

1

Introduction to Programming

Bertrand Meyer

Last revised 2 February 2004

Chair of Software Engineering Introduction to Programming – Lecture 26

2

Lecture 26: From Programming to Software Engineering

Chair of Software Engineering Introduction to Programming – Lecture 26

3

Software engineering (1)

The processes, methods, techniques, tools and languages for developing **quality** operational software.

Chair of Software Engineering Introduction to Programming – Lecture 26

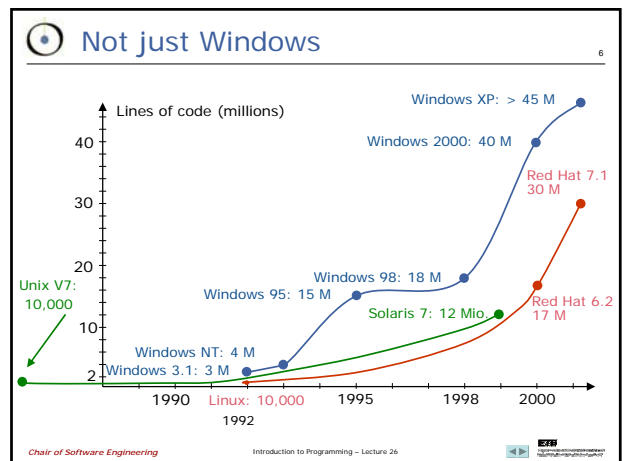
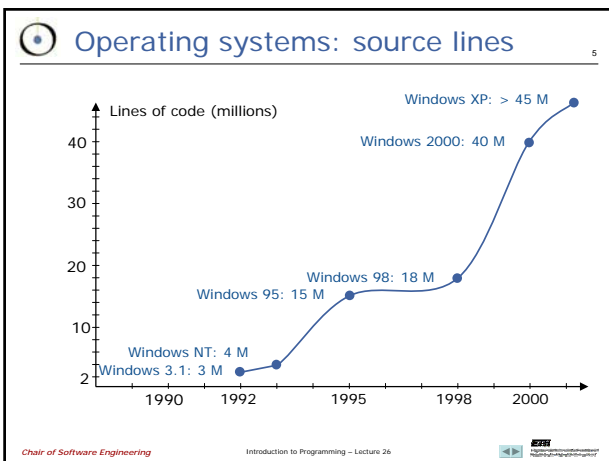
4

Software engineering (2)

The processes, methods, techniques, tools and languages for developing **quality** operational software that may need to

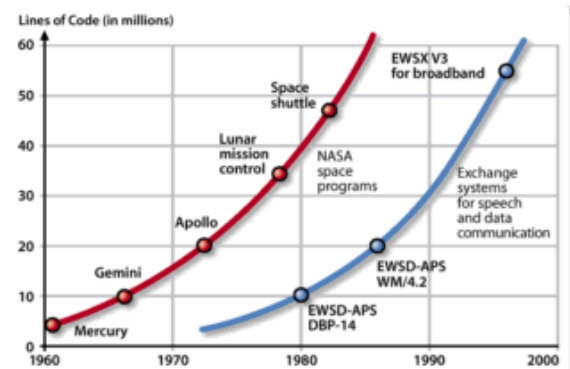
- Be of large size
- Be developed and used over a long period
- Involve many developers
- Undergo many changes and revisions

Chair of Software Engineering Introduction to Programming – Lecture 26



Not just operating systems

7



Chair of Software Engineering

Introduction to Programming – Lecture 26



The basic issue

8

Developing software systems that are

- On time and within budget
- Of high immediate quality
- Possibly large and complex
- Extendible

Chair of Software Engineering

Introduction to Programming – Lecture 26



US Software industry, 1998

9

Standish Group: "Chaos" Report

250,000 Software projects, \$275 billions

- Project results:
 - 28% abandoned (1994: 31%)
 - 27% successful (1994: 16%)

The rest: "challenged"

- Smaller projects have higher chances of success

Chair of Software Engineering

Introduction to Programming – Lecture 26



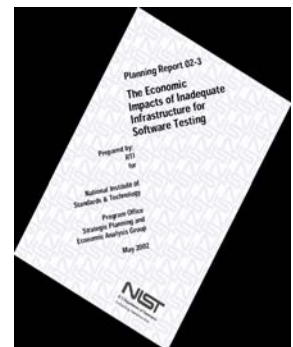
NIST report on testing (May 2002)

10

- Financial consequences, on developers and users, of "insufficient testing infrastructure"

\$ 59.5 B.

(Finance \$ 3.3 B, Car and aerospace \$ 1.8 B. etc.)



Chair of Software Engineering

Introduction to Programming – Lecture 26



Software quality factors

11

- **External:** of interest to customers
 - Examples: Reliability, Extendibility
- **Internal:** of interest to customers
 - Examples: Modularity, Style

Chair of Software Engineering

Introduction to Programming – Lecture 26



Some internal factors

12

- Modularity
- Observation of style rules
- Consistency
- Structure
- ...

Chair of Software Engineering

Introduction to Programming – Lecture 26



External factors: reliability

13

Reliability = Correctness + Robustness + Integrity

Chair of Software Engineering Introduction to Programming – Lecture 26

Reliability

14

- **Correctness**
The system's ability to perform its functions according to the specification, in cases covered by the specification
- **Robustness**
The system's ability to handle erroneous cases safely
- **Integrity**
The system's ability to protect its users, its data and itself against hostile uses

Chair of Software Engineering Introduction to Programming – Lecture 26

External factors

15

Product quality (immediate):

- Reliability
- Efficiency
- Ease of use
- Ease of learning

Process quality:

- Timeliness
- Cost-effectiveness

Product quality (long term):

- Extensibility
- Reusability
- Portability

Chair of Software Engineering Introduction to Programming – Lecture 26

Software tasks

16

- Requirements analysis
- Specification
- Design
- Implementation
- Validation & Verification (V&V)
- Management
- Planning and estimating
- Measurement

Chair of Software Engineering Introduction to Programming – Lecture 26

Validation & Verification

17

- **Verification:** checking that you have built the system right
(followed all rules)
- **Validation:** checking that you have built the right system
(satisfied user needs)

Chair of Software Engineering Introduction to Programming – Lecture 26

Requirements analysis

18

- Understanding user needs
- Understanding constraints on the system

Chair of Software Engineering Introduction to Programming – Lecture 26

Software lifecycle models

19

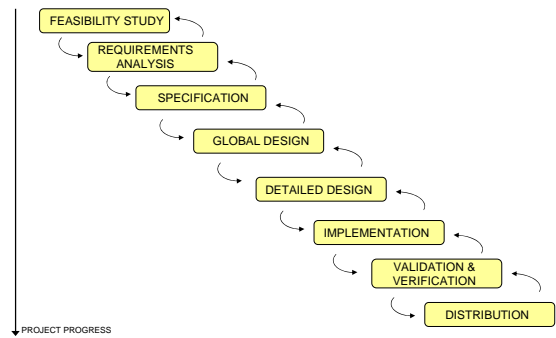
Describe an overall distribution of the software construction into tasks, and the ordering of these tasks

They are models in two ways:

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

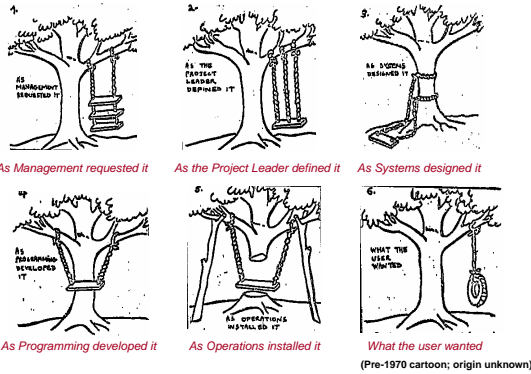
The waterfall model (Royce, 1970)

20



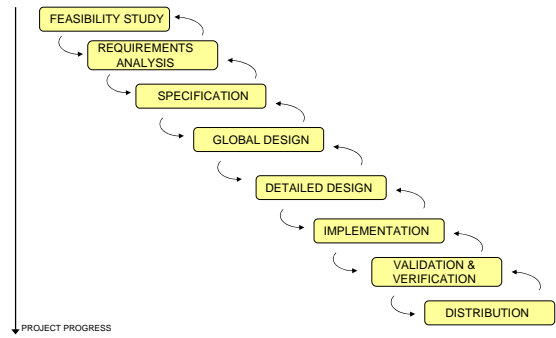
Lifecycle: what not to achieve

21



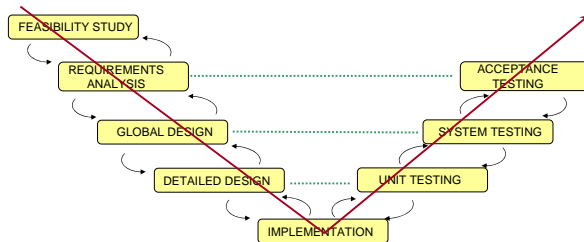
The waterfall model

22



A more realistic version

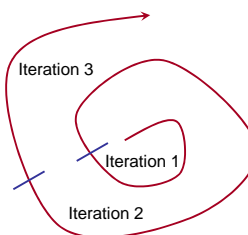
23



The spiral model

24

- Apply a waterfall-like approach to successive prototypes





The problem with prototyping

25

- 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



Seamless, incremental development

26

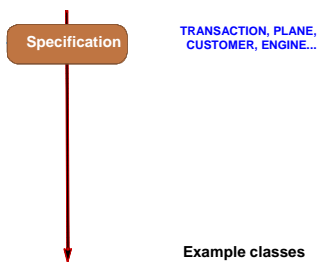
The Eiffel view:

- Single set of notation, tools, concepts, principles throughout
- Eiffel is as much for analysis & design as for implementation & maintenance
- Continuous, incremental development
- Keep model, implementation and documentation consistent
- Reversibility: can go back and forth



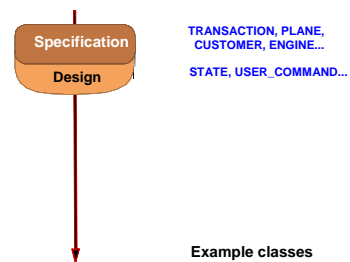
Seamless development (1)

27



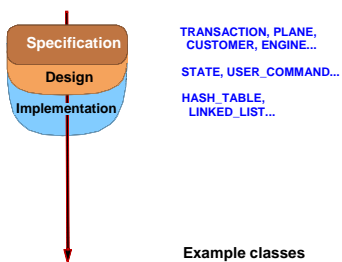
Seamless development (2)

28



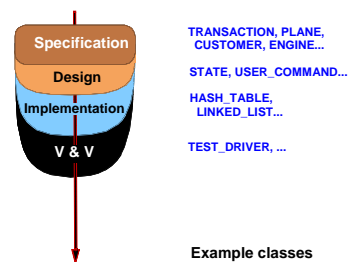
Seamless development (3)

29



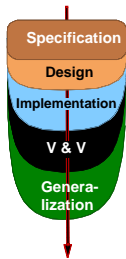
Seamless development (4)

30



Seamless development (5)

31



TRANSACTION, PLANE,
CUSTOMER, ENGINE...
STATE, USER_COMMAND...
HASH_TABLE,
LINKED_LIST...
TEST_DRIVER, ...
AIRCRAFT, ...
Example classes

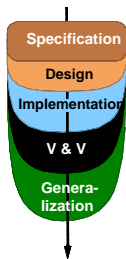
Generalization

32

- Prepare for reuse
- For example:
 - Remove built-in limits
 - Remove dependencies on specifics of project
 - Improve documentation, contracts...
- Few companies have the guts to provide the budget for this

Seamless development

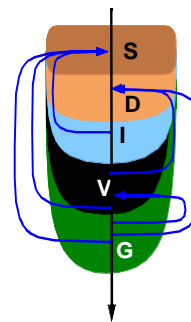
33



TRANSACTION, PLANE,
CUSTOMER, ENGINE...
STATE, USER_COMMAND...
HASH_TABLE,
LINKED_LIST...
TEST_DRIVER, ...
AIRCRAFT, ...
Example classes

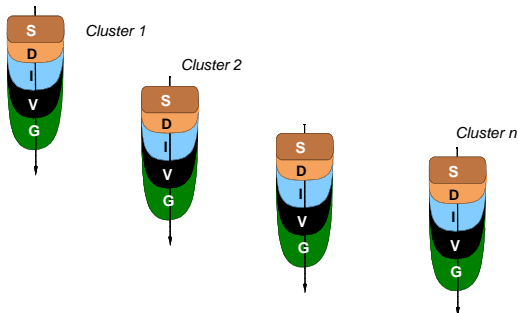
Reversibility

34



The cluster model

35



Agile methods and extreme programming

36

- De-emphasize process and reuse
- Emphasize the role of tests to guide the development

Validation and Verification

37

- Not just testing:

Static Analysis tools explore code for possible deficiencies, e.g. uninitialized variables

- Should be performed throughout the process, not just at the end

Formal methods

38

- Use mathematics as the basis for software development
- A software system is viewed as a mathematical theory, progressively refined until directly implementable
- Every variant of the theory and every refinement step is **proved**
- Proof supported by computerized tools
- Example: *Atelier B*, security system of newest Paris Metro line

Metrics

39

Things to measure:

- Product attributes: lines of code, number of classes, complexity of control structure (“cyclomatic number”), complexity and depth of inheritance structure, presence of contracts...
- Project attributes: number of people, person-months, costs, time to completion, time of various activities (analysis, design, implementation, V&V etc.)

Taking good measurements helps take good measures

Cost models

40

- Attempt to evaluate cost of software development ahead of project, based on estimate of parameters
- Example: COCOMO (Constructive Cost Model), Barry Boehm

L: 1000 * Delivered Source Instructions (KDSI)

Program type	Effort (pm)	Time
Application	$2.4 * L * 1.05$	$2.5 * pm * 0.38$
Utility	$3.0 * L * 1.12$	$2.5 * pm * 0.35$
System	$3.6 * L * 1.20$	$2.5 * pm * 0.32$

Software reliability models

41

- Estimate number of bugs from
 - Characteristics of program
 - Number of bugs found so far
- Variant: “Fault injection”

Project management

42

- Team specialties: customer relations, analyst, designer, implementer, tester, manager, documenter...
- What role for the manager: managerial only, or technical too?
- “Chief Programmer teams”



- In the end it's code
- Don't underestimate the role of tools, language and, more generally, technology
- Bad management kills projects
Good technology makes projects succeed



End of lecture 26