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.

Operating systems: source size

Lines of code (millions)
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

Software engineering

What does software quality mean?

Non-quality

Associated Press, 21 October 2000

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 resulted 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:

Getting the input

```
from i := 1 until i > input_size
loop
  buffer[i] := 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)

Overflowing a buffer!

```
from i := 1 until i > input_size or i > buffer_size
loop
  buffer[i] := input[i]
i := i + 1
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

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

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

- Correctness: The system's ability to perform according to specification, in cases covered by the specification.
- Robustness: The system's ability to perform reasonably in cases not covered by the specification.
- Security (integrity): The system's 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): Extensibility, 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

Validation & Verification

- Verification: checks of internal consistency
  - Example: type check
    "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

V-shaped

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

Problems with the waterfall

- Separate tools:
  - Programming environment
  - Analysis & design tools, e.g. UML
- Consequences:
  - Hard to keep model, implementation, documentation consistent
  - Constantly reconciling views
  - Inflexible, hard to maintain systems
  - Hard to accommodate bouts of late wisdom

The spiral model (Boehm)

Apply a waterfall-like approach to successive prototypes
"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
- 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

Example classes:

- PLANE, ACCOUNT, TRANSACTION…
- STATE, COMMAND…
- HASH_TABLE…
- TEST_DRIVER…
- TABLE…

Antoine de Saint-Exupéry

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 one, 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)

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.
Reversibility

Design
Implementation
V&V
Generalization

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

... Demo

Quality goals: the Osmond curves

Desirable
Common

Functionality

Envisaged
Release

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
**Dilbert and XP**

*Extreme Programming: I can’t even do all of these features in the first version.*

*And each feature needs to have some kind of user story.*

*Okay, here’s a story: you give me all of my features or I’ll ruin your life.*

Copyright © 2003 United Feature Syndicate, Inc.

---

**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, SourceSafe...)

(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

When poorly done, one of the biggest sources of software catastrophes

Good tools exist today, e.g. CVS, SourceSafe

Any project not using one of these tools is managed by a complete idiot

---

**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 * 1.06</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>3.8 * L * 1.20</td>
<td>2.5 * pm * 0.32</td>
</tr>
</tbody>
</table>

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 TOUGH 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).

Algor 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)

C

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

LISP 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 (x1 x2...) where each xi is either
An atom (number, identifier etc.)
(Recursively) a list:

Examples:

- ()
- (x1 x2)
- (x1 (x2 x3) x4 (x5 x6 () x7))

((x1 x2)) is not the same as (x1 (x2))

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 my_list = (A B C)

- (CAR my_list) = A
- (CDR my_list) = (B C)
- (CONS A (B C)) = (A B C)

Functions working on lists

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

```scheme
(define (mapcar function f)
  (if (null? ls) '()
      (cons (function (car ls))
            (mapcar function (cdr ls)))))
)
```

Object-oriented programming

Simula 67: Algol 60 extensions for simulation, University of
Oslo, 1967 (after Simula 1, 1964). Kristen Nygaard, Ole
Johan Dahl

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

End of lecture 23