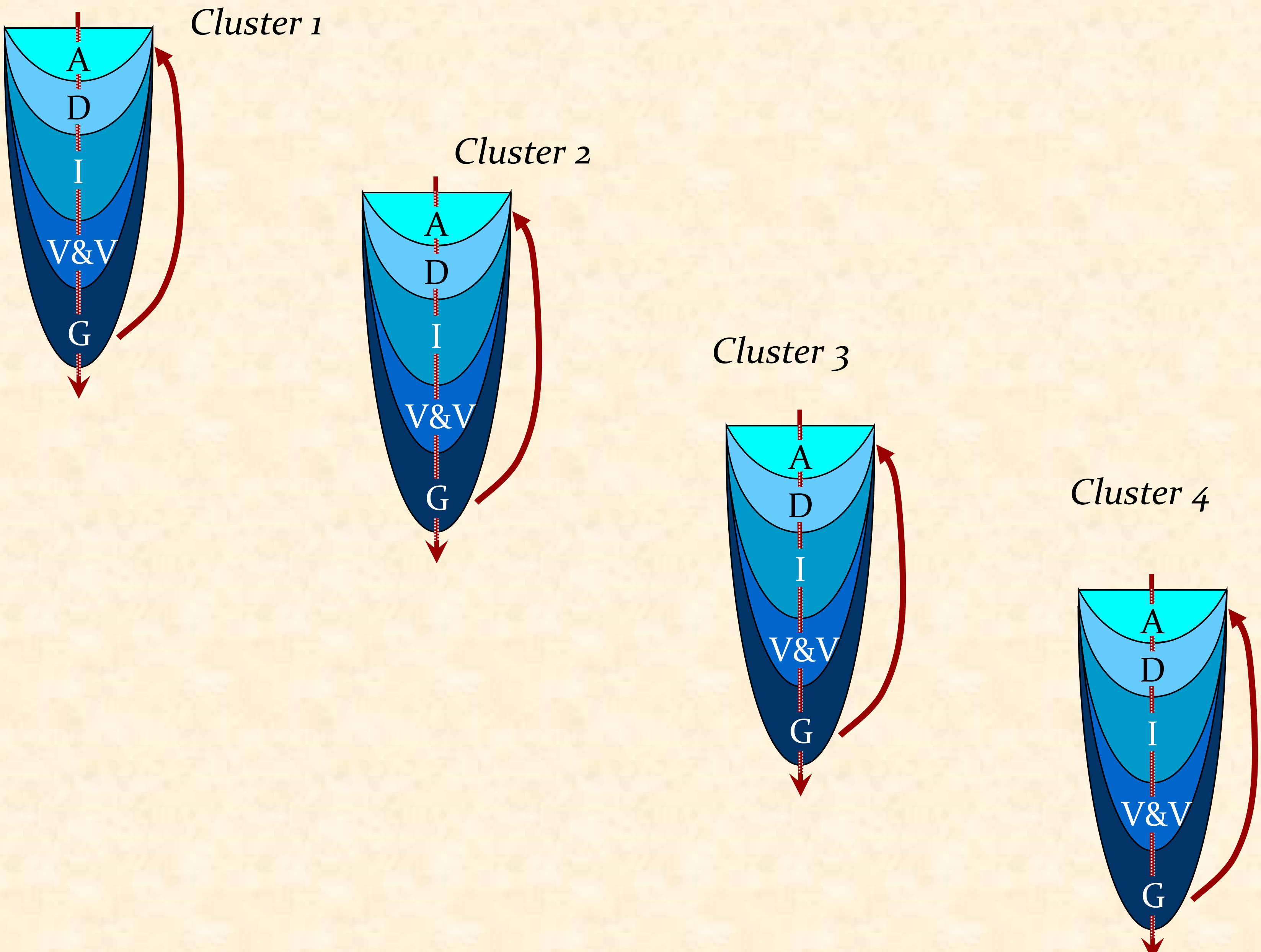
“Clusterfall”




*Cluster 1*

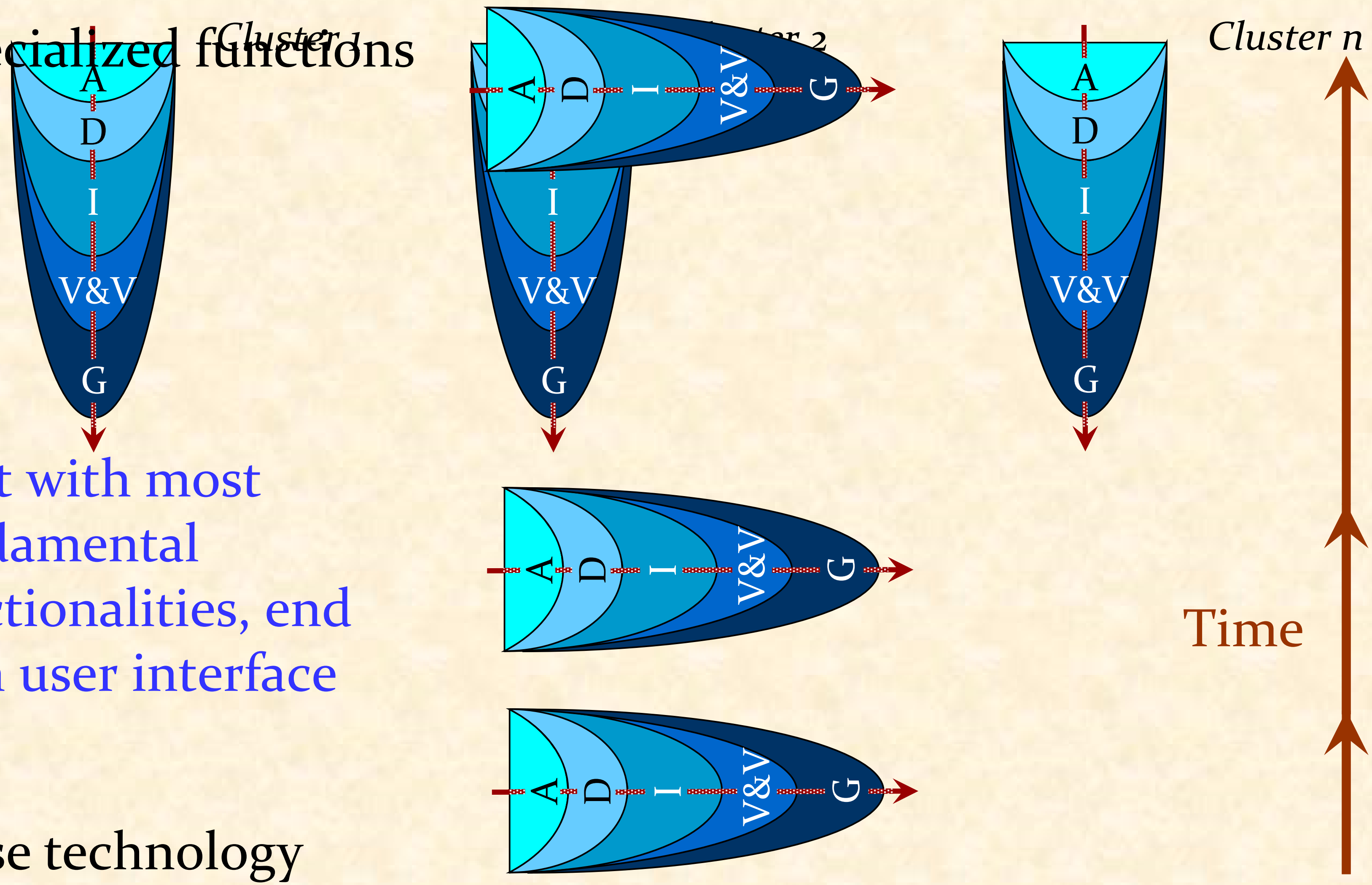
*Cluster 2*

# Dynamic rearrangement



# Bottom-up order of cluster development

Specialized functions




Start with most fundamental functionalities, end with user interface

Base technology

# Seamless development with EiffelStudio

Diagram Tool

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

No need for separate UML tool

Metrics Tool

Profiler Tool

Documentation generation tool

...

# Complementary approaches

Seamless development: “vertical”

Agile: horizontal

# Lifecycle models: summary

Software development involves fundamental tasks such as requirements, design, implementation, V&V, maintenance…

Lifecycle models determine how they will be ordered

The Waterfall is still the reference, but many variants are possible, e.g. Spiral, Cluster

Seamless development emphasizes the fundamental unity of the software process