Software Engineering

What is SE?---SE vs. traditional eng.
How did it develop?---History
Software lifecycle---process models
Process and product
Process and product qualities

1

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

2

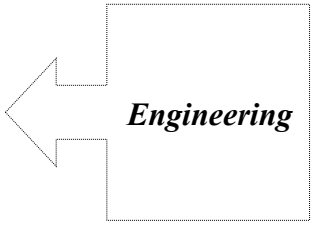
Definitions

- 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
- Application of science and mathematics by which the properties of software are made useful to people (B. Boehm, ICSE 2006)

3

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*



4

Engineering tasks

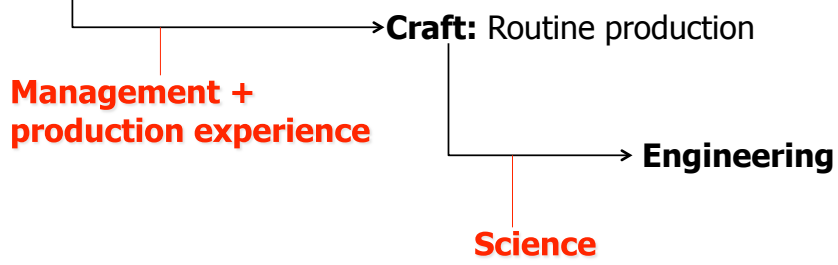
- Routine (normal) design
  - solution of familiar problems
  - reuse of previous solutions
- Innovative (radical) design
  - novel solutions for unfamiliar problems
- Software is treated more often as original than routine
  - we do not capture and organize what we know!
    - mature engineering disciplines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

5

Engineering: the final stage

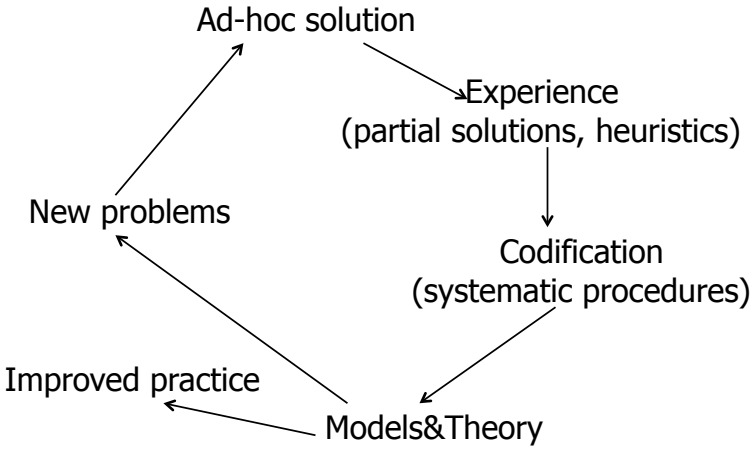
Art: ad-hoc solutions
- intuition
- talented amateur
- invent/reinvent

(slow transmission of knowledge)



6

Another view



7

Software engineering vs "traditional" engineering

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

8

Traditional engineering

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

9

Software engineering: *difference 1*

- Frozen specifications and designs unrealistic in practice
- Continuous change and evolution
  - Because software is a key component of applications and services that are at the heart of business, and business evolves very rapidly

10

The world: continuous change

- From monolithic centralized enterprises
- To dynamic federations of enterprises
- Time to market and customer satisfaction essential
- Technological infrastructures also evolve

11

The world: continuous change

- Embedded systems, pervasive systems
- They live in a changing physical world
- Sometimes the world changes even at run time

12

Software engineering: *difference 2*

- Bridge design is 3000 years old
- Large body of knowledge: underlying theories and methods
- This is true for most engineering fields
- Software engineering is about 40 years old

13

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 the overall architecture
    - designs components to be combined with other components, developed, maintained, used by others; component can become part of several systems
    - works in a team

14

Required skills

- Software implements a machine that interacts with the external environment
- Sw engineers must be able to understand and analyze external environments
  - not enough to be a "Java guru" or "XXX guru"
- The external environment is where the requirements can be found
- *see later discussion on requirements engineering*

15

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

16

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

17

Skills

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

The quality of people is of primary importance

18

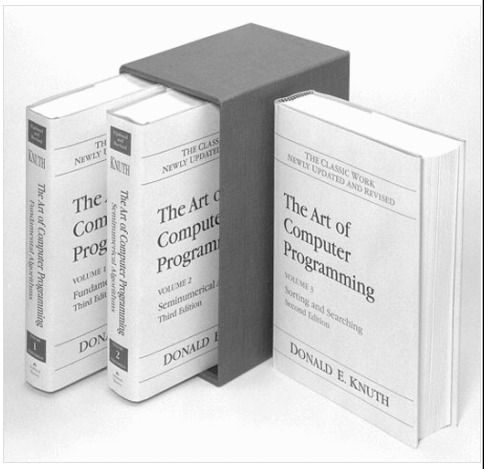
History: initial situation

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

19

Pioneers: D. Knuth

- Project started in 1962
  - volume 1 in 1968
  - volume 2 in 1969
  - volume 3 in 1973



20

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 failures
  - time and budget, human cooperation failures
  - wrong specifications

21

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

22

Vintage picture

23

Large systems—critical systems

- Examples of large systems
  - from several to hundreds man-year
- Critical systems
  - failures generate risks or losses (financial, life)

The engineering approach should improve quality

24

Lessons from Brooks

The mythical man-month

- "Program for yourself" to "program for others" squares the *cost*
- Adding more *people* to a late project delays the project further

25

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!

26

More on failures

Source: Standish Group "Chaos Report" on American software projects

- Only 16% on time & on budget
- At least 94% will have at least 1 restart
- 88% cost overrun
- 22% schedule overrun
- Only 61% have original scope

27

From Standish group

	1994	1998
Successful	16%	26%
Challenged	53%	46%
Cancelled	31%	28%

Perceived causes of successes and failures (Top 3)

	Successful	Challenged	Cancelled
1.	User involvement	Lack of user input	Incomplete reqts
2.	Executive management support	Incomplete reqts	Lack of user input
3.	Clear statement of reqts	Changing reqts	Lack of resources

28

Should we care?

- In 1996 software became the third industrial sector after car and electronics
- In the EU the wider field of "information technology" (also, ICT) is viewed as the key strategic sector
  - focused research funding its framework programs

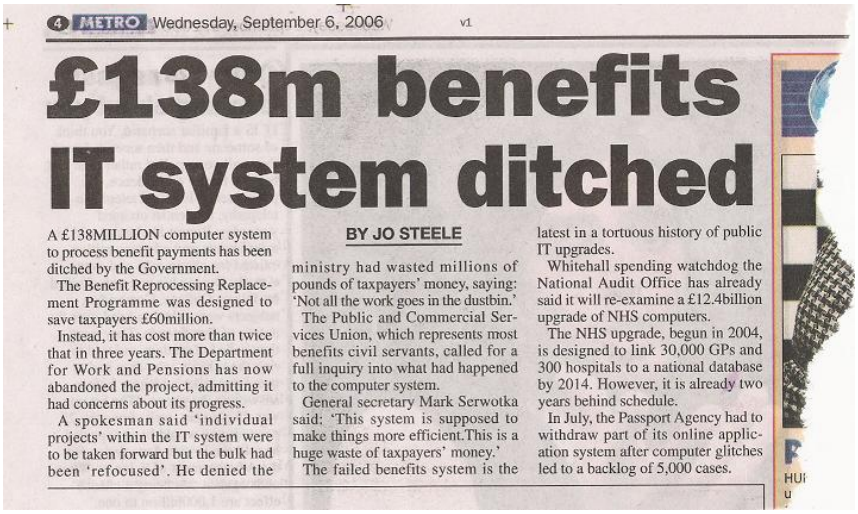
29

A failure example

- Therac-25, 1985-87
  - A computerized radiation therapy machine used in Canada (6 installations) e in USA (5 installations) for cancer therapy
  - A software defect caused massive overdoses of radiations
  - At least 4 persons died and others have been seriously injured

30

Still today…



4 METRO Wednesday, September 6, 2006 v1

£138m benefits IT system ditched

BY JO STEELE

A £138MILLION computer system to process benefit payments has been ditched by the Government.

The Benefit Reprocessing Replacement Programme was designed to save taxpayers £60million.

Instead, it has cost more than twice that in three years. The Department for Work and Pensions has now abandoned the project, admitting it had concerns about its progress.

A spokesman said 'individual projects' within the IT system were to be taken forward but the bulk had been 'refocused'. He denied the ministry had wasted millions of pounds of taxpayers' money, saying: 'Not all the work goes in the dustbin.'

The Public and Commercial Services Union, which represents most benefits civil servants, called for a full inquiry into what had happened to the computer system.

General secretary Mark Serwotka said: 'This system is supposed to make things more efficient.This is a huge waste of taxpayers' money.'

The failed benefits system is the latest in a tortuous history of public IT upgrades.

Whitehall spending watchdog the National Audit Office has already said it will re-examine a £12.4billion upgrade of NHS computers.

The NHS upgrade, begun in 2004, is designed to link 30,000 GPs and 300 hospitals to a national database by 2014. However, it is already two years behind schedule.

In July, the Passport Agency had to withdraw part of its online application system after computer glitches led to a backlog of 5,000 cases.

Software lifecycle

From the problem to the product and its deployment and evolution until its retirement

32

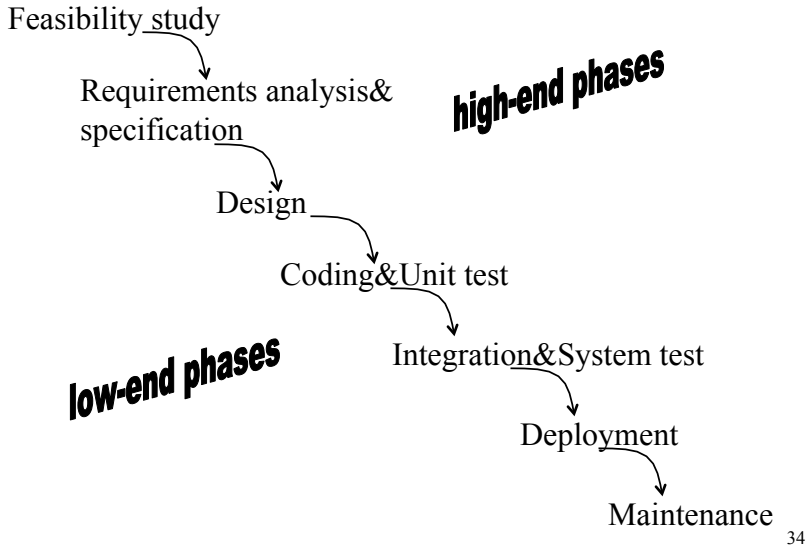
Software lifecycle models

- In some cases, development does not follow any reference model
  - code&fix
- The traditional "**waterfall**" model
  - identify phases and activities
  - force linear progression from a phase to the next
  - no returns (considered harmful)
    - better planning and control
  - standardize outputs (artifacts) from each phase

software like manufacturing

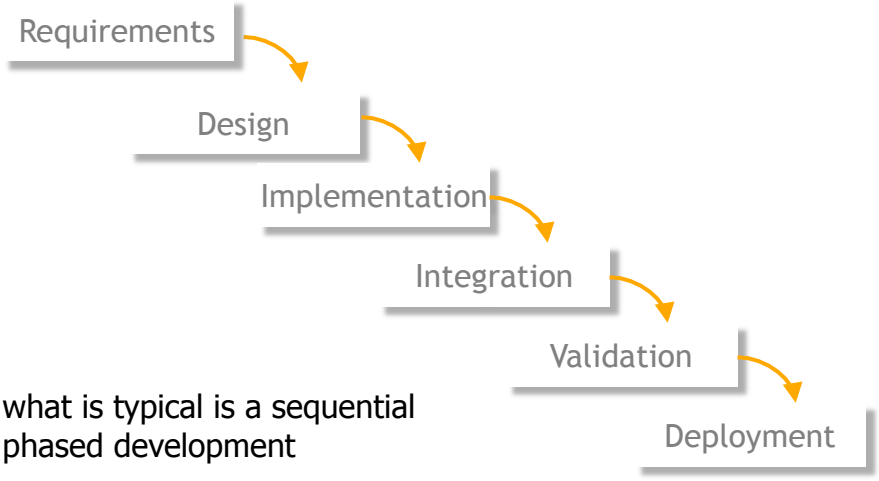
33

A waterfall model (Royce, 1970)



34

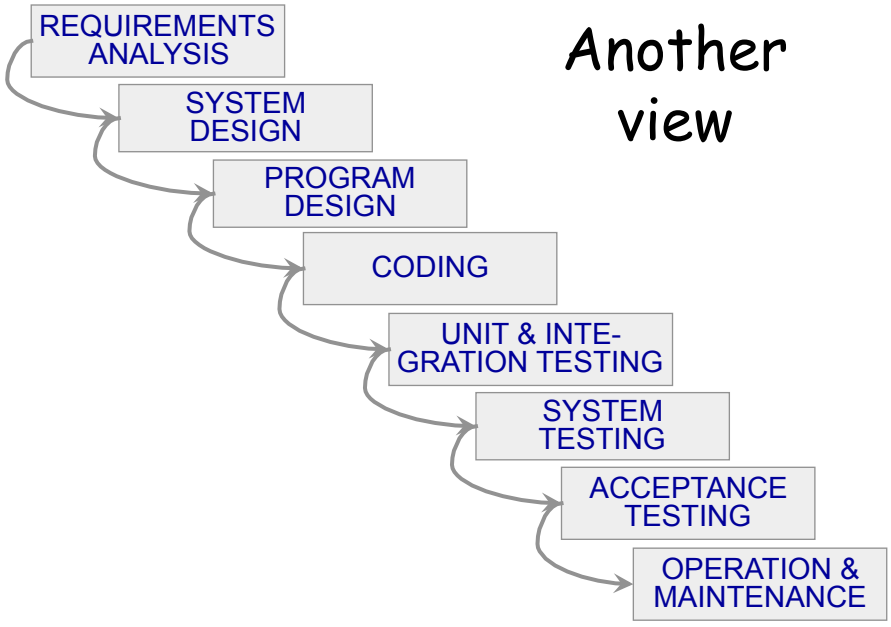
Other views



what is typical is a sequential phased development

35

Another view



36

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

37

Requirements 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)

38

Design

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

39

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

40

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*

41

Effort distribution

- Study in the 1990's of 125 projects within HP
  - 18% requirements and specification
  - 19% design
  - 34% coding
  - 29% testing
- typical variations of 10%

42

Deployment

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

43

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
  - Survey among EU companies
    - 80% of IT budget spent on maintenance

44

Maintenance

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

45

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

46

How to face evolution

- Likely changes must be anticipated
- Software must be designed to accommodate future changes reliably and cheaply
- THIS IS MORE AND MORE VITAL

This is one of the main goals of software engineering

47

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

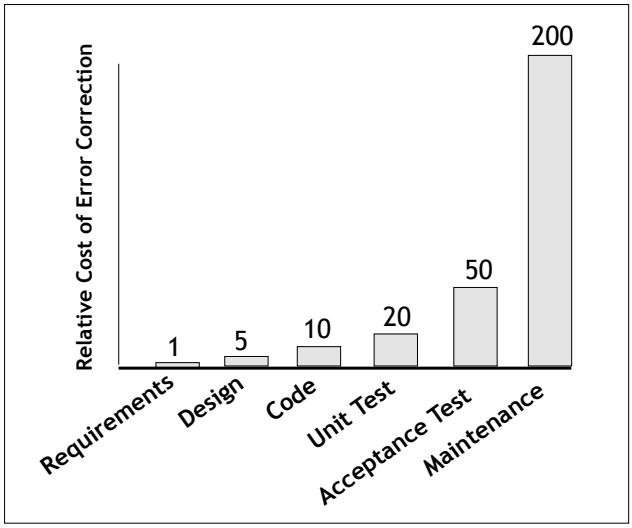
48

Folk data on errors

- 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
- 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

49

Cost of late correction (Boehm, 1981)

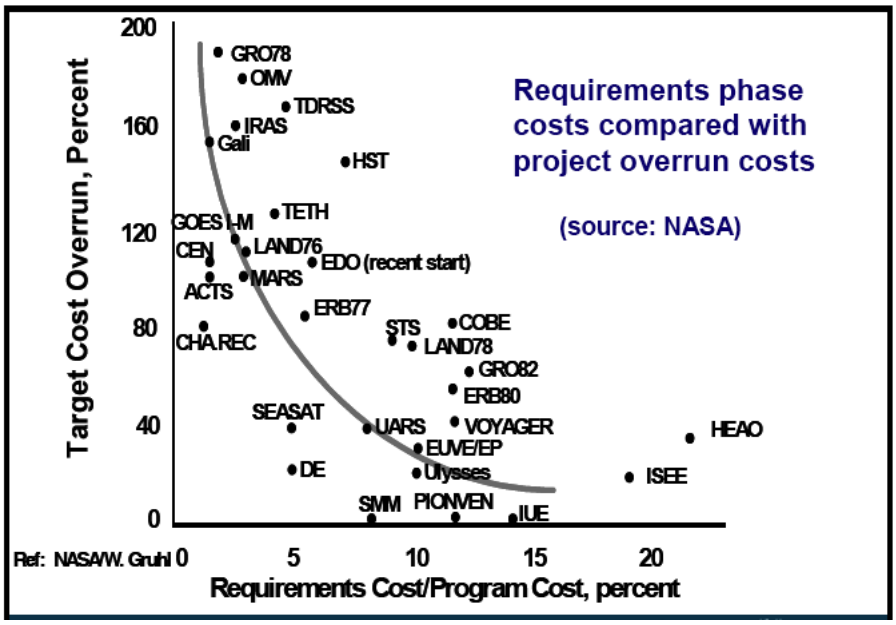


The cost of correcting an error depends on the number of subsequent decisions that are based on it

Errors made in understanding requirements have the potential for greatest cost, because many other design decisions depend on them

50

Why understanding requirements is important



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

52

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

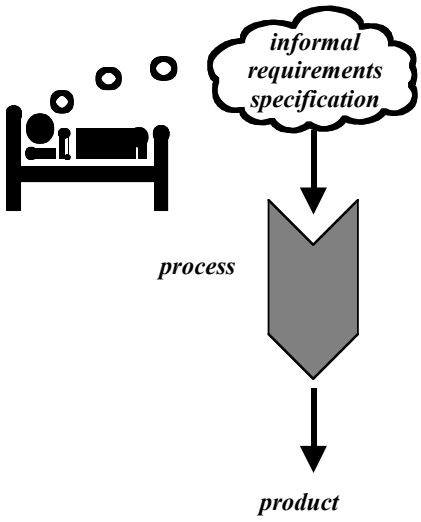
53

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

54

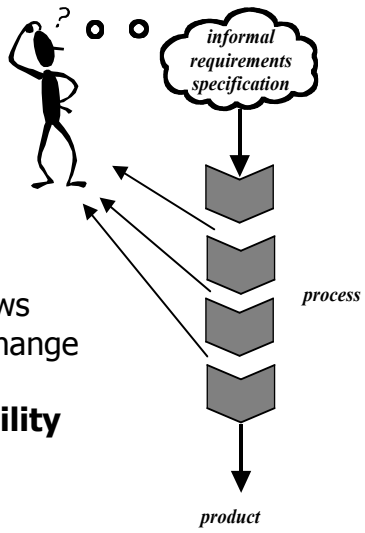
Waterfall is "black box"



55

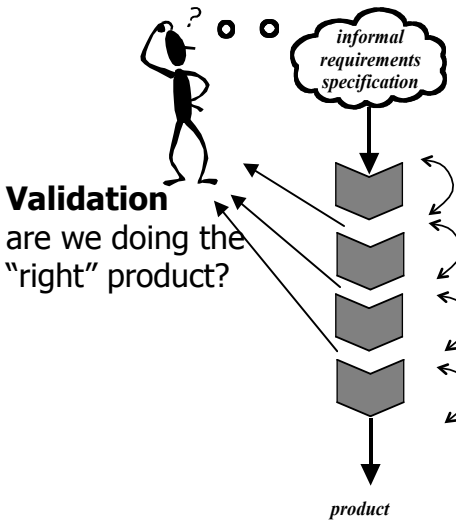
Need for transparency

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



56

Verification and validation



57

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

58

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

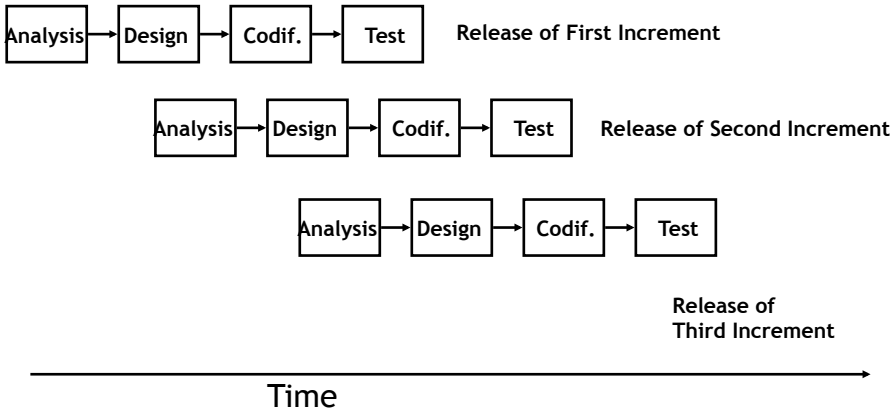
59

Incremental delivery

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

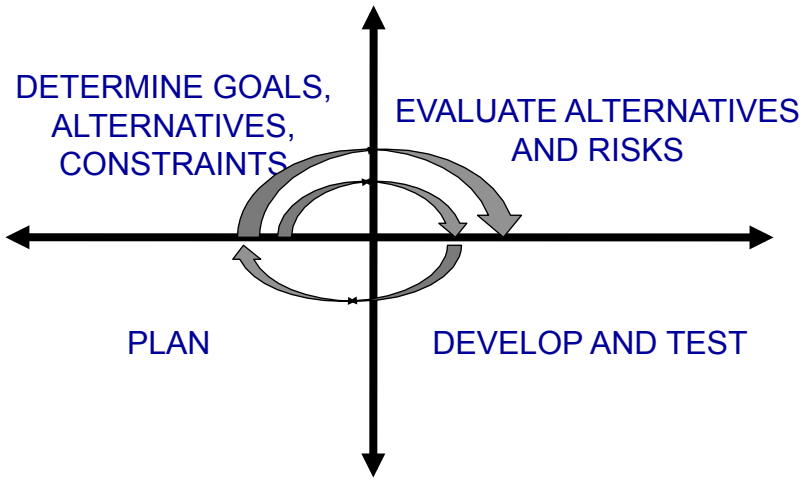
60

Incremental delivery



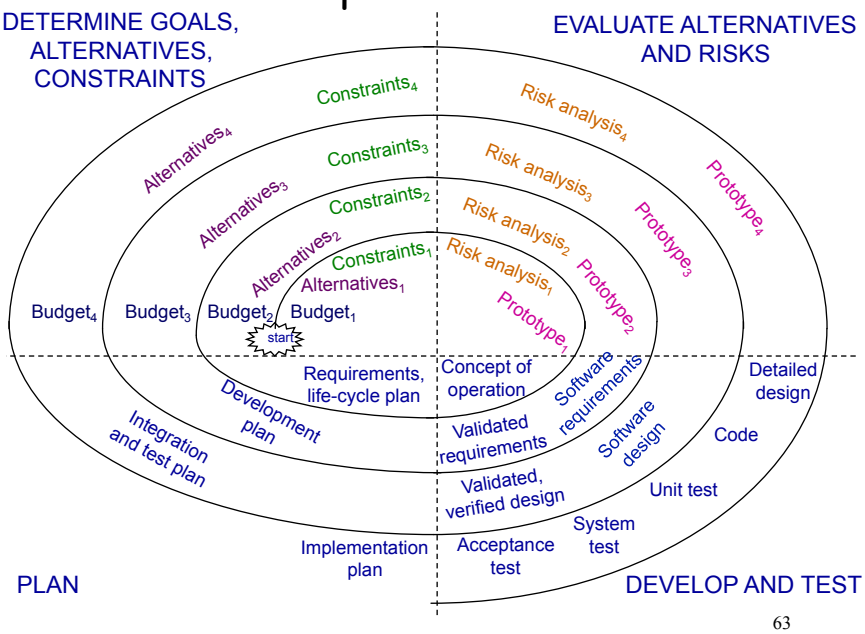
61

The spiral model B. Boehm, 1988



62

The spiral model



Spiral sectors

- DETERMINE GOALS, ALTERNATIVES, CONSTRAINTS
  (Goal: Objective setting)
  - Specific objectives for the phase are identified
- EVALUATE ALTERNATIVES AND RISKS
  (Goal: Risk assessment and reduction)
  - Risks are assessed and activities put in place to reduce the key risks
- DEVELOP AND TEST
  - A development model for the system is chosen which can be any of the generic models
- PLAN
  - The project is reviewed and the next phase of the spiral is planned

On the spiral

- Cost (radius) increases as we go around
- It is a metamodel that may describe any model, including waterfall (just go around a pre-determined number of times)

65

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

66

Lifecycle case studies

67

Synchronize&Stabilize (The Microsoft Software Process)

- Planning phase
  - vision of the product, specification, schedule
- Development
  - team composed of 2 groups
    - developers and testers (continuous testing)
  - process prescribes
    - daily synchronization
    - product stabilization

68

The sync and stabilize approach

- Develop
- Sync and stabilize
- Develop
- Sync and stabilize
- …
- Develop
- Sync and stabilize

69

Extreme programming and agile methods

- Analogy: like extreme climbing
- Response to stringent requirements of reduced time to market
- Very fashionable today
- Brings flexibility to the extreme
- Premise
  - sw impossible to pre-specify
  - like driving: continuous re-adjustment

70

XP

XP is a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements (K. Beck, 2000)

"Agile" is the generic name (see also http://agilemanifesto.org/)

71

4 basic principles

- communication
  - favor the right amount of communication in the project
- simplicity
  - try to find simplest solution that works
- feedback
  - continuous feedback from customers, and within the project via testing
- courage
  - throw away and redo when necessary

72

4 main activities

Back to basics:

- coding
- testing
- listening
- design

73

XP key practices (1)

- The planning game
  - determine the scope of next release; define plan; as reality overtakes the plan, replan
    - users define the goals via **stories**
    - developers estimate the effort for each **story**
    - priorities are defined jointly
  - corresponds to *feasibility study*

74

User Stories

- Individual functionality in a XP project
- A well-designed user story is
  - I. independent
  - II. negotiable
  - III. valuable for users
  - IV. assessable
  - V. small
  - VI. testable

75

Responsibilities

- Responsibilities of developers
  - **help** users to **write good stories** (see previous list)
  - write the stories that include **technical details** so that the description of their **impact** on user activities is clear
  - **estimate** the **effort** to implement stories
- Responsibility of users
  - **write good stories**…

76

I. Independent

- Hypothesis (ideal case)
  - no dependencies between user stories
- Real case
  - Ignore dependencies, but:
    - whenever possible combine dependent stories into a single story
    - re-organize stories if there are many dependencies

77

I. Independent (examples)

- Dependent stories
  - S1: users can pay with a Visa credit card
  - S2: users can pay with a Mastercard credit card
  - S3: users can pay with an AmericanExpress credit card
- Unification
  - S1: it is possible to pay with several credit cards
- Re-organization
  - S1: It is possible to pay with a credit card
  - S2: Additional credit cards are supported

78

II. Negotiable

- Stories must be considered as reminders for running discussions
- They do not include all details
  - Functionality is not defined in detail
  - Notes report relevant details
  - Stories can be re-negotiated and modified at any time

A company can pay for a job posting with a credit card.

Note: Accept Visa, MasterCard, and American Expreee. Consider Discover.

79

II. Negotiable: limit details

- While negotiation continues, many details can be added to stories
- …then working on the story is complex

story with too many details

A company can pay for a job posting with a credit card.
Note: Accept Visa, MasterCard, and American Expreee. Consider Discover. On purchases over $100, ask for card ID number from back of card. The system can tell what type of card it is from the first two digits of the card number. The system can store a card number for future use. Collect the expiration month and date of the card.

80

II. Negotiable: evolution of stories

- Keep the main definition of the functionality
- Decisions that are made are reported in the notes to support acceptance testing
- Only important information and issues still to be addressed continue to be part of the story

A company can pay for a job posting with a credit card.

Note: Will we accept Discover cards?
Note for UI: Don't have a field for card type (it can be derived from first two digits on the card).

81

III. Valuable for Users

- Stories must be useful and understandable from users
  - drop stories only useful to developers
  - remove technical details from stories
- Examples:
  - useful stories
    - S1: users can only pay with credit cards
  - useless stories
    - S2: connection to a DB is managed with a unique connection pool

82

IV. Assessable

- Developers should be able to evaluate the effort required to the implementation of the user story
- Difficulties about the estimation of the effort:
  1. Lack of knowledge about the domain
  2. Lack of knowledge about related techniques and technologies
  3. Inadequate size of the story (too big)
- Solutions
  1. Discuss with users
  2. Plan a task for investigating existing technologies (if this task is particularly long, it can be a story itself)
  3. Split the story in multiple stories

83

V. Size

- It should be possible to implement a story in less than 1 week (better if 2-3 days)
- Big stories must be split
  - by distributing number of functionalities
  - by distributing the data that can be processed
  - e.g.,
    - users can provide their CV
    - Splitting wrt functionality:
      - Users create their CV, which includes personal information, past working experience, …
      - Users can edit their CV
      - Users can delete their CV
      - …
    - Splitting wrt data
      - Users can add and modify the CV's section about part working experiences
      - Users can add and modify the CV's section about personal information
      - …
- Short stories must be grouped

84

VI. Testable

- Testability is a fundamental property
  - If a story is not testable, how do developers know if it has been correctly implemented?
- Always add acceptance testing to user stories
  - What will users do to verify the story?
- Include measurable aspects as much as possible
  - In particular for non-functional aspects
- As far as possible, include test cases that can be automatically executed

85

VI. Testable - Examples

- S1: Users can pay with several credit cards
- Acceptance testing
  - Pay products with Visa, Mastercard and AmericanExpress credit cards

86

XP key practices (2)

- Short releases
  - similar to "early subset, early deliverey"
- Simple design
  - at any given time
    - runs all tests
    - has no duplicate logic
    - has fewest possible classes and methods

  (…don't think of future changes)

87

XP key practices (3)

- Refactoring
  - when implementing a feature, always consider possible changes and improvements (restructuring)
- Testing (test first)
  - do continuous (automatic) testing
    - system test based on stories
    - use automatic tyools (e.g., JUnit)

88

Refactoring

- At any time during development:
  - if necessary, rework the code
  - small incremental changes (5 min)
  - **always execute test cases after changes!!**
- **Before** the addition of a new functionality
  - Is it possible to modify the system to favor the addition of the new functionality?
- **After** the addition of a new functionality
  - Is it possible to simplify the system, without modifying test cases?
- NOTE: if building and testing are expensive, refactoring is limited
- Check-in only when
  - all test cases have been passed
  - duplicated code has been removed
  - the code is readable (expressive)
  - the code is as simple as possible

89

XP key practices (4)

- Pair programming
  - code writing done by pairs working at the same terminal
    - keeps attention high
- Collective ownership
  - anybody who sees an oppurtunity for improvements can change code
    - simple code organization and common coding conventions favor this

90

Collective Ownership

- Each **developer can access to the whole code**
  - If a programmer needs a change… he/she accesses to the code and modifies it
  - If the **code** is **reworked** by multiple developers, its quality **improves**
  - If a member of a pair has extended knowledge of a part of the system, the other member can take advantage of this knowledge
  - A version control system is necessary to enforce this practice

91

Pair Programming

- 2 developers working together at each host
- they alternate development and programming
- pairing
  - transfer knowledge between team members
  - new people is trained
- high productivity
  - pairs no worse of two programmers
  - pairs produce less faults
  - pairs produce compact code
  - pairs more happy than singles

92

XP key practices (5)

- Continuous integration
  - frequent check-in, daily builds
  - not final big-bang
- 40-Hour week
  - programmer is not a zombie
  - work is pleasure and creativity, not stress

93

XP key practices (6)

- On-site customer
  - customer participates; sits with the team
- Coding standards
  - never duplicate code
  - use common conventions

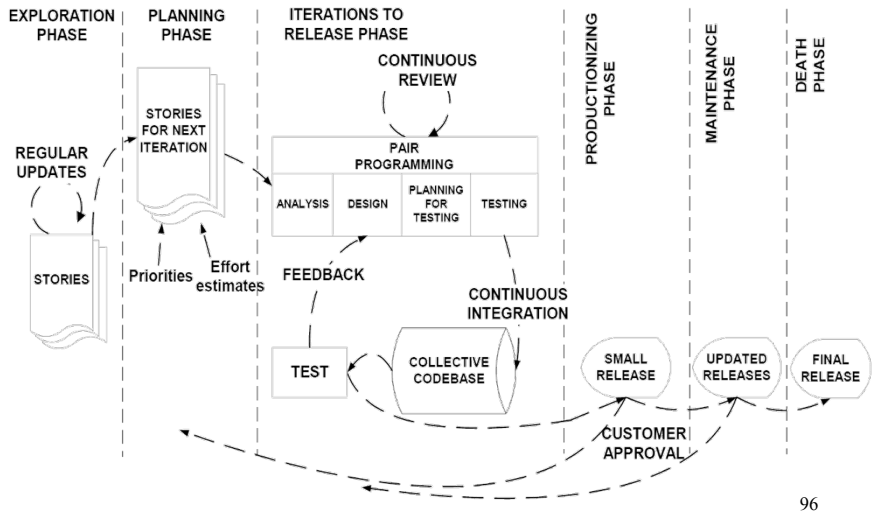
94

Coding Standard

- **Coding conventions shared** between team members
  - they **spontaneously emerge** overtime (as a consequence of collective ownership)
  - continuous **evolution**
  - increase **expressiveness** and **readability** of the code

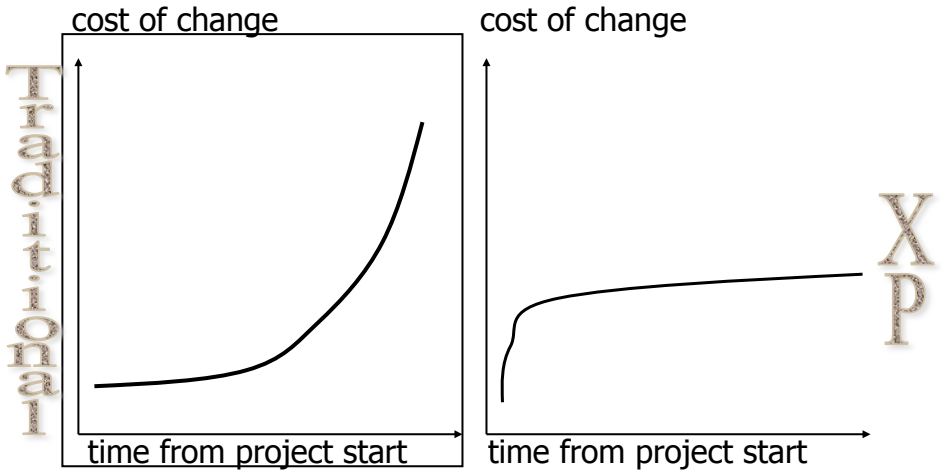
95

XP in 1 chart



96

XP claims



97

Preliminary assessment

- Some practices seem to be very effective (e.g., pair programming)
- Effectiveness depends on the type of project
  - small size
  - volatile requirements
  - end-user oriented

98

Unified Process (Jacobson, 1999)

- Known also as RUP (Rational Unified Process)
- Uses the UML notation throughout the process
- Supports an iterative and incremental process
- Decomposes a large process into controlled iterations (*mini projects*)

99

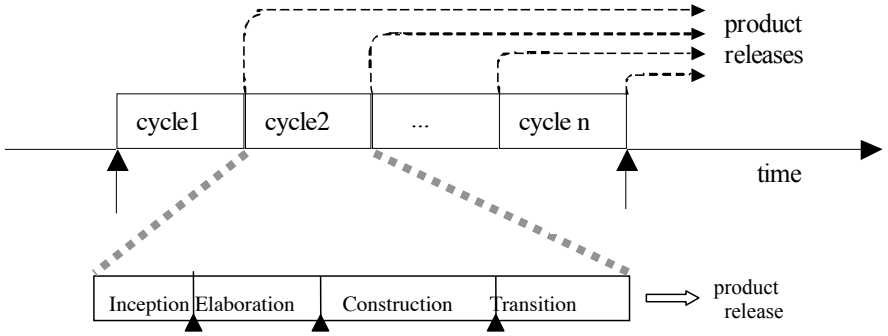
(Rational) Unified Process

- It is a configurable process:
  - It defines a "framework" or "generic process" that can be customized according to specific software systems, organizations, application domain, competence level, and project size.
- And …
  - It is also a product commercialized by Rational.
  - Rational offers also a set of tools to automatically support the activities that are part of the RUP.

100

Cycles and phases of a cycle

each cycle provides an "increment" of the product
increment implement use cases that are of use to users



milestone ▲ subject to quality control via reviews and inspections

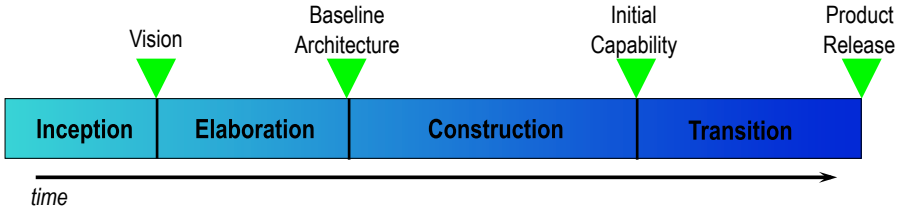
101

Main features

- Iterative and incremental
  - To take into account complexity
  - To accommodate the evolution of requirements
- Emphasis is on modeling rather than on informal languages
  - Increased readability and modifiability
  - Amenable to automatic support
- Centered on software architecture
  - The software architecture emerges from the very beginning of the project
  - Parallel development
  - Reuse and maintenance

102

Phases and milestones



- Inception: Identification of the "Business Case" and project scoping
- Elaboration: Project planning, feature definition (problem description) and architectural design
- Construction: System development
- Transition: System deployment

103

Inception

- Roughly corresponds to a feasibility study
  - goal is to document a vision of the product and a business analysis that justifies why the development should be undertaken
    - Who are the expected users of the system, and why should they be willing to use it?
    - What is the preliminary, high-level architecture of the system?
    - What is the development plan and what are the expected development costs?

104

Elaboration

- The use cases that are assumed to become part of the current release are specified in detail, and the software architecture is developed and specified
- Expected result of this phase is an architecture model
- UP insists that this should constitute a baseline
  - the set of documents describing the architecture must represent an agreed-upon basis for further development within the development team, thus any change is regulated by a formal procedure to avoid inconsistencies
- At the end of the phase, project managers can refine the development plan and the cost estimations made in the inception phase

105

Construction

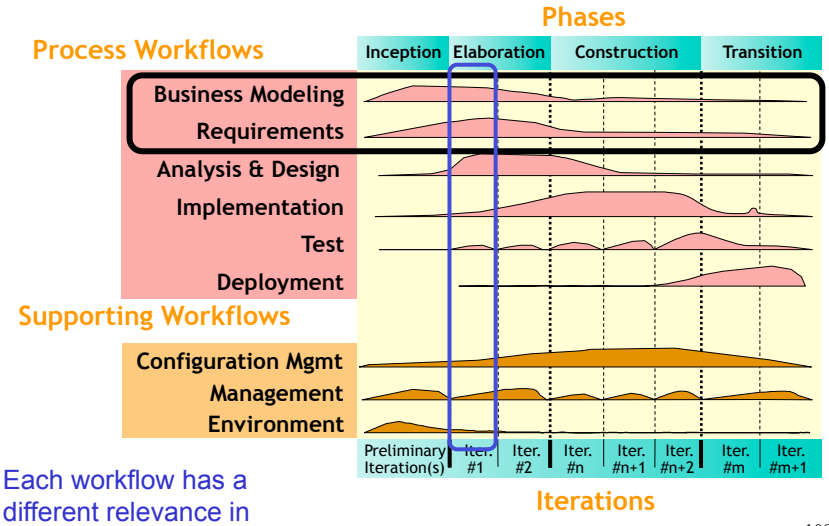
- The product is built by enriching the architecture baseline and developing and testing the code
- The final milestone of this phase will be a quality check that responds to the following question:
  - Does the product satisfy the user's needs adequately to warrant its delivery?

106

Transition

- Corresponds to beta testing
  - trusted set of early adopters tries the product and provides reports on possible defects and deficiencies
    - some of these will immediately be taken care of by developers; others might be delayed to a subsequent release
    - tracking of defects and deficiencies and policy adopted to handle them is crucial in this phase
  - the milestone of the phase is the delivery of the current cycle's product

Iterations and workflows



Each workflow has a different relevance in each iteration.

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

109

Concluding remark

- Most of the approaches share some similarities.
- Many of them can be combined to create "your own lifecycle". This has to be accomplished by keeping mind:
  - The market segment.
  - The application domain.
  - The technology being used.
  - The human resources involved in the project.
  - The business goals and mission of the development company.

110

Software

Product

Process

Qualities

Principles

111

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

112

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

113

Quality dimensions

Process vs. Product

Internal vs. External

Internal qualities affect external qualities
Process quality affects product quality

114

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)

115

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)

116

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

121

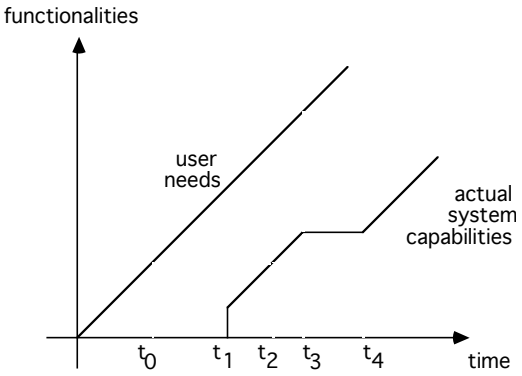
Productivity: folk data

- 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"

122

Process qualities: timeliness

- Ability to respond to change requests in a timely fashion



123

Key SE principles

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

124