Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Lecture 15: From Programming to Software Engineering
Software engineering (1)

The processes, methods, techniques, tools and languages for developing quality operational software.
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
Moore’s “Law”

Approximate doubling of computer power, for comparable price, every eighteen months

Speed of Intel processors

- 8008: < 1 MHz
- 80386: 33 MHz
- 80486: 50 MHz
- Pentium: 133 MHz
- Pentium IV: 1.3 GHz
- 3.8 GHz

(1 Hertz = 1 clock cycle per second)

- From 1 MHz to 2 GHz: 8 months
- To 1 GHz: 26 years
In other application areas

(source: Siemens)
The basic issue

Developing software systems that are

- On time and within budget
- Of high immediate quality
- Possibly large and complex
- Extendible
What does software quality mean?
Non-quality

A problem has been detected and Windows has been shut down to prevent damage to your computer.

The problem seems to be caused by the following file: ati2dvag

If this is the first time you’ve seen this stop error screen, restart your computer. If this screen appears again, follow these steps:

The device driver got stuck in an infinite loop. This usually indicates a problem with the device itself or with the device driver programming the hardware incorrectly.

Please check with your hardware device vendor for any driver updates.

Technical information:

*** STOP: 0x000000EA (0x8A5C31B8, 0x8A418330, 0xF78BECB4, 0x00000001)

ati2dvag
Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further assistance.
LOS ANGELES. Failure of the Southwest's main air traffic radar system was traced to new software unable to recognize data typed manually by Mexico controllers.

The software installed Wednesday evening at the FAA's Los Angeles Center in the Mojave Desert, which controls aircraft over a 100,000-square-mile area, is the same upgrade completed successfully at 19 other FAA radar centers. But designers didn't allow for information typed in by Mexico controllers, who don't have a computerized system, the FAA spokesman said. "The computer didn't recognize the information from Mexico and it aborted". "A digit out of place could do it."

When controllers at the LA Center switched to the new system Thursday morning, it quickly failed when data from a Mexico controller was received. The radar system instantly switched to backup. The computer with the new software was restarted later, but failed again. The old system was reinstalled and the system returned to operation more than two hours later. Air travel schedules were left in disarray as the FAA ordered a nationwide ground stop for all flights bound for the Southwest, causing cancellations, rerouting, long delays and airport gridlock.

Technicians must now rewrite the software to recognize Mexico controller information. It wasn't clear when a revised program would be installed.
1998 Mars Orbiter Vehicle*

The orbiter was lost due to a miscalculation in trajectory. The miscalculation was caused by an unintended and undetected mismatch between metric and English units of measurement. The use of metric units as well as the data formats to employ were specified in a navigation software interface specification (SIS) published by JPL in 1996. Despite this, the flight operations team at Lockheed Martin provided impulse data in English units of pound-force seconds rather than newton seconds. These values were incorrect by a factor of 4.45 (1 lbf = 4.45 N). The mix-up caused erroneous course corrections that resulting in the orbiter descending too low in Mars atmosphere. The vehicle either burned up or bounced off into space.

Ariane-5 maiden launch, 1996

37 seconds into flight, exception in Ada program not processed; order given to abort mission. Loss estimated to $10 billion. Exception was caused by an incorrect conversion: a 64-bit real value was incorrectly translated into a 16-bit integer.

Systematic analysis had “proved” that the exception could not occur - the 64-bit value (“horizontal bias” of the flight) was proved to be always representable as a 16-bit integer!

It was a REUSE error:
- The analysis was correct - for Ariane 4!
- The assumption was documented - in a design document!

Security example: the buffer overflow

System expects some input from an external user:

First name: 
Last name: 
Address: 

from $i := 1$ until $i > \text{input\_size}$

loop

$\text{buffer}[i] := \text{input}[i]$

$i := i + 1$

end
C brings in a special twist…

For a string, there’s no way to know `input_size` in advance

You have to read until you find the string terminator, \0 (the null character)

These two strings walk into a bar. The bartender says: "What will it be?"

The first string says: "I think I’ll have a beer zdiup tako^jDjftk /. \134.206.21.02 C#VB.NET 8086%N ~~|~~#@$ Dz @-)"

"Please excuse my friend," the second string says, "He isn't null-terminated."
Getting the input

\[
\text{from } i := 1 \text{ until } i > \text{input\_size} \\
\text{loop} \\
\quad \text{buffer}[i] := \text{input}[i] \\
\quad i := i + 1 \\
\text{end}
\]
Overflowing a buffer!

The buffer array (overflowing)

"The stack" (activation records)

Return address, arguments, locals

Memory

Code of routine $n-1$

My nasty code

My return

Routine $n$

Routine 2

Routine 1

Main

Data

Programs

Max

0
Getting the input

\[i := 1 \text{ until } i > \text{input\_size or } i > \text{buffer\_size}\]

\textbf{loop}

\[\text{buffer}[i] := \text{input}[i]\]

\[i := i + 1\]

\textbf{end}
US software industry, 2000

Source: Standish report

Project leaders and CIOs representing several thousand software projects

Project outcome:

- 28% success (1998: 26%)
- 23% failure (1998: 28%)
- Rest: “challenged” (1998: 46%)
  (completed over budget, over time, under features)

Smaller projects have a higher chance of succeeding
NIST report on “testing” (May 2002)

Monetary effect on
Developers and
User due to
“insufficient testing infrastructure“:

$59.5 billion

(Financial sector: $3.3 billion,
auto/aerospace $1.8 billion etc.)
Software quality: external vs internal

External factors: visible to customers

(not just end users but e.g. purchasers)

• *Examples*: ease of use, extendibility, timeliness

Internal factors: perceptible only to developers

• *Examples*: good programming style, information hiding

Only external factors count in the end, but the internal factors make it possible to obtain them.
Software quality: product vs process

**Product**: properties of the resulting software

For example: correctness, efficiency

**Process**: properties of the procedures used to produce and “maintain” the software
Some external factors

Product quality (immediate):
- Reliability
- Efficiency
- Ease of use
- Ease of learning

Process quality:
- Production speed (timeliness)
- Cost-effectiveness
- Predictability
- Reproducibility
- Self-improvement

Product quality (long term):
- Extendibility
- Reusability
- Portability
Reliability

Correctness:
The systems’ ability to perform according to specification, in cases covered by the specification.

Robustness:
The systems’ ability to perform reasonably in cases not covered by the specification.

Security (integrity):
The systems’ ability to protect itself against hostile use.
Some internal factors

Modularity
Observation of style rules
Consistency
Structure
...


External factors

**Product quality (immediate):**
- Reliability
- Efficiency
- Ease of use
- Ease of learning

**Process quality:**
- Timeliness
- Cost-effectiveness

**Product quality (long term):**
- Extendibility
- Reusability
- Portability
Software tasks

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

Understanding user needs
Understanding constraints on the system
  ➢ Internal constraints: class invariants
  ➢ External constraints
The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

*For sources cited, see bibliography*
Goals of performing requirements

- Understand the problem or problems that the eventual software system, if any, should solve
- Prompt relevant questions about the problem & system
- Provide basis for answering questions about specific properties of the problem & system
- Decide what the system should do
- Decide what the system should not do
- Ascertain that the system will satisfy the needs of its stakeholders
- Provide basis for development of the system
- Provide basis for V & V* of the system

*Validation & Verification, especially testing

Source: OOSC
Products of requirements

- Requirements document
- Development plan
- Test plan
15 quality goals for requirements

- Justified
- Correct
- Complete
- Consistent
- Unambiguous
- Feasible
- Abstract
- Traceable

- Delimited
- Interfaced
- Readable
- Modifiable
- Verifiable
- Prioritized*
- Endorsed

Marked attributes are part of IEEE 830, see below
* "Ranked for importance and/or stability"
Difficulties of requirements

- Natural language and its imprecision
- Formal techniques and their abstraction
- Users and their vagueness
- Customers and their demands
- The rest of the world and its complexity
Bad requirements

The Background Task Manager shall provide status messages at regular intervals not less than 60 seconds.

Better:

The Background Task Manager (BTM) shall display status messages in a designated area of the user interface

1. The messages shall be updated every 60 plus or minus 10 seconds after background task processing begins.

2. The messages shall remain visible continuously.

3. Whenever communication with the background task process is possible, the BTM shall display the percent completed of the background task.
Bad requirements

The XML Editor shall switch between displaying and hiding non-printing characters instantaneously.

Source: Wiegers

Better:

The user shall be able to toggle between displaying and hiding all XML tags in the document being edited with the activation of a specific triggering mechanism. The display shall change in 0.1 second or less.
Bad requirements

The XML parser shall produce a markup error report that allows quick resolution of errors when used by XML novices.

Better:

1. After the XML Parser has completely parsed a file, it shall produce an error report that contains the line number and text of any XML errors found in the parsed file and a description of each error found.

2. If no parsing errors are found, the parser shall not produce an error report.

Source: Wiegers
Verifiable requirements

Non-verifiable:
- The system shall work satisfactorily
- The interface shall be user-friendly
- The system shall respond in real time

Verifiable:
- The output shall in all cases be produced within 30 seconds of the corresponding input event. It shall be produced within 10 seconds for at least 80% of input events.
- Professional train drivers will reach level 1 of proficiency (defined in requirements) in two days of training.
Practical advice

Favor precise, falsifiable language over pleasant generalities
IEEE 830-1998

“IEEE Recommended Practice for Software Requirements Specifications”

Approved 25 June 1998 (revision of earlier standard)

Descriptions of the content and the qualities of a good software requirements specification (SRS).

Goal: “The SRS should be correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable, traceable.”
Recommended document structure:

1. Introduction
   1.1 Purpose
   1.2 Scope
   1.3 Definitions, acronyms, and abbreviations
   1.4 References
   1.5 Overview

2. Overall description
   2.1 Product perspective
   2.2 Product functions
   2.3 User characteristics
   2.4 Constraints
   2.5 Assumptions and dependencies

3. Specific requirements

Appendixes

Index
Practical advice

Use the recommended IEEE structure
Practical advice

Write a glossary
Some recipes for good requirements

Managerial aspects:
- Involve all stakeholders
- Establish procedures for controlled change
- Establish mechanisms for traceability
- Treat requirements document as one of the major assets of the project; focus on clarity, precision, completeness

Technical aspects: how to be precise?
- Formal methods?
- Design by Contract
Validation & Verification

**Verification**: checks of internal consistency
   Example: type checks

   “Checking that you have built the system right”

   (followed all rules)

**Validation**: checking against a higher-level description
   Example: validating a program against its specification

   “Checking that you have built the right system”

   (satisfied user needs)
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

- Feasibility study
- Requirements
- Specification
- Global design
- Detailed design
- Implementation
- V & V
- Distribution
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
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.
Quality control?

Analysts → Designers → Implementers → Testers → Customers
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

How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it

How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed
The spiral model (Boehm)

Apply a waterfall-like approach to successive prototypes
The Spiral model

Figure from: Ghezzi, Jazayeri, Mandrioli, *Software Engineering, 2nd edition*, Prentice Hall
“Prototyping” in software

The term is used in one of the following meanings:

- Experimentation:
  - Requirements capture
  - Try specific techniques: GUI, implementation (“buying information”)

- Pilot project
- Incremental development
- 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.
Seamless, incremental development

The Eiffel view:

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

Reversibility: can go back and forth
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
It seems that the sole purpose of the work of engineers, designers, and calculators in drawing offices and research institutes 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.

(Terre des Hommes, 1937)
Reversibility

- Analysis
- Design
- Implementation
- V&V
- Generalization
The cluster model

Cluster 1

Cluster 2
The cluster model

Cluster 1

Cluster 2

Cluster n
Seamless development: the Eiffel example

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
...

66
Quality goals: the Osmond curves

The advice: add functionality at constant quality
Agile/lean methods and extreme programming

De-emphasize formal process

Emphasize short-cycled, time-boxed iterative development

Emphasize the role of tests to guide the development (“TDD”, Test-Driven Development)

Emphasize the benefit of a second set of eyes: Pair programming

Emphasize the role of refactoring

Emphasize self-organizing teams

Emphasize customer involvement
Open-source processes

Collaborative, distributed developments

Concentric trust circles

Success with strong project leader (e.g. Linux)

“Given enough eyes, all bugs are shallow”
Validation and Verification

Not just testing:

- **Static Analysis** tools explore code for possible deficiencies, e.g. uninitialized variables
- **Proofs of correctness** are increasingly becoming realistic
- **Model checking** explores the state space of an abstracted version of the program

Quality assurance should be performed throughout the process, not just at the end
Software engineering tools

Development environments (compiler, browser, debugger, ...): “IDE”

Documentation tools

Requirements gathering tools

Analysis and design tools

Configuration & version management (CVS, Source Safe...) (also “make” etc.)

Formal development and proof tools

Integrated CASE (Computer-Aided Software Engineering) environments
Configuration management

Aim: make sure that versions used for the various components of a system are compatible

Two principal variants:
- Build management
- Version management
Build management

Make (late seventies): automatic reconstruction of a system from a “makefile” listing dependencies

Example

```
make program
```

with the makefile

```
program: main.o module1.o module2.o
    cc main.o module1.o module2.o
%.c: %.o
    cc $<
```

Main limitation: need to describe dependencies manually
Version management

Examples:
RCS,
Subversion

Main operations:
• Commit
• Update

Stores “diffs” between versions
Advice: avoid branching; reconcile early and often
Configuration management

These tools are available and easy to use

No project can afford to ignore them
Formal methods

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

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

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

<table>
<thead>
<tr>
<th>Program type</th>
<th>Effort (pm)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>2.4 * L * 1.05</td>
<td>2.5 * pm * 0.38</td>
</tr>
<tr>
<td>Utility</td>
<td>3.0 * L * 1.12</td>
<td>2.5 * pm * 0.35</td>
</tr>
<tr>
<td>System</td>
<td>3.6 * L * 1.20</td>
<td>2.5 * pm * 0.32</td>
</tr>
</tbody>
</table>

L: 1000 * Delivered Source Instructions (KDSI)
Software reliability models

Estimate number of bugs from
  - Characteristics of program
  - Number of bugs found so far

Variant: “Fault injection”
Project management

Team specialties: customer relations, analyst, designer, implementer, tester, manager, documenter...

What role for the manager: managerial only, or technical too?

“Chief Programmer teams”
Software engineering

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
Programming languages

Not just for talking to your computer!

A programming language is a tool for thinking
A bit of history

“Plankalkül”, Konrad Zuse, 1940s

Fortran (FORmula TRANSlator), John Backus, IBM, 1954

Algol, 1958/1960
Some FORTRAN code

100 IF (N) 150, 160, 170
150 A (I) = A (I) ** 2
   READ ("I6") N
   GOTO 100
C THE NEXT ONE IS THE TOUCH CASE
160 A (I) = A (I) + 1
   READ ("I6") N
   GOTO 100
170 STOP
END
Algol

International committee, Europeans and Americans; led to IFIP. Algol 58, Algol 60.

Influenced by (and reaction against) FORTRAN; also influenced by LISP (see next). Recursive procedures, dynamic arrays, block structure, dynamically allocated variables.

New language description mechanism: BNF (for Algol 60).
Algol W and Pascal

Successors to Algol 60, designed by Niklaus Wirth from ETH

Algol W introduced record structures

Pascal emphasized simplicity, data structures (records, pointers). Small language, widely adopted for teaching.

Helped trigger the PC revolution through Turbo Pascal from Borland (Philippe Kahn)
1968: Brian Kernighan and Dennis Richie, AT&T Bell Labs

Initially, closely connected with Unix

Emphasis on low-level machine access: pointers, address arithmetic, conversions

Frantically adopted by industry in the 80s and 90s
Lisp and functional languages

**LiSt Processing**, 1959, John McCarthy, MIT then Stanford

Fundamental mechanism is recursive function definition

Automatic garbage collection (in 1959!)

Numerous successors, e.g. Scheme (MIT)

**Functional languages**: Haskell, Scheme, ML
LISP “lists”

A list is of the form \((x_1 \; x_2 ...\) where each \(x_i\) is either
An atom (number, identifier etc.)
(Recursively) a list:

Examples:
- \((\)\)
- \((x_1 \; x_2)\)
- \((x_1 \; (x_2 \; x_3) \; x_4 \; (x_5 \; (x_6 \; () \; x_7)))\)

\(((x_1 \; x_2))\) is not the same as \((x_1 \; (x_2))\)
LISP function application and definition

The application of function $f$ to arguments $a$, $b$, $c$ is written

$$(f \ a \ b \ c)$$

Example function definition (Scheme):

```scheme
(define (factorial n)
  (if (eq? n 0)
      1
      (* n (factorial (- n 1)))))
```
Basic functions

Let \( \text{my\_list} = (A \ B \ C) \)

\((\text{CAR my\_list}) = A\)
\((\text{CDR my\_list}) = (B \ C)\)

\((\text{CONS A (B C)}) = (A \ B \ C)\)
Functions working on lists

(define double-all (list)
    (mapcar
        '(lambda (x) (* 2 x)) list))

(define (mapcar function f)
    (if (null? ls) '()
        (cons
            (function (car ls))
            (mapcar function (cdr ls))))))
Object-oriented programming


Grew into a full-fledged programming language

Smalltalk (Xerox PARC) added ideas from Lisp and innovative user interface ideas. Alan Kay, Adele Goldberg, Daniel Bobrow
“Hybrid” languages

Objective-C, around 1984: Smalltalk layer on top of C

C++, around 1985: “C with classes”

Made O-O acceptable to mainstream industry

Key moment: first OOPSLA, 1986
Java and C#

C++ with enough restrictions to permit type safety and garbage collection

Java initially marketed for applets in connection with the explosion of the Internet, 1995

C# adds “delegates” (agent-like mechanism)
Eiffel

First version goes back to mid-eighties, first demonstrated at OOPSLA 86

Emphasis on software engineering principles: information hiding, Design by Contract, static typing (through genericity), full application of O-O principles

Has found its main application area in mission-critical industrial applications