Software Engineering

Introduction

History

The process

Process and product

Process and product qualities

Definitions

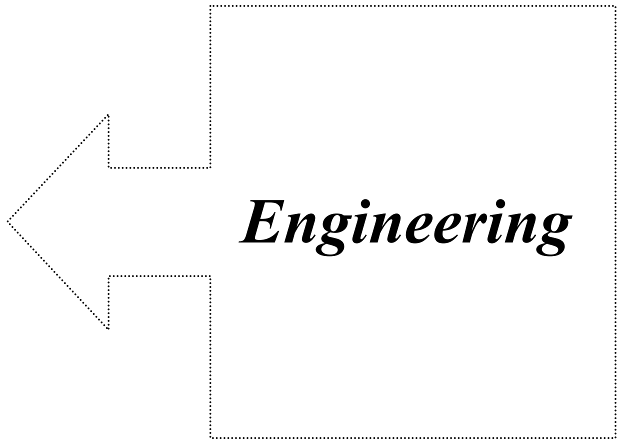
- Field of computer science dealing with software systems
  - large and complex
  - built by teams
  - exist in many versions
  - last many years
  - undergo changes
- Multi-person construction of multi-version software

Definitions

- Systematic approach to development, operation, maintenance, deployment, retirement of software
- Methodological and managerial discipline concerning the systematic production and maintenance of software products that are developed and maintained within anticipated and controlled time and cost limits

Definitions

- Deals with cost-effective solutions to practical problems by applying scientific knowledge in building software artifacts in the service of mankind
  - *cost-effective*
  - *practical problems*
  - *scientific knowledge*
  - *building things*
  - *service of mankind*

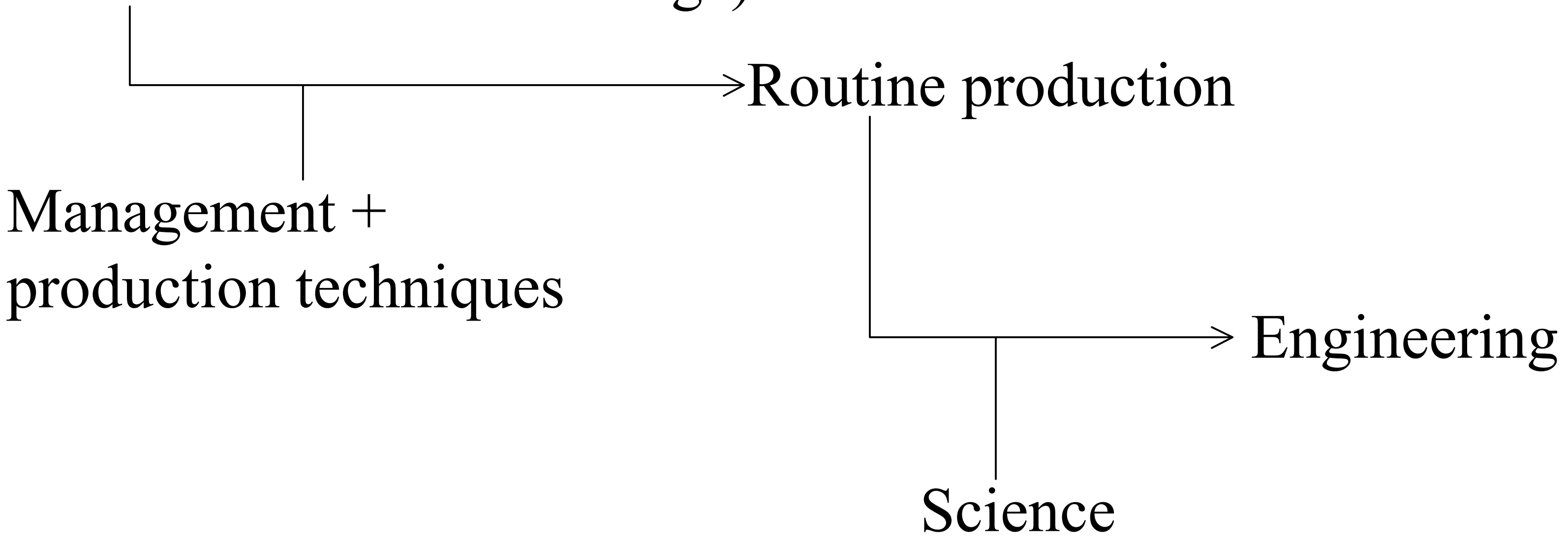


Engineering tasks

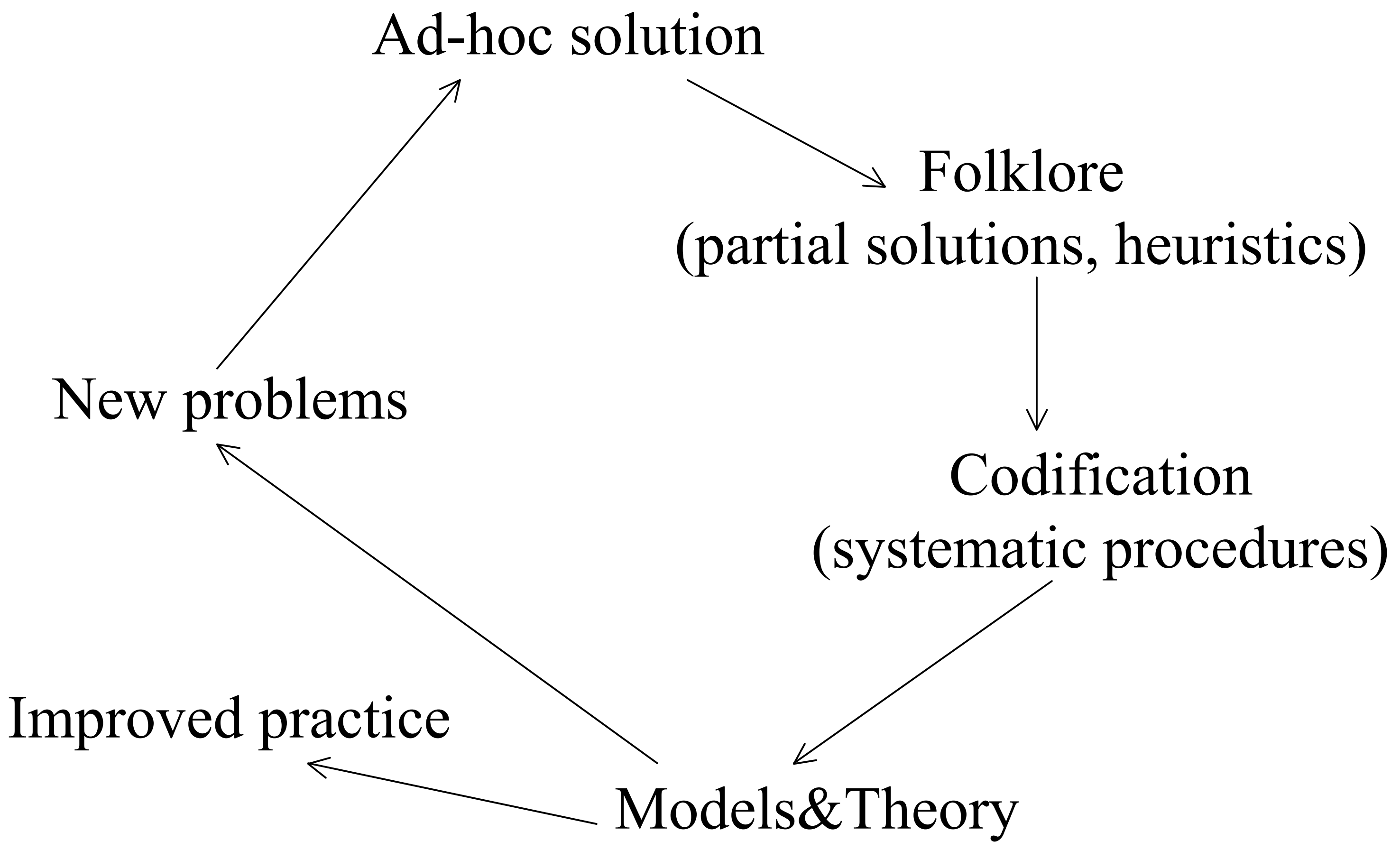
- Routine design
  - solution of familiar problems
  - reuse of previous solutions
- Innovative design
  - novel solutions for unfamiliar problems
- Software is treated more often as original than routine
  - we do not capture and organize what we know!
  - mature engineering disciplines capture, organize and share design knowledge

Engineering: evolution

Art: ad-hoc solutions
- intuition
- talented amateur
- invent/reinvent
  (slow transmission of knowledge)



The engineering lifecycle



Software engineering vs traditional engineering

- Software engineering still practiced (and taught) in a non systematic way
- Less stable and organized than traditional engineering
- Standards for specification of software designs do not exist yet
  - see design of electronic apparatus (amplifier)
  - use of design notation and model
    - for communication and proof of concept

Differences and analogies

- Software engineering vs traditional engineering
- Many points are in common
  - But must be careful because deep differences exist as well!

Bridge building vs. software development

- Bridges normally built on-time, on-budget, and do not fall
- Software is rarely on-time and on-budget

Why?

Traditional engineering

- Extreme detail of design
- Alternative designs can be validated through models
- Design is frozen and contractor has almost no flexibility in changing specifications
- Standard processes are followed

Software engineering: *difference1*

- Frozen specifications and designs do not accommodate changes in the business practices
- Software is a key component of applications and services that are at the heart of business
- Continuous change and evolution

Software engineering: *difference 2*

- Bridge design is 3000 years old
- Large body of knowledge: underlying theories and methods
  - We are moving in this direction
- Large body of knowledge: analysis of failures
  - In sw we seldom analyze failures!

Software engineer: required skills

- Programming skill not enough
  - programmer
    - develops a complete program
    - works on known specifications
    - works individually
  - software engineer
    - identifies requirements and develops specifications
    - designs a component to be combined with other components, developed, maintained, used by others; component can become part of several systems
    - works in a team

Required skills

- Software implements a machine that interacts with the external environment
- Sw engineers must be able to understand and analyze external environments
- The external environment is where the requirements can be found
- *see later discussion on requirements engineering*

Examples

- A telephone system

  when one picks up the phone, the tone must be heard within x msec.

  From requirements and knowledge of the environment one derives a specification of the application
- Other examples:
  - –traffic control systems
  - –health control systems
  - –banking systems

Domain knowledge

- Plays a fundamental role in sw development
- To develop a flight control system, one must understand how an aircraft works
- Software developed based on wrong domain assumption can generate disasters

Skills

- Technical
- Project management
- Cognitive
- Enterprise organization
- Interaction with different cultures
- Domain knowledge

The quality of human resources is of primary importance

History: initial situation

- Software is *art*
- Computers used for "computing"
  - mathematical problems
  - designers = users
  - no extended lifetime
- The art of "programming"
  - *low level languages*
  - *resource constraints (speed and memory)*

From art to craft

- From computing to information management
- Requests for new (custom) software explode
  - users ≠ designers
  - EDP centers and software houses
- New high level languages
- First large projects and first fiasco's
  - time and budget, human cooperation failures
  - wrong specifications

Need for software engineering

- Development methods and standards
- Planning and management
- Automation
- Verifiable quality
- Componentization
- ….

From art to industry

Term “software engineering” defined in a NATO conference in Garmisch, Oct. 1968

Large systems--critical systems

- Examples of large systems
  - Space shuttle project
    - 5.6 million code lines, 22k man year, 1200 Million$
  - CityBank teller machine
    - 780000 code lines, 150 man year, 13.2 Million$
- Critical systems
  - failures generate risks or losses (financial, life)

The engineering approach should improve quality

$ size of applications

Average development cost for

- Large company
  - 2,3 million $
- Medium company
  - 1,3 million $
- Small company
  - 430.000 $

Relevance

- In 1996 software was the third industrial sector after car and electronics
  - More than 250 billion$ a year spent on application development
- In the EU the wider field of “information technology” is viewed as the key strategic sector
  - focused research funding within the 4th and 5th framework programs

Needs

- Recent (2000) study by Microsoft with IDC and Datamonitor
  - *In the year 2003 in Europe there will be a deficit of 1.700.000 people in IT*
  - *Requests exceed availability by 33%*

Understand and manage software lifecycle

LIFECYCLE

- From the problem to the product and its deployment and evolution until its retirement
- However: not all software goes through the complete lifecycle

Project failures

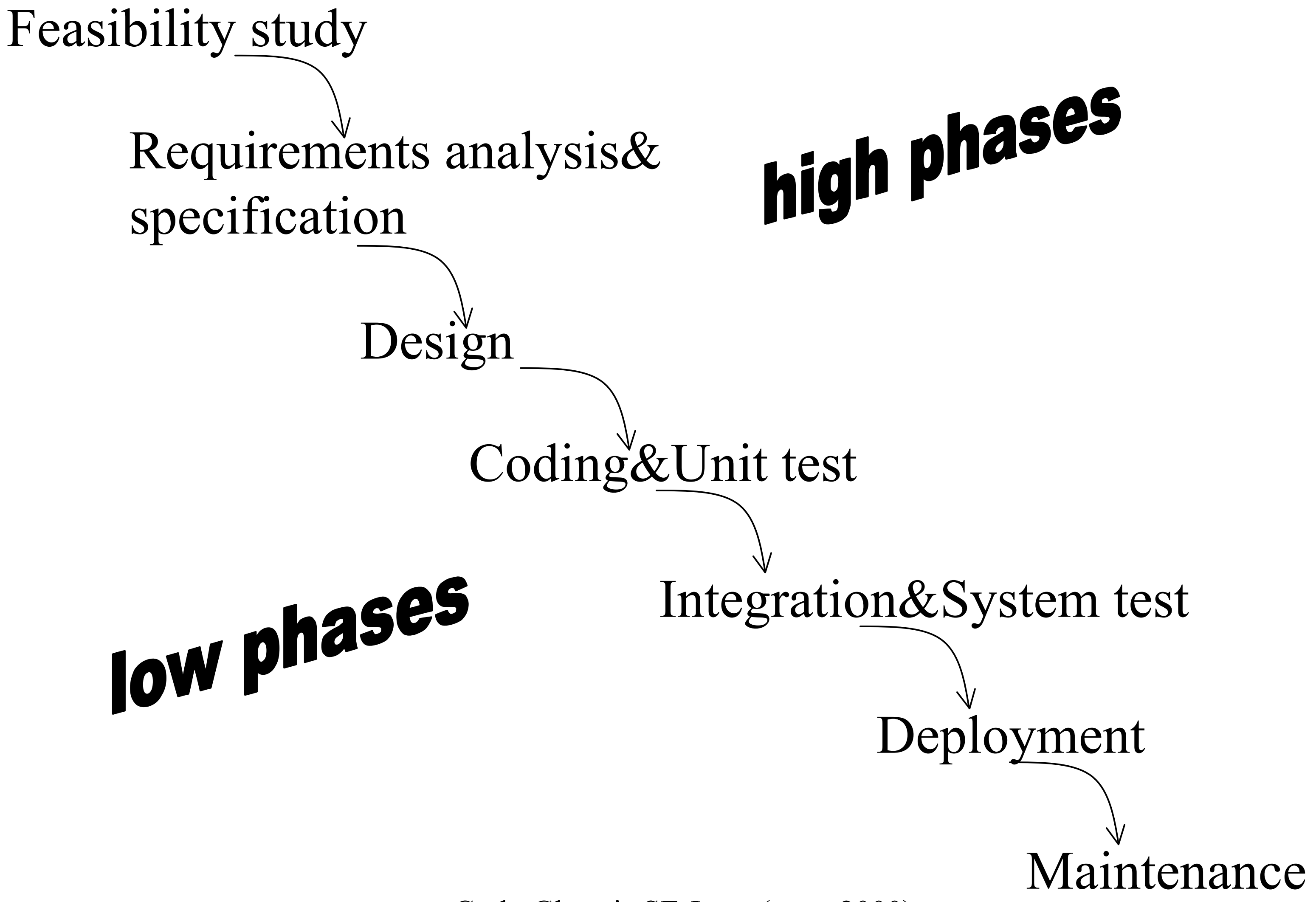
- Almost 30% of development projects are canceled before they are completed!
- About 53% of projects cost 190% of original estimates
  - And hidden costs due to lost opportunities!!!
  - Example: failure to produce reliable software for baggage handling at Denver airport cost over 1 million $ per day!

Software lifecycle models

- In some cases, no reference model:
  - code&fix
- The traditional “waterfall” model
  - identify phases and activities
  - force linear progression from a phase to the next
  - no returns (they are harmful)
    - better planning and control
  - standardize outputs (artifacts) from each phase

software like manufacturing

A waterfall model



Feasibility study

- Cost/benefits analysis
- Determines whether the project should be started (e.g., buy vs make), possible alternatives, needed resources
- Produces a Feasibility Study Document
  - Preliminary problem description
  - Scenarios describing possible solutions
  - Costs and schedule for the different alternatives

Req. analysis and specification

- Analyze the domain in which the application takes place
- Identify requirements
- Derive specifications for the software
  - Requires an interaction with the user
  - Requires an understanding of the properties of the domain
- Produces a Requirements Analysis and Specification Document (RASD)

The 5 W's

- Who
    - who will use the system
- Why
    - why should it be developed + why will the users use it
- What (vs How)
    - what will it provide
- Where
    - where will it be used, on which architecture
- When
    - when and how long will it be used

RASD

- Required properties
  - Precise
  - Complete
  - Consistent
- May include
  - Preliminary User Manual
  - System Test Plan

Design

- Defines the software architecture
  - Components (modules)
  - Relations among components
  - Interactions among components
- Goal
  - Support concurrent development, separate responsibilities
- Produces the Design Document

Coding&Unit test

- Each module is implemented using the chosen programming language
- Each module is tested in isolation by the module’s developer
- Programs include their documentation

Integration&System test

- Modules are integrated into (sub)systems and integrated (sub)systems are tested
- This phase and the previous may be integrated in an incremental implementation scheme
- Complete system test needed to verify overall properties
- Sometimes we have *alpha test* and *beta test*

Effort distribution

- From 125 projects within HP
  - 18% requirements and specification
  - 19% design
  - 34% coding
  - 29% testing
- typical variations of 10%

Deployment

- The goal is to distribute the application and manage the different installations and configurations at the clients’ sites

Maintenance

- All changes that follow delivery
- Unfortunate term: software does not wear out
  - if a failure occurs, the cause was there
- Often more than 50% of total costs
  - Recent survey among EU companies
    - 80% of IT budget spent on maintenance

Maintenance

- It includes different types of change: correction + evolution
  - corrective maintenance $\approx$ 20%
  - adaptive maintenance $\approx$ 20%
  - perfective maintenance $\approx$ 50%

Folk data on errors

- Systematic inspection techniques can discover up to 50-75% of errors
- Modules with complex control flow are likely to contain more errors
- Often tests cover only about 50% of code
- Delivered code contains 10% of the errors found in testing
- Early errors are discovered late, and the cost of removal increases with time
- Eliminating errors from large and mature systems costs more (4-10 times) than in the case of small and new systems
- Error removal causes introduction of new errors
- Large systems tend to stabilize to a certain defect level

Why evolution?

- Context changes (adaptive maintenance)
  - EURO vs national currencies
- Requirements change
    - New demands caused by introduction of the system
    - Recent survey among EU companies indicates that 20% of user requirements are obsolete after 1 year
- Wrong specifications (rqmts were not captured correctly or domain poorly understood)
- Requirements not known in advance

How to face evolution

- Likely changes must be anticipated
- Software must be designed to accommodate future changes reliably and cheaply

This is one of the main goals of software engineering

Correction vs evolution

- Distinction can be unclear, because specifications are often incomplete and ambiguous
- This causes problems because specs are often part of a contract between developer and customer
  - early frozen specs can be problematic, because they are more likely to be wrong

Software changes

- Good engineering practice
  - first modify design, then change implementation
  - apply changes consistently in all documents
- Software is very easy to change
  - often, under emergency, changes are applied directly to code
  - inconsistent state of project documents

software maintenance is (almost) never anticipated and planned; this causes disasters

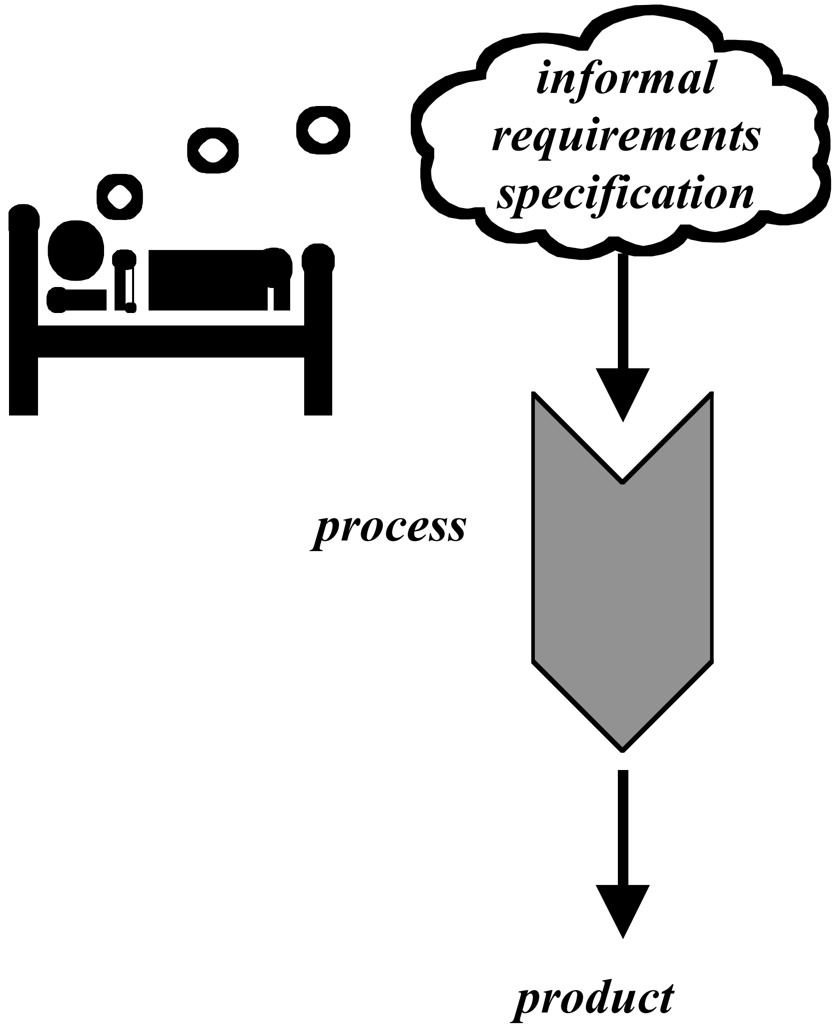
Waterfall lifecycles

- Many variations exist
- Each organization tends to define “its own”
- Sample cases
  - software developed for personal use
  - customer (user) belongs to same organization
  - custom software developed by sw house
  - application for the market

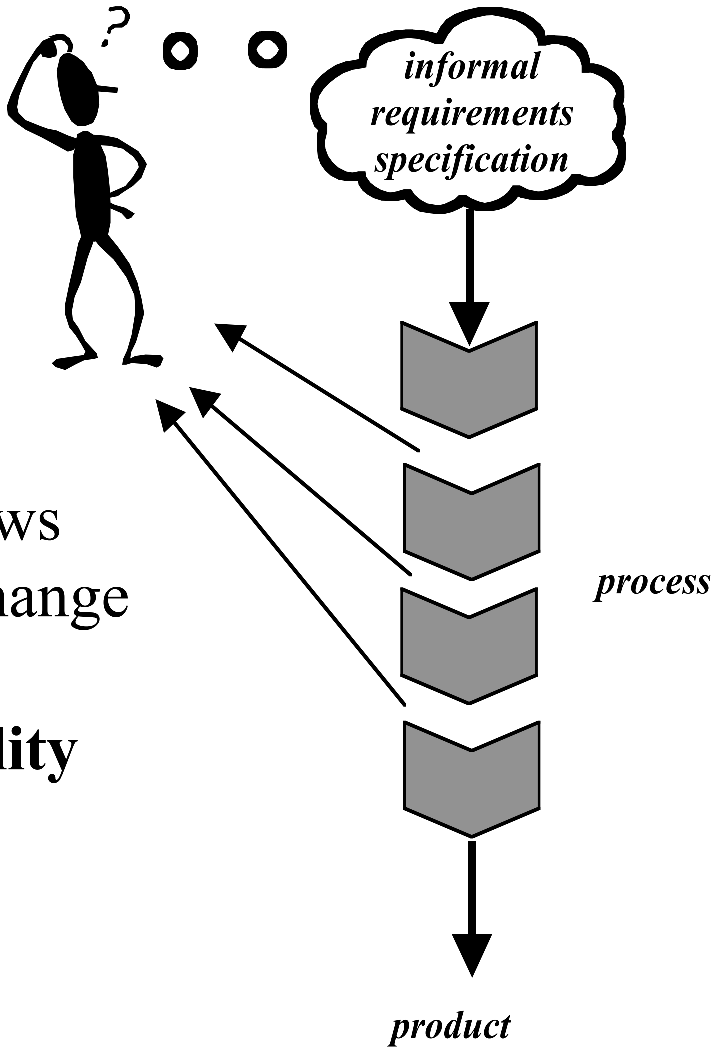
Waterfall can be harmful

- “Waterfall” requires that the domain is understood and requirements are known and stable
- This happens in only a few cases
- Recycling cannot be eliminated; it is part of our problem

Waterfall is “black box”

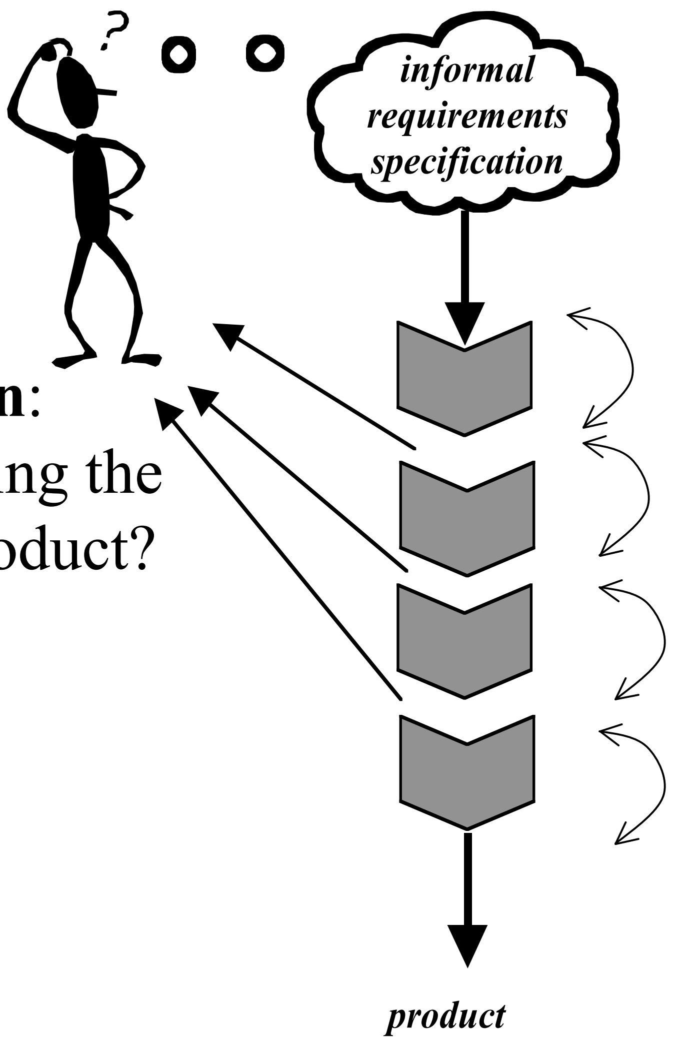


Need for transparency



- Transparency allows early check and change via **feedback**
- It supports **flexibility**

Verification and validation



Validation:
are we doing the
“right” product?

Verification:
are we doing
the product right?

Flexible processes

- Adapt to changes, in particular in the requirements and specification
- The idea is to have incremental processes and be able to get feedback on increments
- Exist in many forms

Prototyping

- A prototype is an approximate model of the application, used to get feedback or prove some concept
- “What” to prototype depends on what is critical to assess (e.g., user interface)
- Throw-away vs evolutionary prototype

Incremental delivery

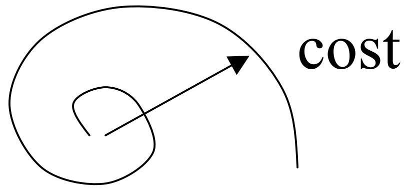
- Early subset, early delivery, … early feedback
- Start from critical subsets, on which feedback is required from customer

New products and lifecycle

- Incrementality even more important for new types of product
- Beta versions available for try-out
- The network as a distribution medium and a showcase

A meta-model: spiral model

- Cycle among the following activities:
  - risk analysis
  - development
  - validation
- As you cycle, cost increases
  - as in a spiral



Lifecycle and activities

- Lifecycles differ mainly in the way and the order in which activities are performed
- One does analysis and specification, design, code, test… also in a flexible lifecycle
- The way and (in particular) the sequencing among them differs

Process and product

- Our goal is to develop software products
- The process is how we do it
- Both are extremely important, due to the nature of the software product
- Both have qualities
  - in addition, quality of process affects quality of product

The software product

- Different from traditional types of products
  - intangible
    - difficult to describe and evaluate
  - malleable
  - human intensive
    - does not involve any trivial manufacturing process

Quality dimensions

Process vs. Product

Internal vs. External

Internal qualities affect external qualities
Process quality affects product quality

Correctness

- Software is correct if it satisfies the specifications
- If specifications are stated formally, since programs are formal objects, correctness can be defined formally
  - It can be proven as a theorem or disproved by counterexamples (testing)
- Try to develop “a priori correct” sw
    - via suitable process (see later)
    - suitable tools (high level languages, reuse)

The limits of correctness

- It is an absolute (yes/no) quality
  - there is no concept of “degree of correctness”
  - there is no concept of severity of deviation
- What if specification are wrong?
  - (e.g., they derive from incorrect requirements or errors in domain knowledge)

Reliability, robustness

- Reliability
  - informally, user can rely on it
  - can be defined mathematically as "probability of absence of failures for a certain time period"
  - if specs are correct, all correct software is reliable, but not vice-versa (in practice, however, specs can be incorrect …)
- Robustness
  - software behaves "reasonably" even in unforeseen circumstances (e.g., incorrect input, hardware failure)

Performance

- Efficient use of resources
  - memory, processing time, communication
- Can be verified
  - complexity analysis
  - performance evaluation (on a model, via simulation)
- Performance can affect scalability
  - a solution that works on a small local network may not work on a large intranet
- Performance can affect usability (see next)
- Performance may change with technology

Usability

- Expected users find the system easy to use
- Important: define the *expected user*
    - if the user is an installer, ease of installation is part of usability
- Other terms: ergonomic, user friendly
- Rather subjective, difficult to evaluate
- Evaluation made less subjective via panels
- Affected mostly by *user interface*
    - e.g., visual vs textual

Other qualities

- Maintainability
- Reusability
  - similar to maintainability, but applies to components
- Portability
  - similar to maintainability (adaptation to different target environment)
- Interoperability
  - coexist and cooperate with other applications

Process qualities: productivity

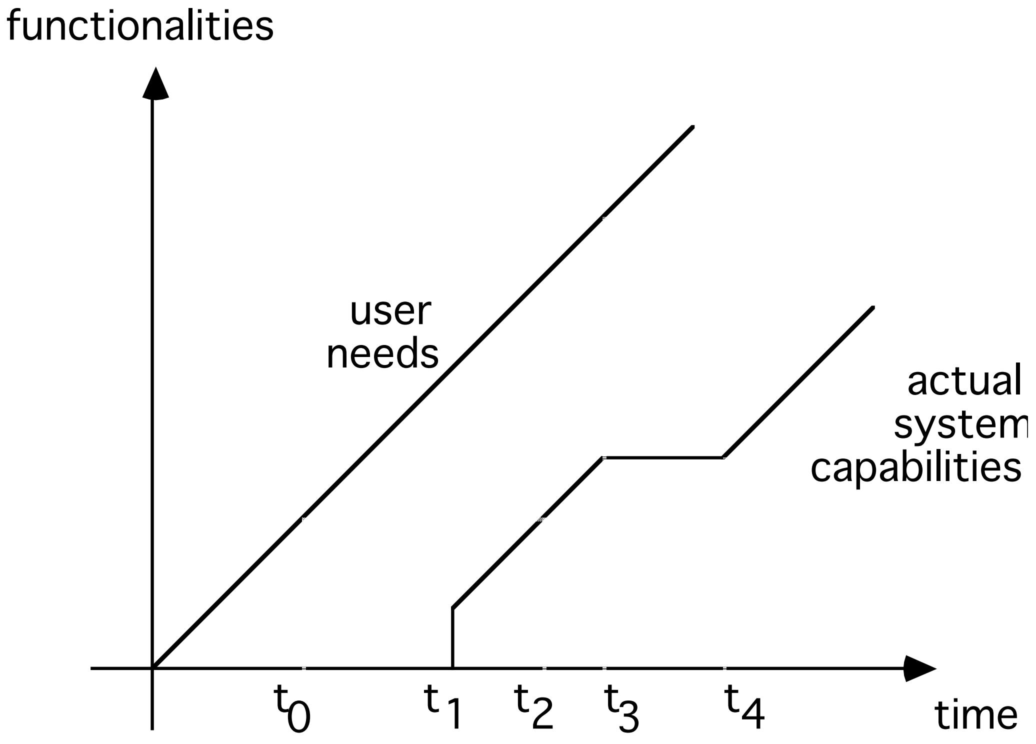
- Productivity
  - how can we measure it?
    - delivered item by a unity of effort
- Unity of effort
  - person month
    - WARNING: persons and months cannot be interchanged
- Delivered item
  - lines of code (and variations)
  - function points

Productivity: folk data

- Result of 135 HP projects (excluding requirements): 350 NCSS/pm
  - NCSS: Non Comment Source Statement
  - pm: person month
- BUT
  - extreme variance among individuals
  - extreme variance with group dynamics
    - Brooks "law":
      - "Adding people to a late project makes the project late"

Process qualities: timeliness

- Ability to respond to change requests in a timely fashion



Key SE principles

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality

Reference

C. Ghezzi, M. Jazayeri, D. Mandrioli
Fundamentals of Software Engineering, Prentice Hall, 1991