ETH Zürich
Chair of Software Engineering

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
- Raffaelle Ranzani

Course page

- http://se.inf.ethz.ch/courses/2013b_fall/rpl

Forum

- https://piazza.com/class/hktyugazkg35d3

Practical details

Schedule

- Lecture: Monday, 16:15 – 18:00, RZ F 21
- Exercise: Thursday, 13:15 – 15:00, IFW C 31
- Laboratory space: Most Wednesdays, 14:00 – 20:00 and Thursdays 09:00 – 12:00, IFW E 42 – Exact schedule on the course page

This is a hands-on laboratory class. The exercise sessions will be much more interactive than in traditional courses.

Use the forum to post your questions and answer questions other have.

Suggestions to improve the course are welcome.

Grading

The grade for this laboratory course is based **entirely on the project**. You must submit your work at the end of each phase and participate in the final competition to receive a grade for this class.

- Assignment 0 (3 Oct): setup - No grade
- Assignment 1 (17 Oct): control and obstacle avoidance - 20%
- Assignment 2 (7 Nov): localization and mapping - 20%
- Assignment 3 (28 Nov): path planning and object recognition - 30%
- Final competition (16 Dec): search and rescue - 30%

Assignment 1 and 2 are individual work. Assignment 3 and the final competition are group work. Please find a partner by the end of Assignment 2.

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"

Course content

Control and obstacle avoidance

- ROS and Roboscoop, SCOOP, Robot control and obstacle avoidance, Design patterns

Localization and mapping

- Localization, Mapping, Modern software engineering tools

Path planning and object recognition

- Path planning, Robot perception, Software architecture in robotics

Search and rescue

- Multirobot system

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

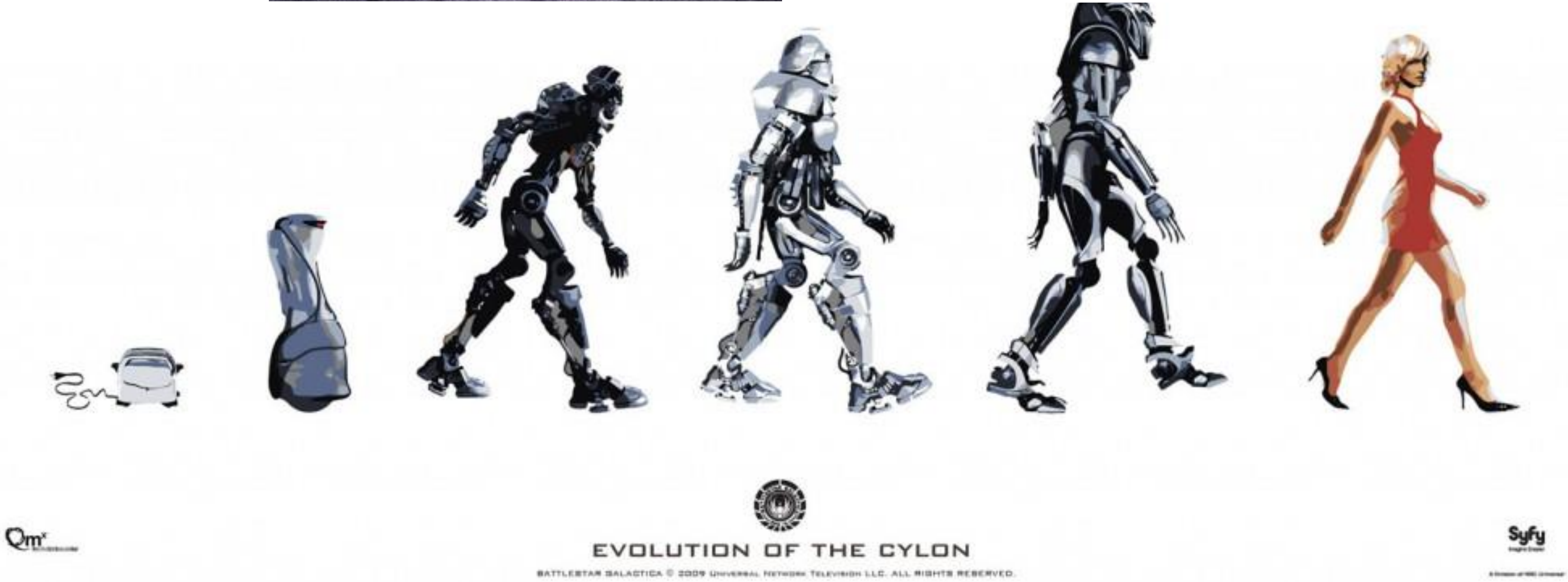
Robotics

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

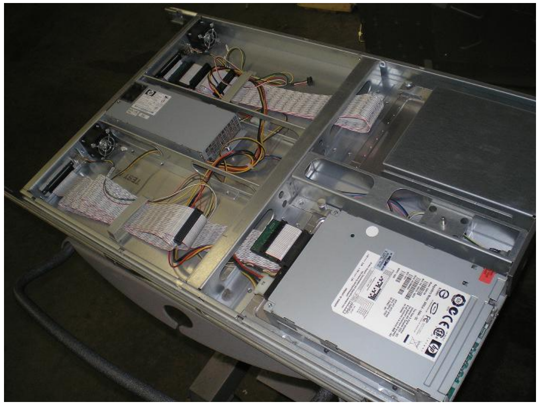
Programming language

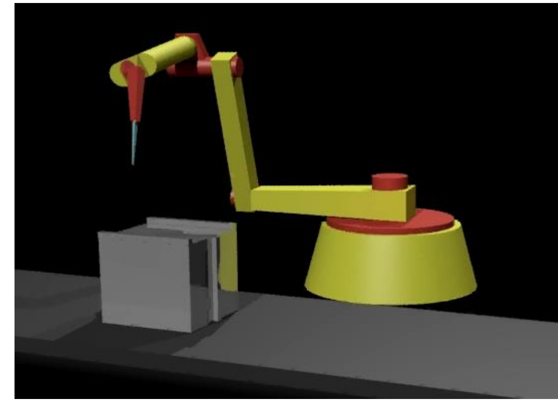
- *Touch of Class*, Meyer
- *The C++ Programming Language*, Stroustrup

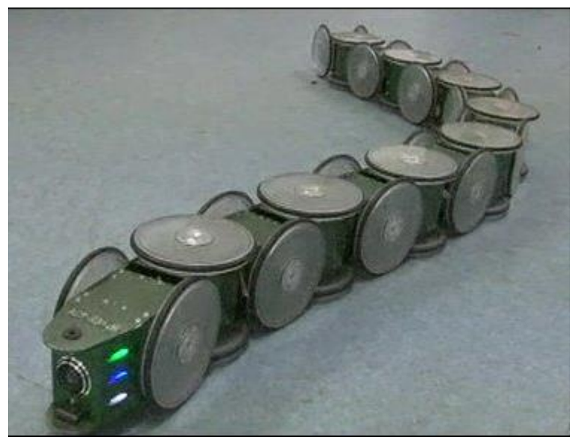
Robots: your point of view



Robots: your point of view

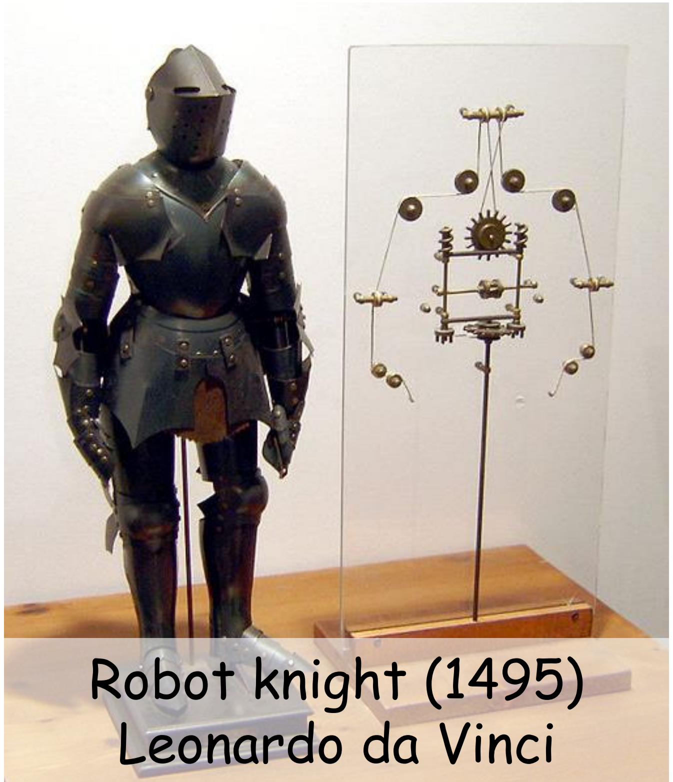






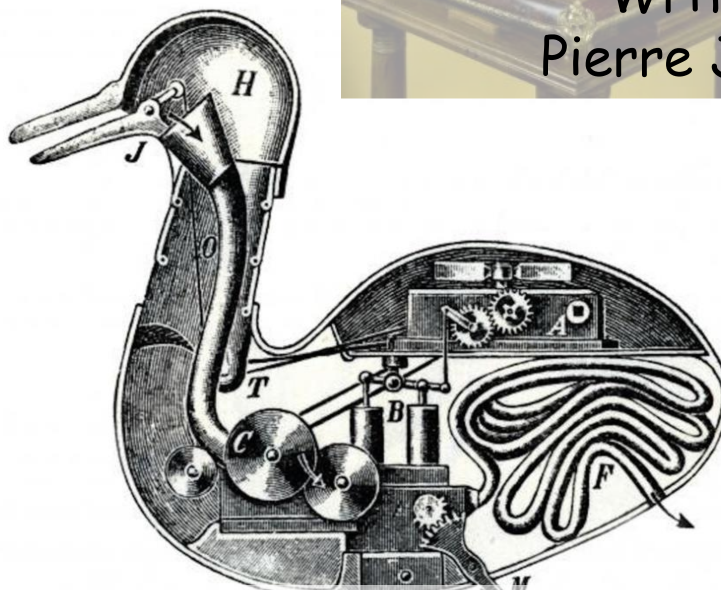
The pictures are for one-time, non-commercial use.

Robots as automata



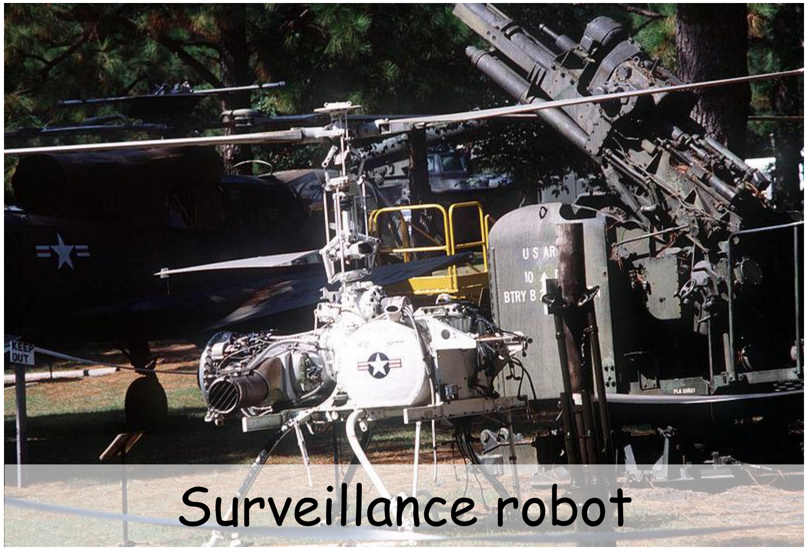
Robot knight (1495)
Leonardo da Vinci

Writer (1774)
Pierre Jaquet-Droz

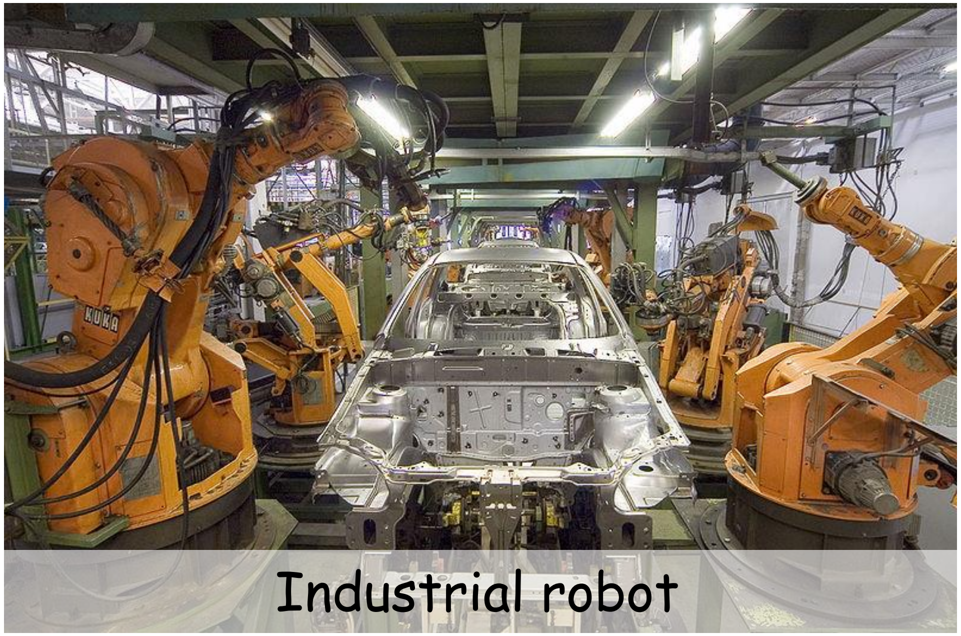


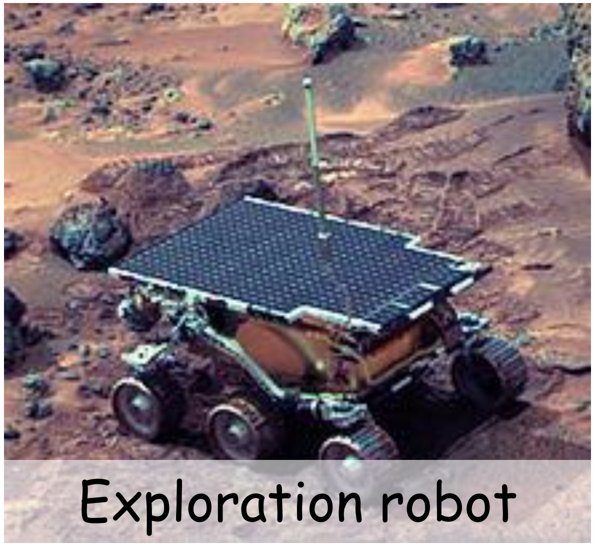
Digesting duck (1738)
Jacques de Vaucanson

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

Challenges in robotics

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

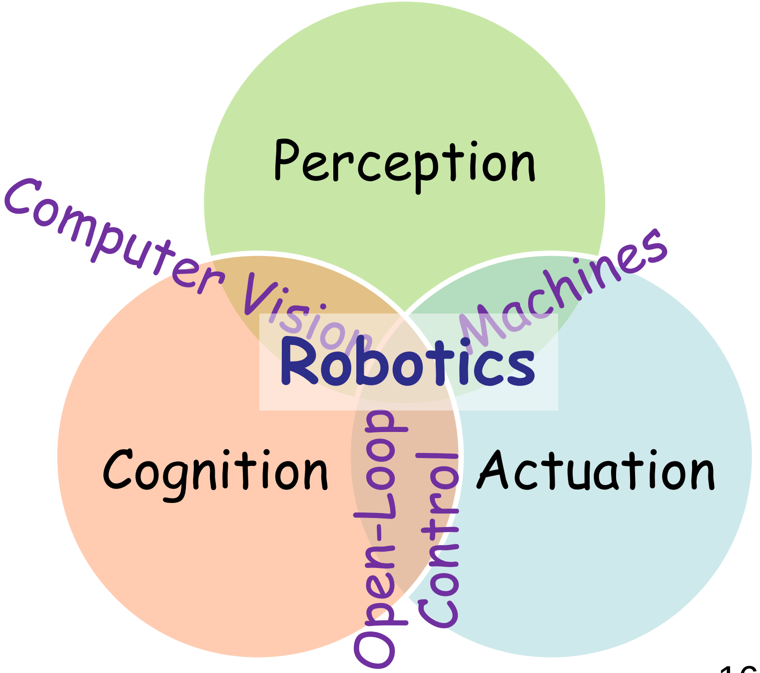
Robotics

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

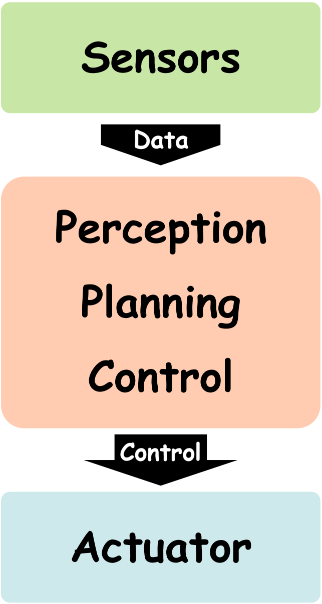
- Perception: Vision, Touch, Range, Sound
- Actuation: Manipulation, Locomotion
- Cognition: Navigation, Recognition, Planning, Interaction



Robotic software architecture

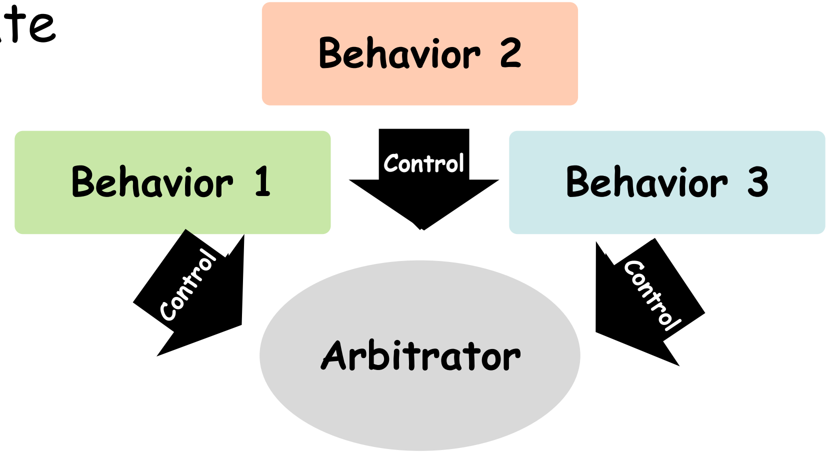
Sense-Plan-Act[1]

1. Sense the environment.
2. Plan the next move based on short- and long- term goals.
3. Execute the plan through the actuators.



Subsumption architecture[2]

1. Divide the control into different behaviors, where the higher level behavior subsumes the lower level behaviors.
2. Let the arbitrator pick the appropriate behavior for the given condition.



[1] Nilsson, N. *Principles of Artificial Intelligence*. Palo Alto: Tioga. 1980.

[2] Brooks, R. "A robust layered control system for a mobile robot". *IEEE Journal of Robotics and Automation*, 2 (1): 14–23. 1986.

Robotic software architecture

Three-layer architecture

- Deliberator

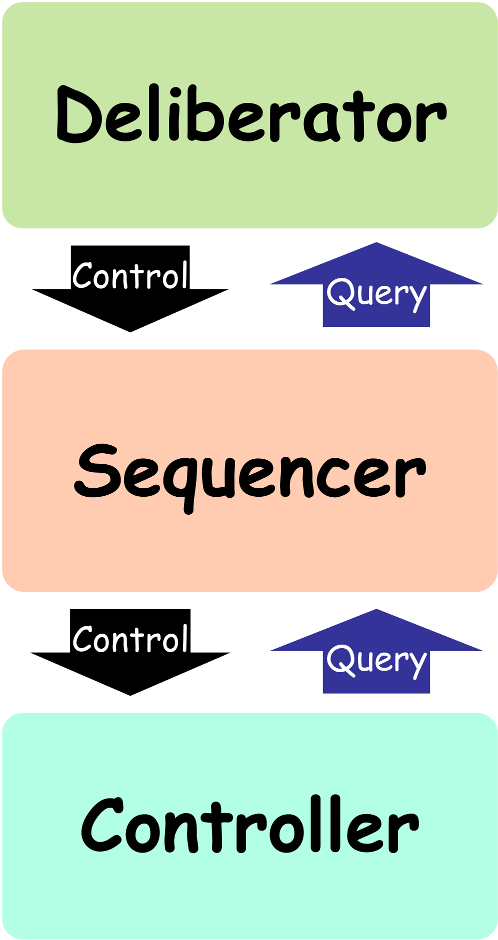
  Perform high-level computations such as planning and vision processing

- Sequencer

  Select which primitive behavior the controller should use at a given time and supply parameters for the behavior.

- Controller

  Perform primitive behaviors, with tight coupling of sensors to actuators



Gat, E. "On three-layer architectures". *Artificial Intelligence and Mobile Robots*. AAAI Press. 1998.

Dimensions of robotic software architecture

- Hierarchical: Components are hierarchically organized.
- Modular: Each component is functionally independent.
- Asynchronous: Components operate asynchronously.
- Distributed: Components are physically separate. They may or may not be functionally independent.
- Interruptible: Architecture can handle interrupts. It may or may not resume the interrupted process afterwards.

Introduction to software engineering

(and software architecture)

A definition of software engineering

Wikipedia (from SWEBOK, the Software Engineering Body of Knowledge)

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software.

(Largely useless definition)

A simpler definition

"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

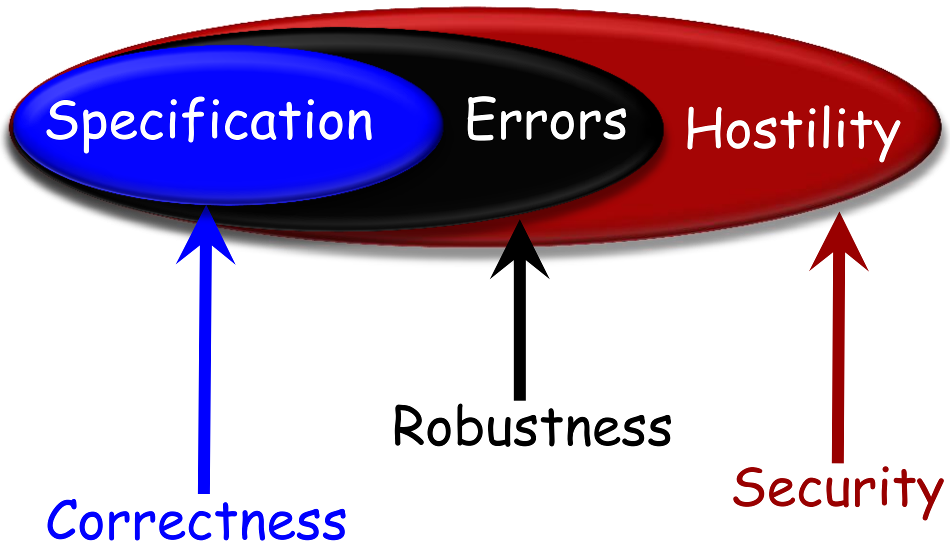
Software quality factors

Product

Immediate

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

"Reliability"



Long-term

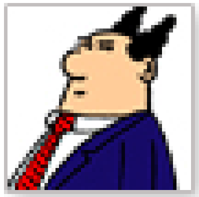
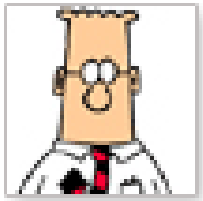
- Extendibility
- Reusability
- Portability

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

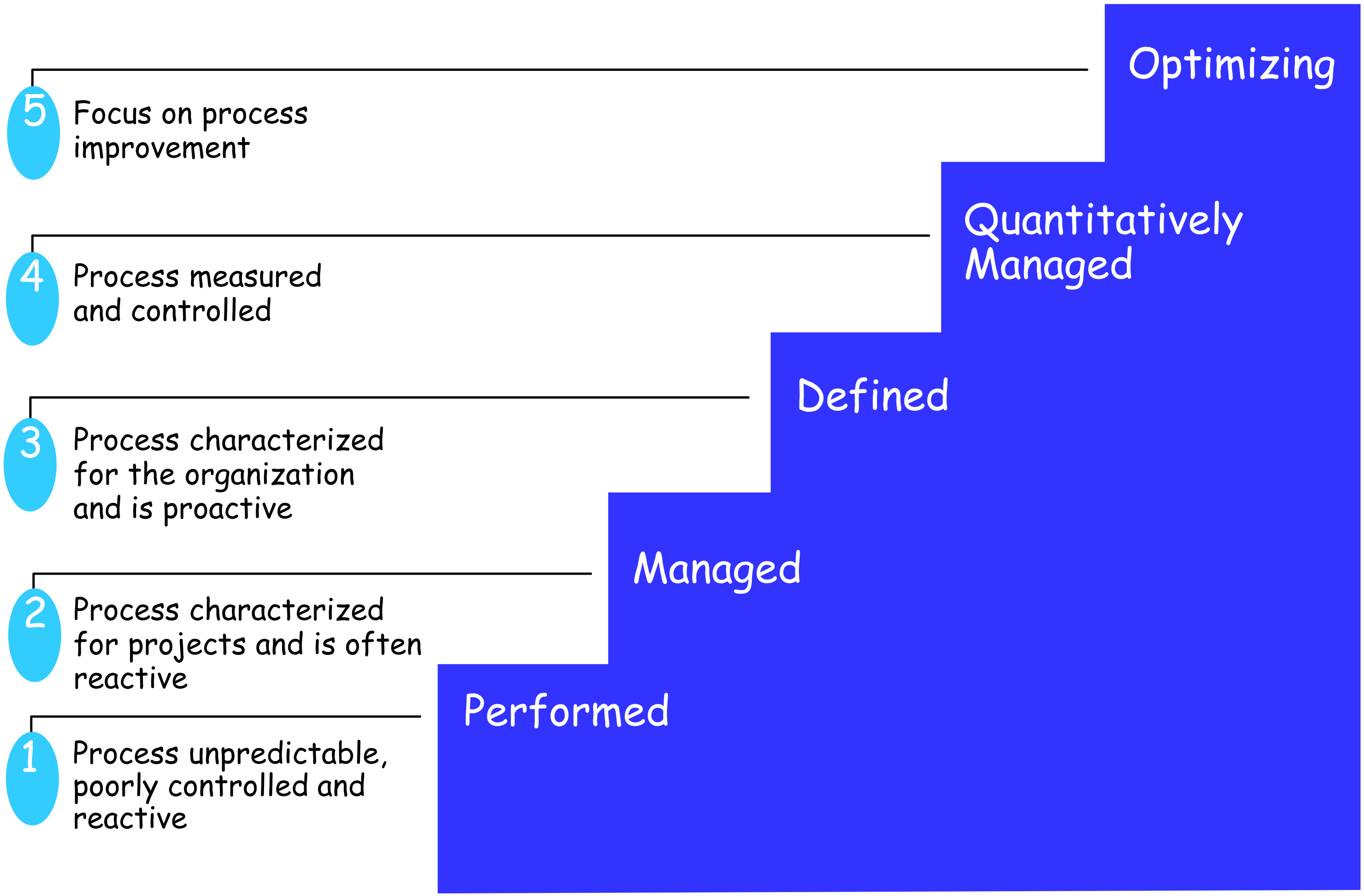
Emphasizes **reproducibility** of results

(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
- Working code; de-emphasis of plans and documents
- Testing; de-emphasis of specifications and design . “Test-Driven Development"
- Communication: customer involvement
- Refusal to commit to both functionality and deadlines
- 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

Object-oriented culture

Emphasizes:

- Seamless development
- Reversibility
- Single Product Principle
- Design by Contract

Six task groups of software engineering

Describe

Implement

Assess

Manage

Operate

Notate

Requirements, design specification, documentation …

Design, programming

V&V*, esp. testing

Plans, schedules, communication, reviews…

Deployment, installation,

Languages for programming etc.

** Validation & Verification*

Software lifecycle models

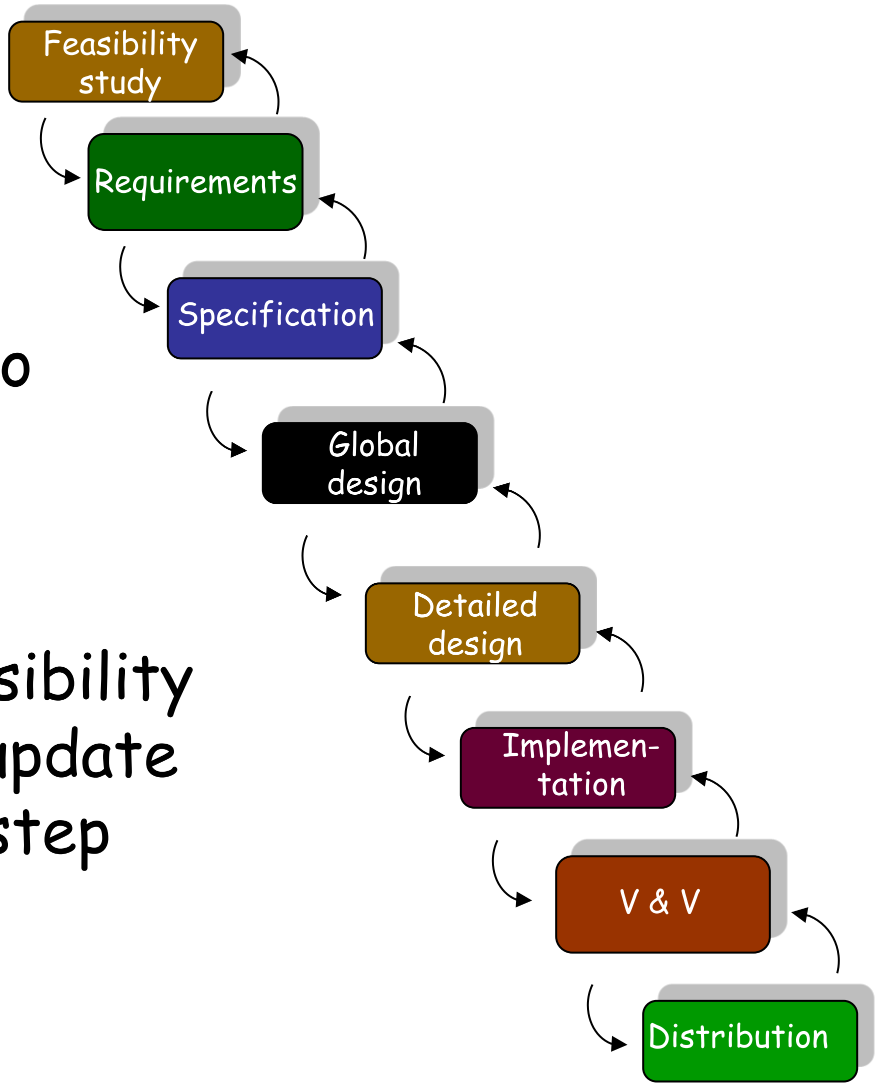
Describe an overall distribution of the software construction into tasks, and the ordering of these tasks

They are models in two ways:

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

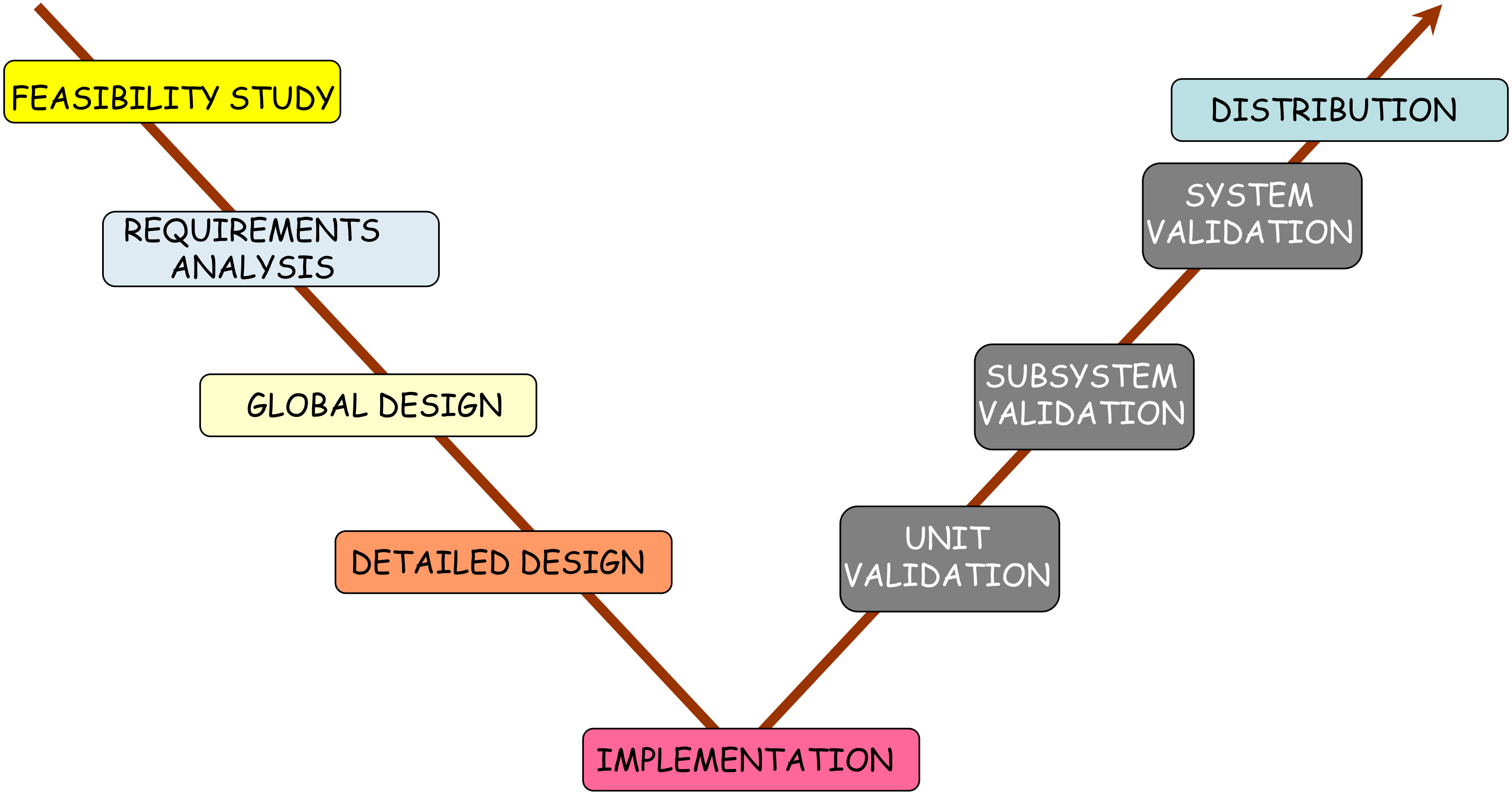
Lifecycle: the waterfall model

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



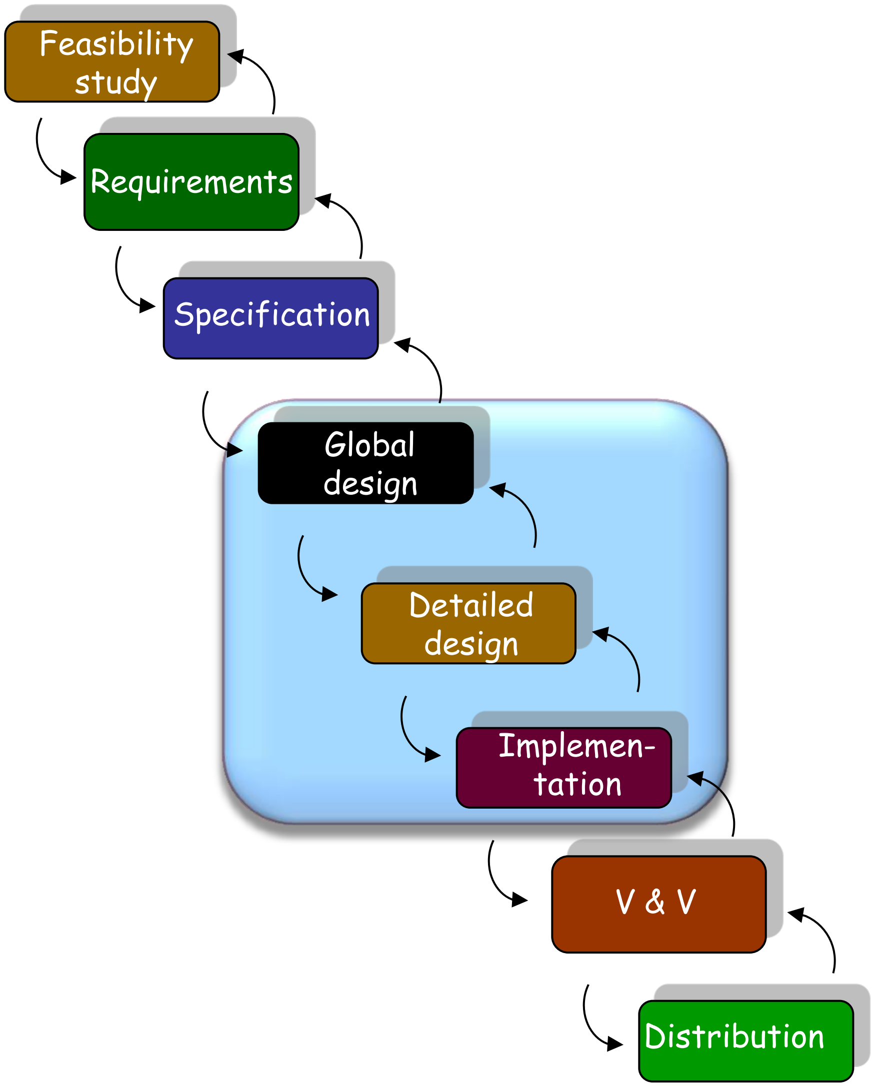
Arguments for the waterfall

(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



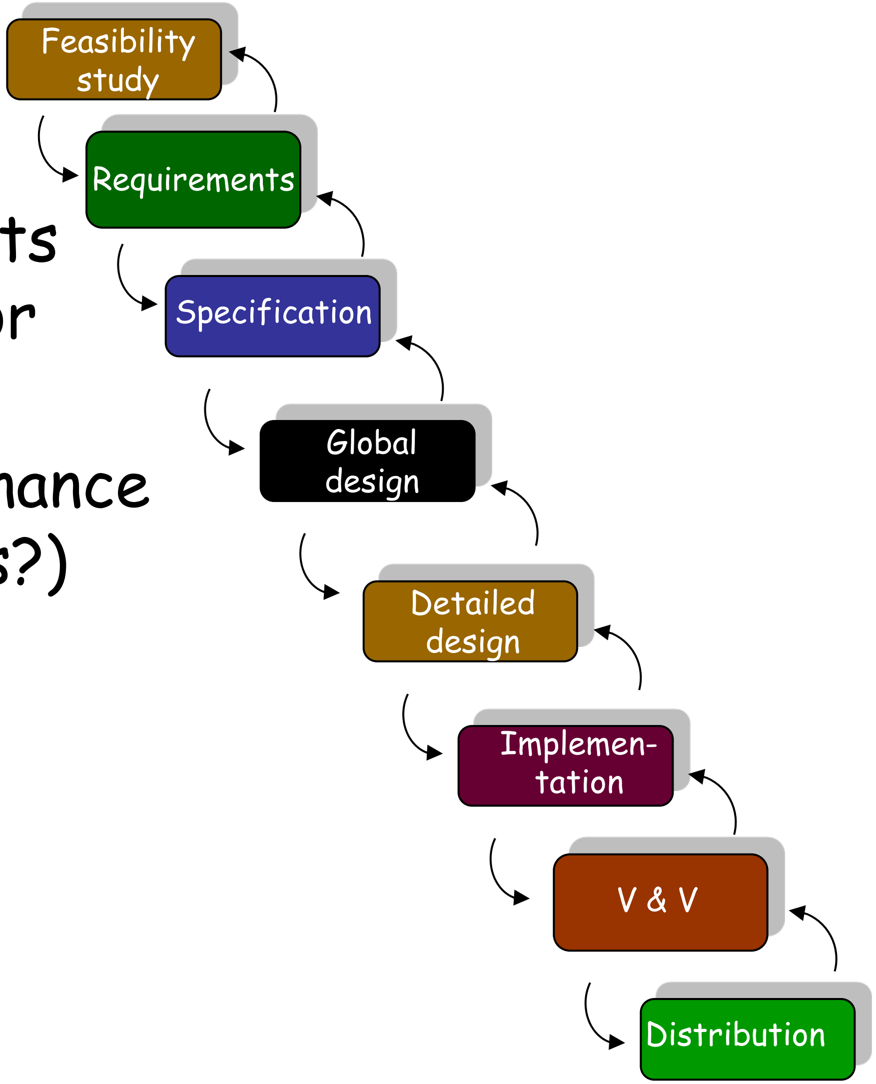
Arguments for the waterfall

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

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

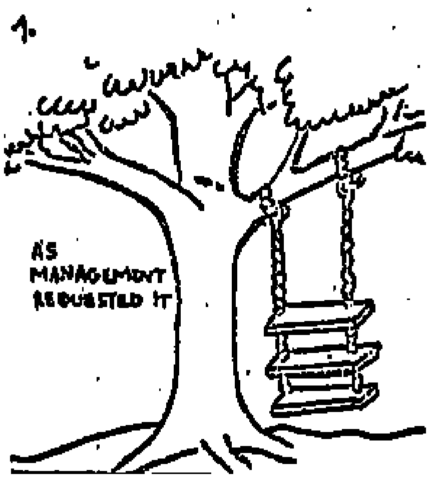
- The order is the right one.

Problems with the waterfall

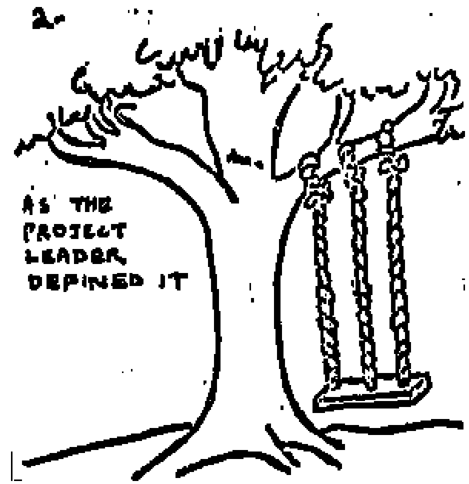


- Late appearance of actual code
- Lack of support for requirements change — and more generally for extendibility and reusability
- Lack of support for the maintenance activity (70% of software costs?)
- Division of labor hampering Total Quality Management
- Impedance mismatches
- Highly synchronous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

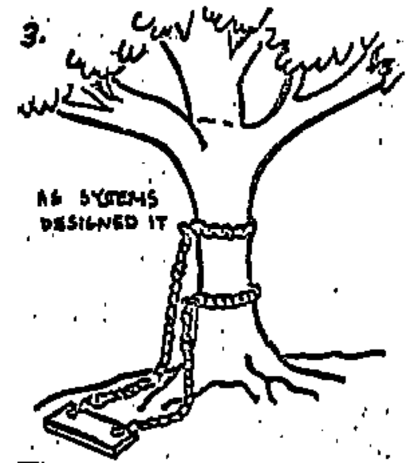
Lifecycle: “impedance mismatches”



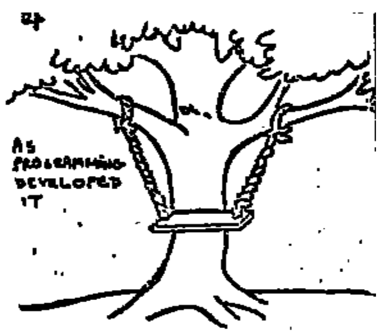
As Management requested it



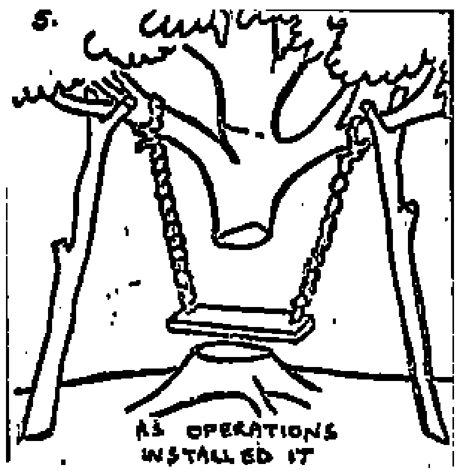
As the Project Leader defined it



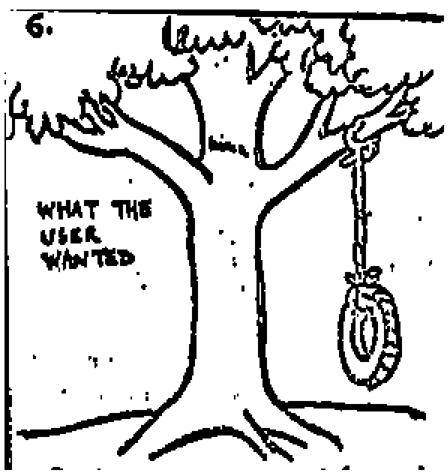
As Systems designed it



As Programming developed it



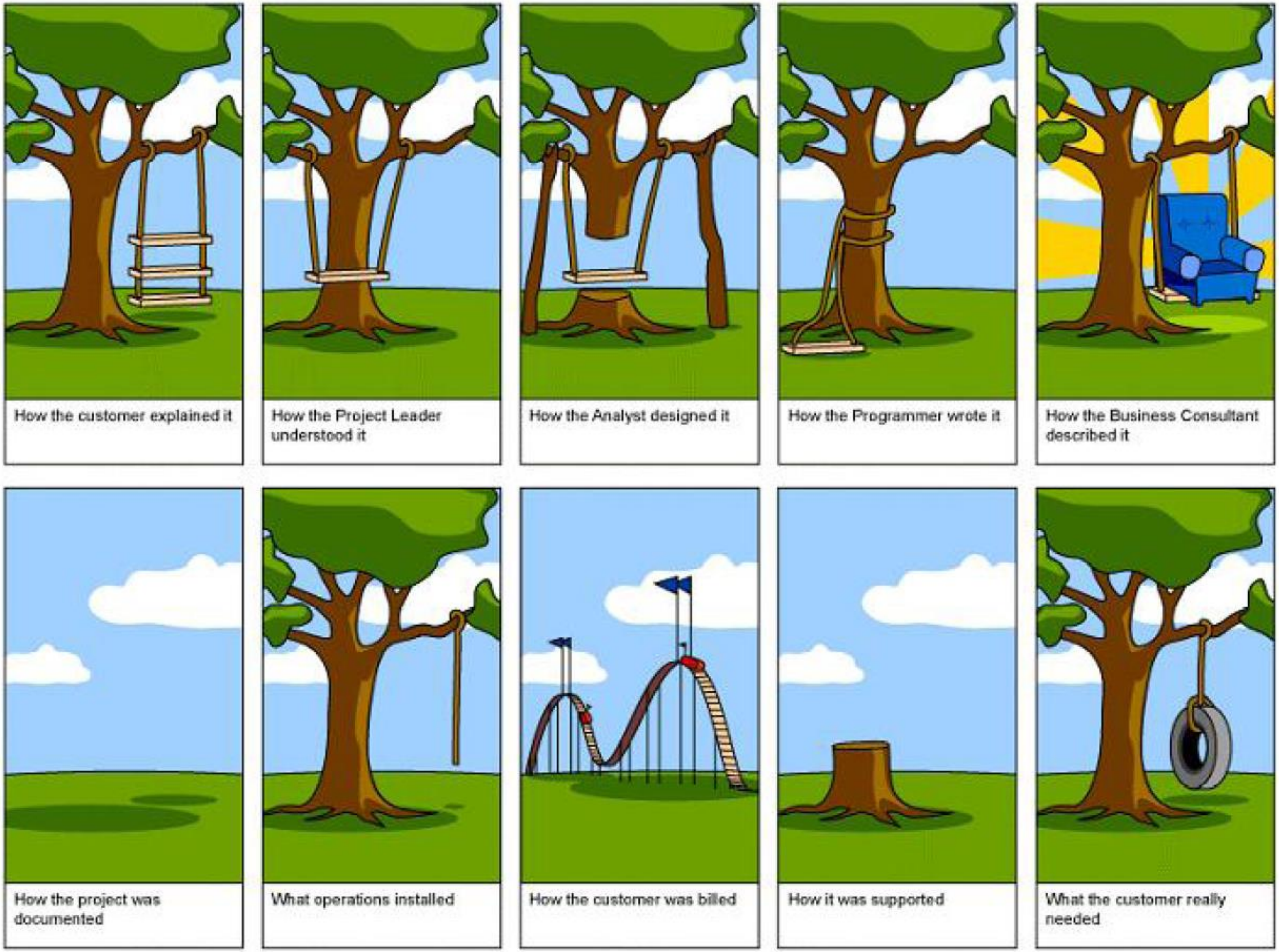
As Operations installed it



What the user wanted

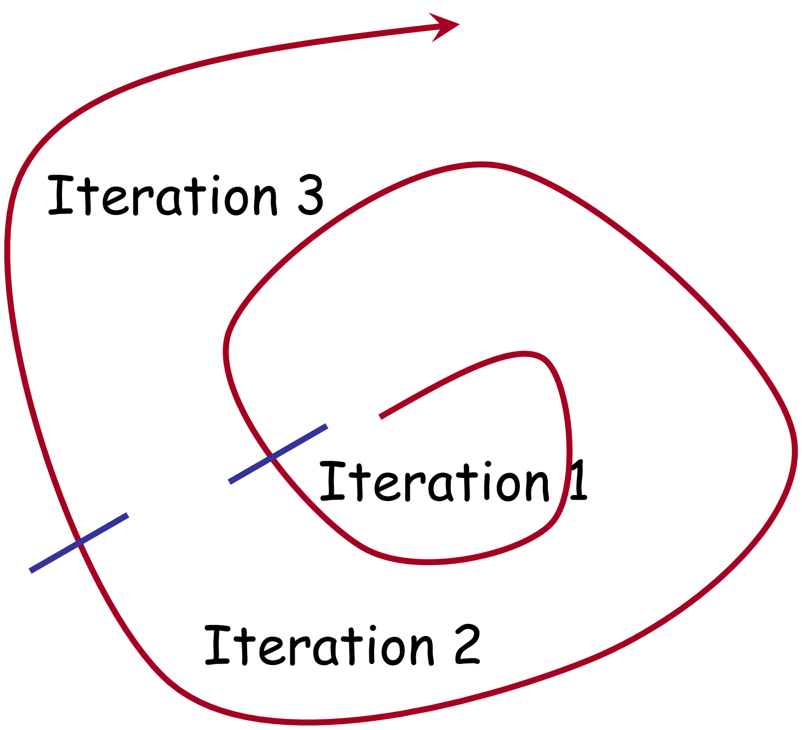
(Pre-1970 cartoon; origin unknown)

A modern variant

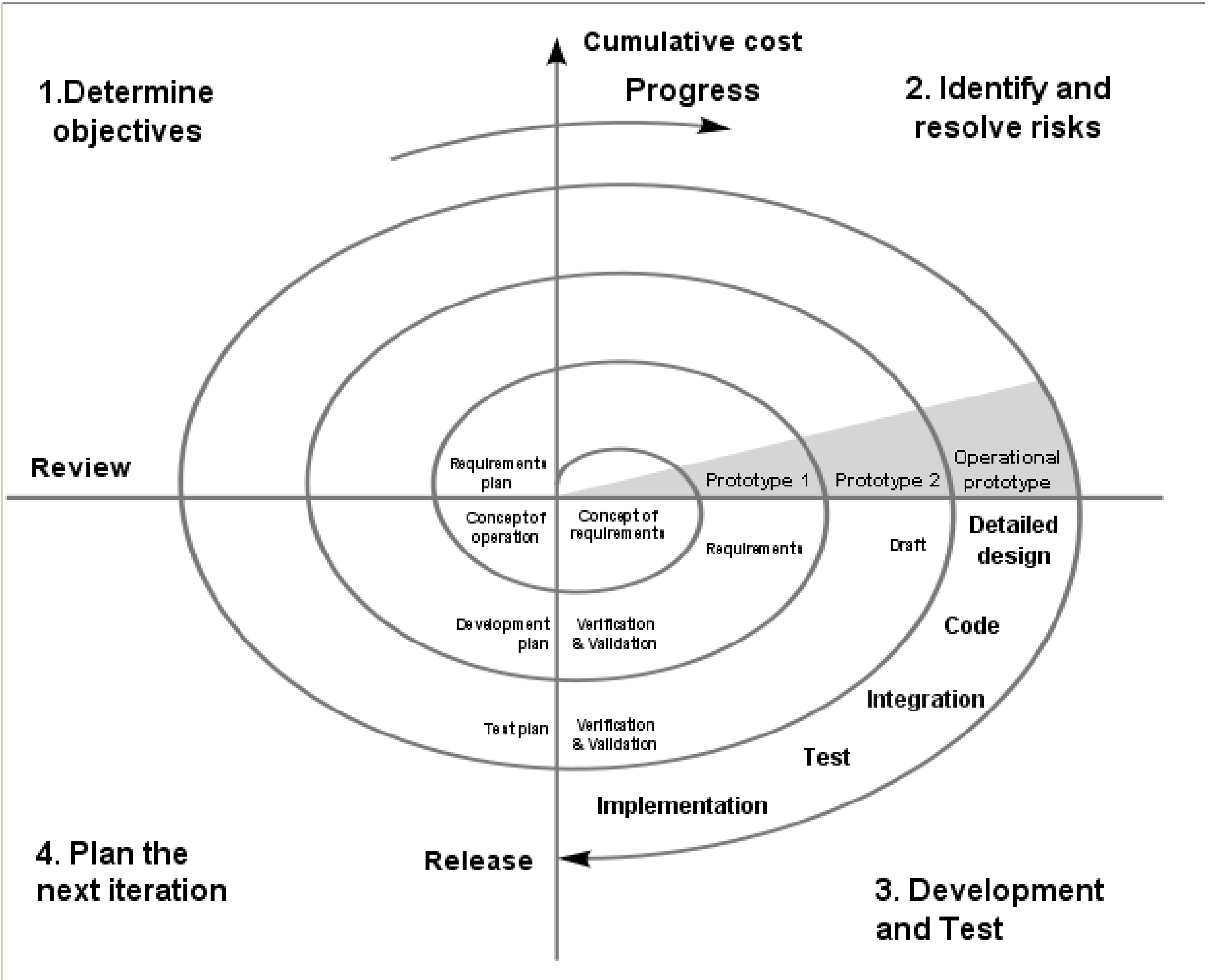


The spiral model (Boehm)

Apply a waterfall-like approach to successive prototypes



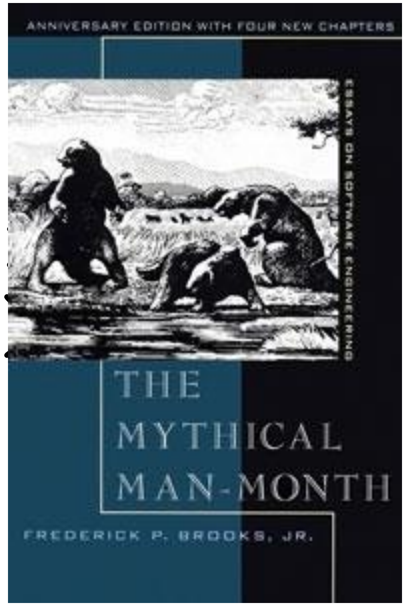
The Spiral model



“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*, “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 20th-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:

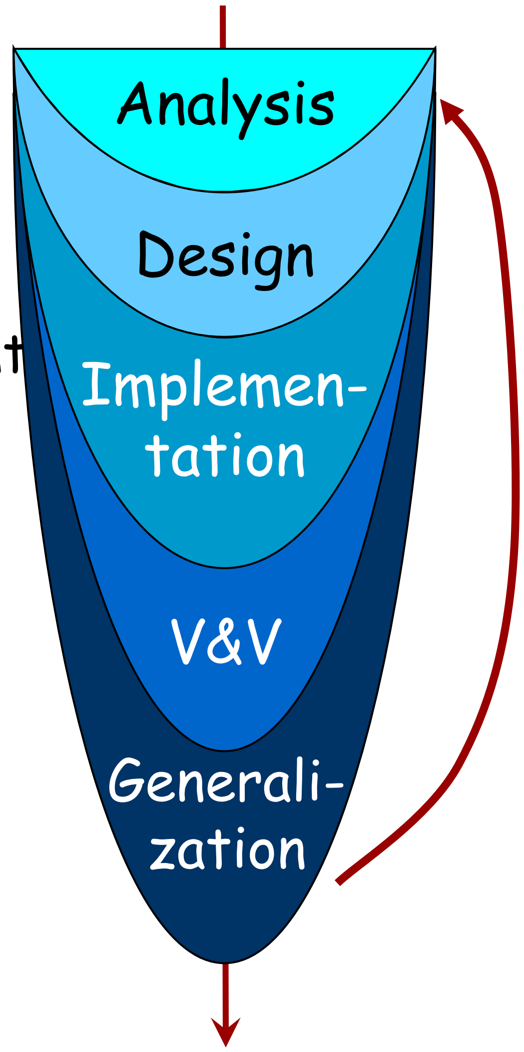
- Single set of notation, tools, concepts, principles throughout
- Continuous, incremental development
- Keep model, implementation and documentation consistent

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…

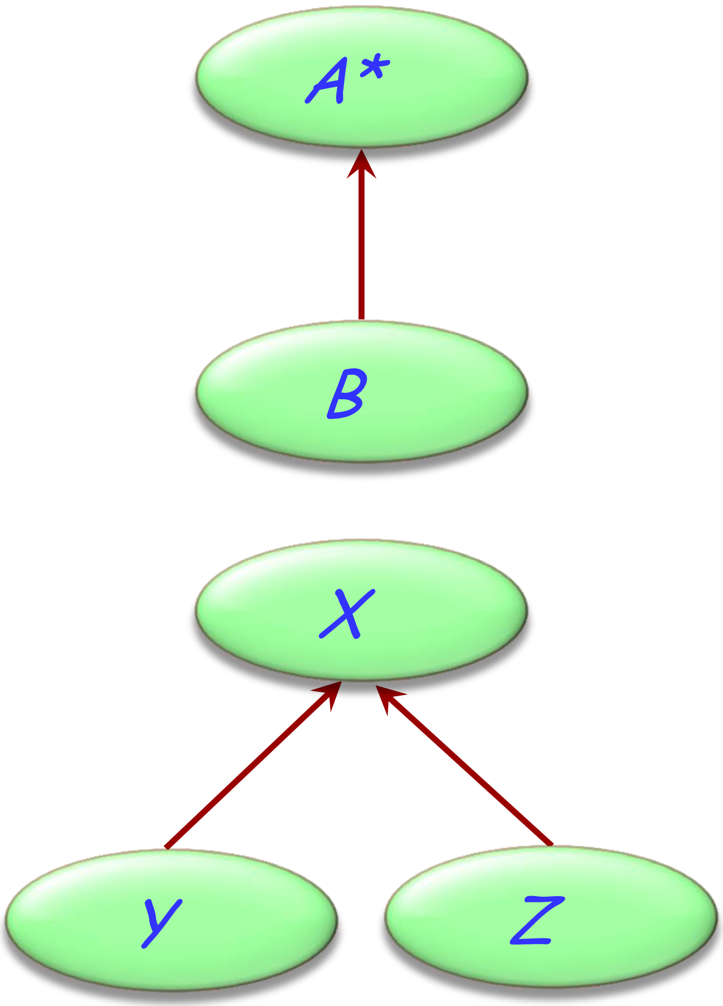
TABLE…

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

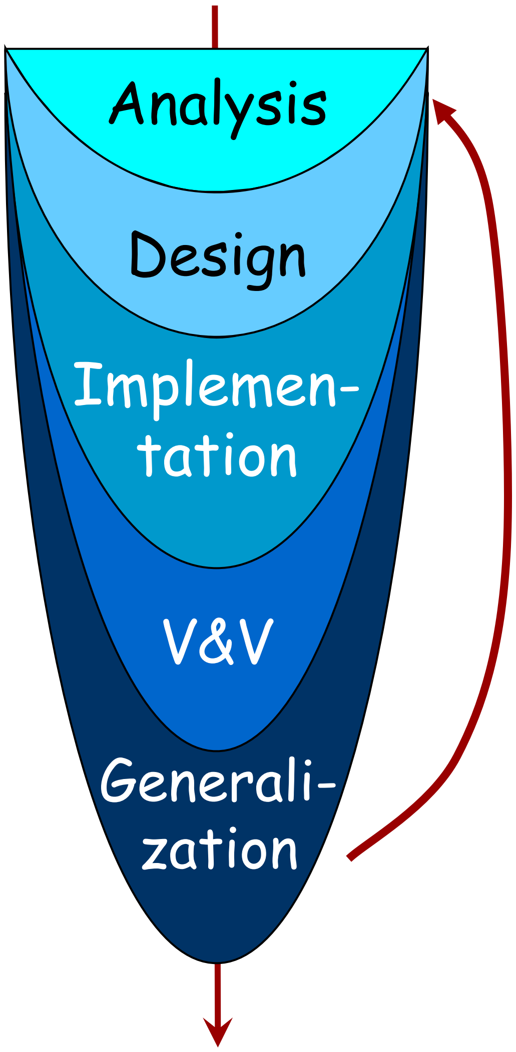
Il semble que tout l'effort industriel de l'homme, tous ses calculs, toutes ses nuits de veille sur les épures, n'aboutissent [...] qu'à la seule simplicité, comme s'il fallait l'expérience de plusieurs générations pour dégager peu à peu la courbe d'une colonne, d'une carène, ou d'un d'avion, jusqu'à 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)

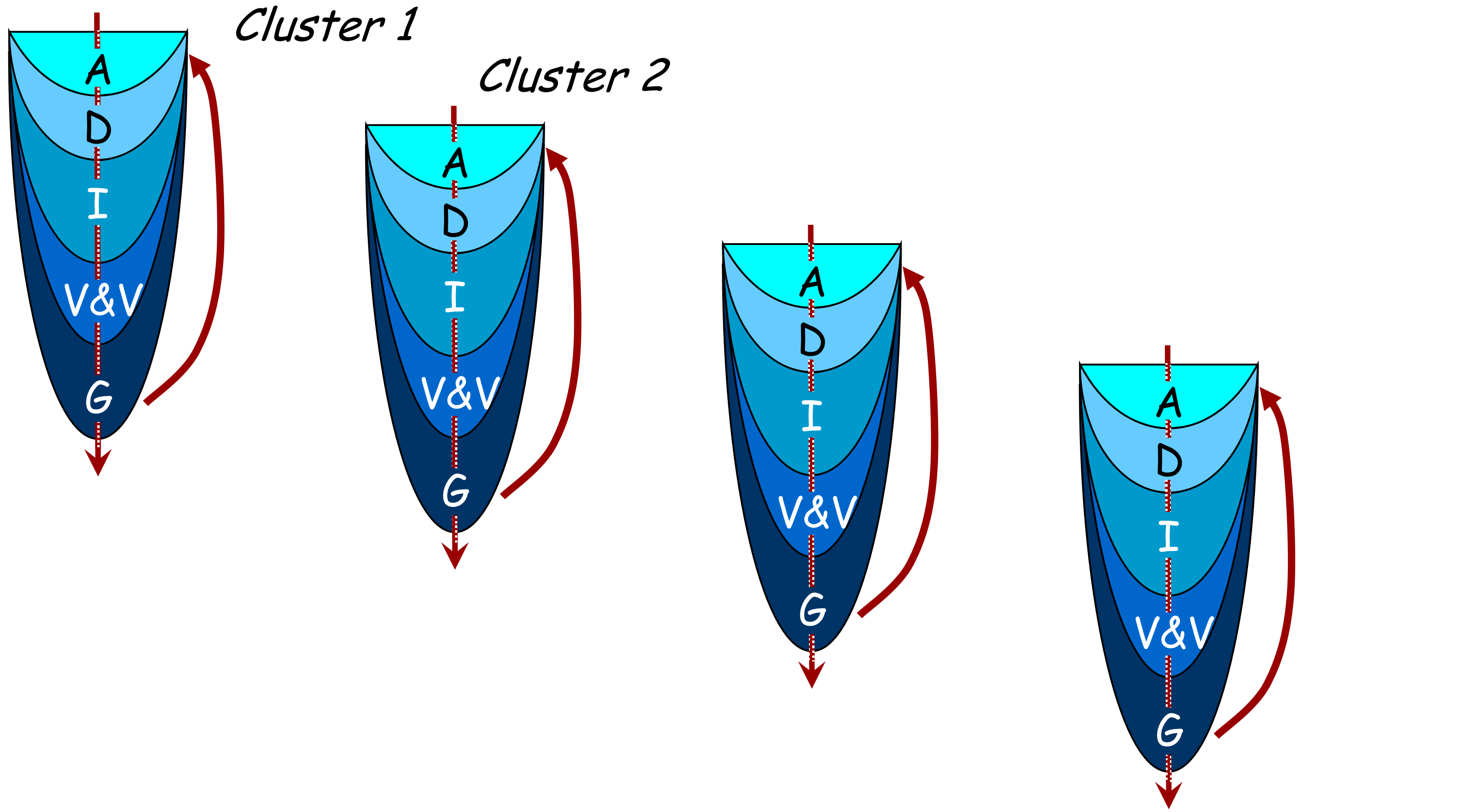
Steve Jobs, 1998

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.

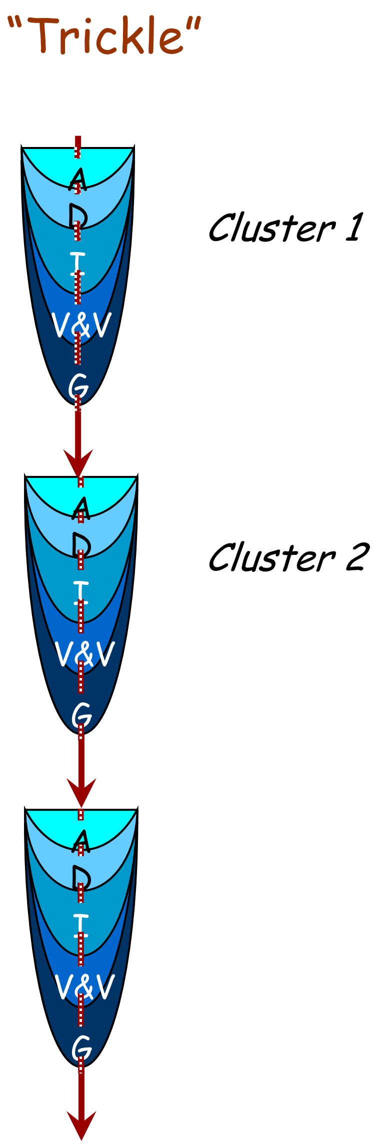
Reversibility

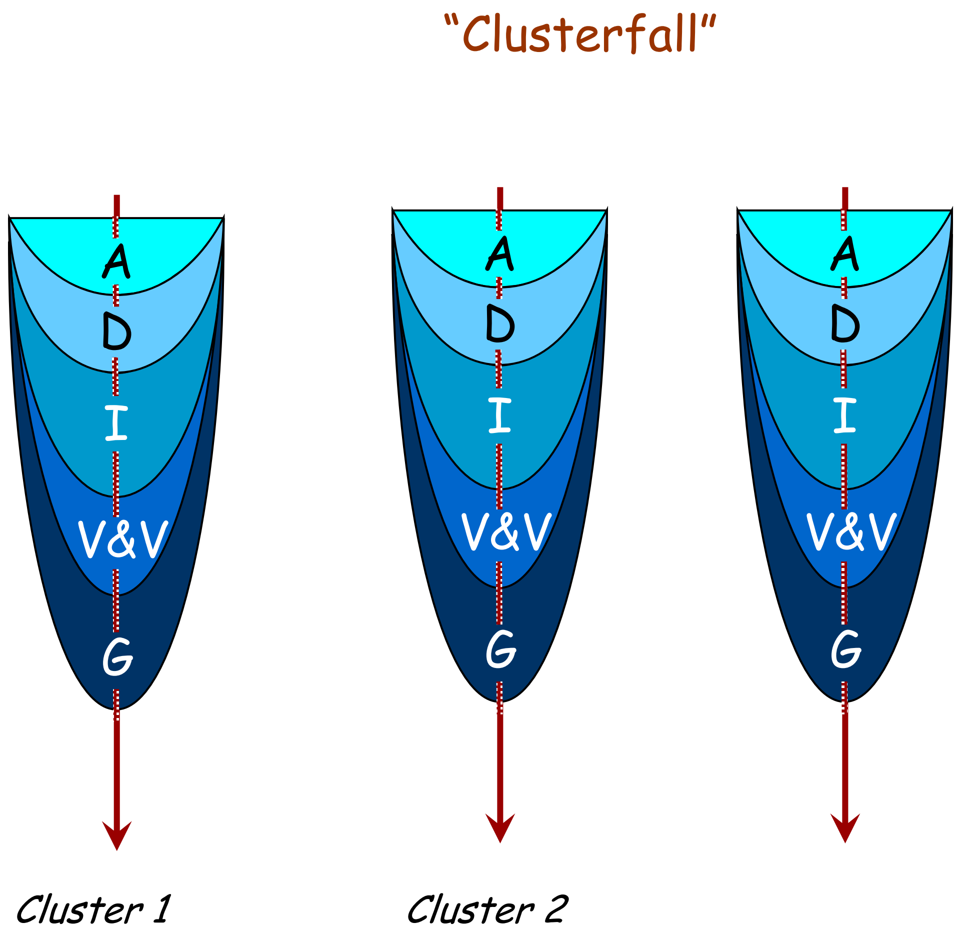


The cluster model

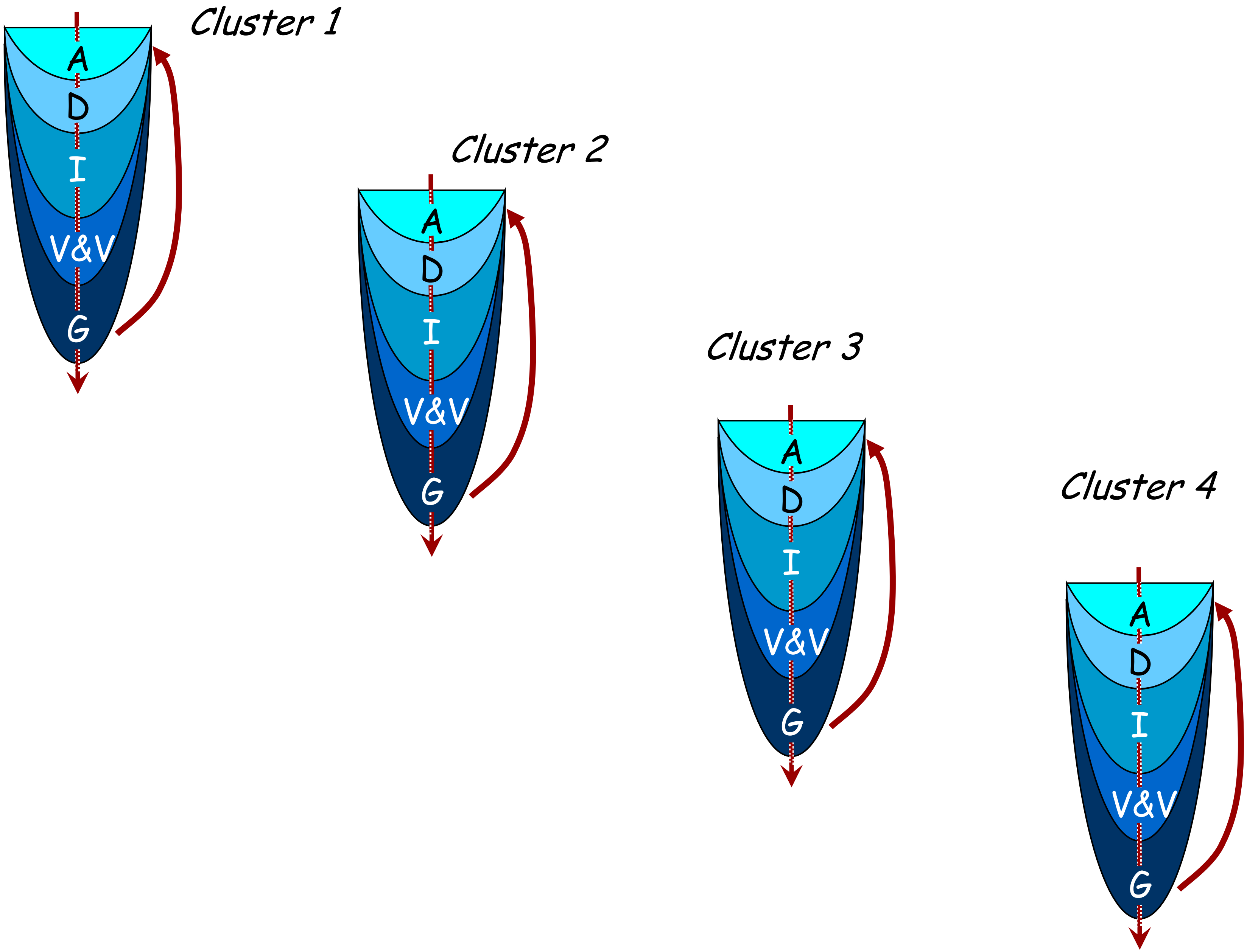


Extremes

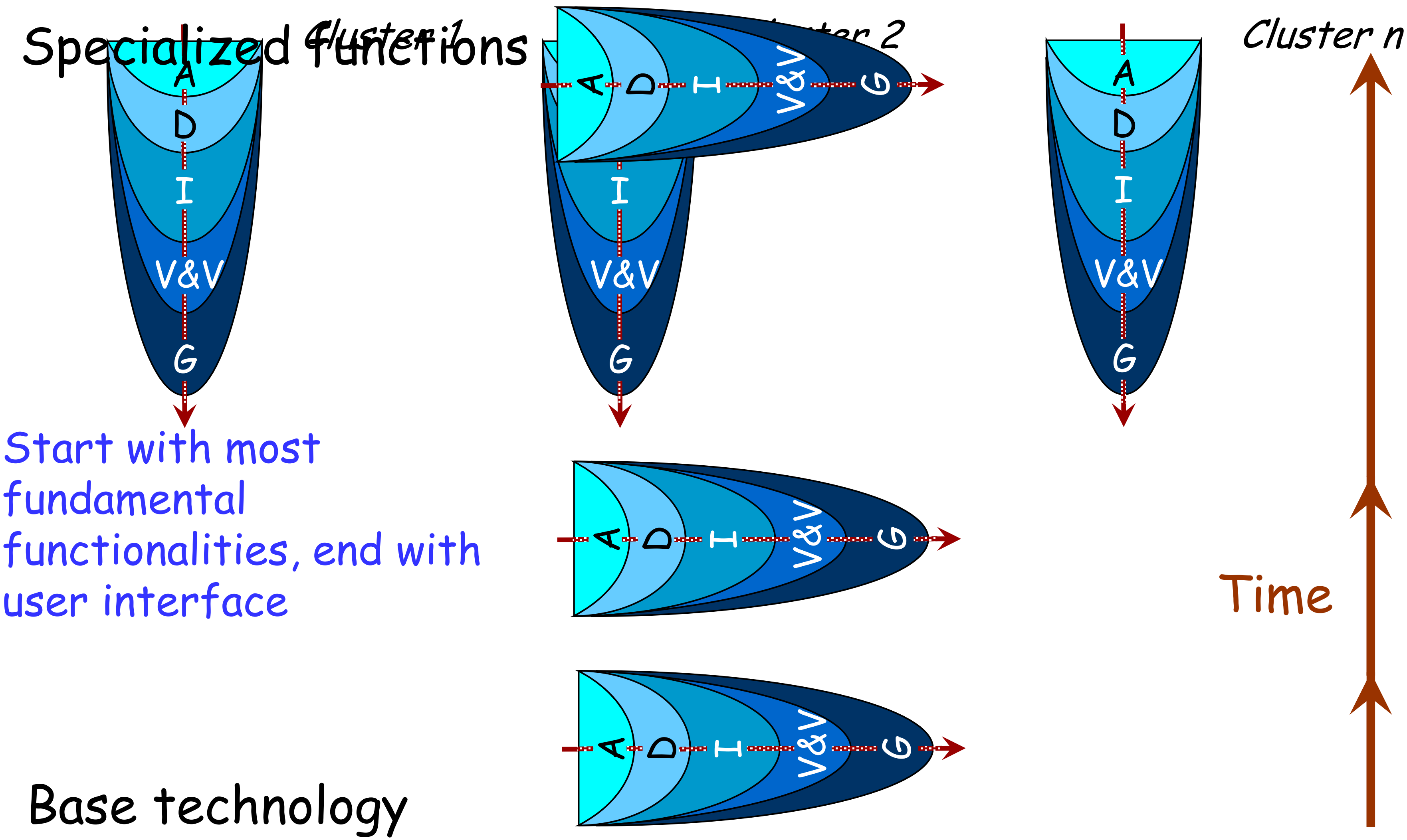

"Trickle"
A
D
I
V&V
G
Cluster 1
A
D
I
V&V
G
Cluster 2
A
D
I
V&V
G


"Clusterfall"
A
D
I
V&V
G
A
D
I
V&V
G
A
D
I
V&V
G
Cluster 1
Cluster 2

Dynamic rearrangement



Bottom-up order of cluster development



Start with most fundamental functionalities, end with user interface

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