Basic concepts of software maintenance

X. Gonze
Université Catholique de Louvain

Software engineering ... for physicists

Our expertise ... is NOT software engineering!

What is software engineering?
- Not the fact of switching from FORTRAN to C++... !!
- A human science: e.g. How to improve the developer’s productivity? (similarly to machine productivity)
- Potentially very important to us ...
- Compare with hardware evolution: “No single software engineering development will produce an order-of-magnitude improvement in programming productivity within ten years” F. Brooks, *No silver bullet*, 1986.
Outline
I. Software: components, development, maintenance
II. What really takes time?
III. The essence of software programming
IV. Linux experience
V. Coding rules


Software: components, development, maintenance

**Software**

Misconception: software is programs

Software = the programs, documentation, and operating procedures by which computers can be made useful to people

Components of a software system:
- Program: source code, object code
- Operating procedures: instructions/scripts to set up and use the program; instructions on how to treat failures; instructions/scripts on how to test the program
- Documentation: specification (formal document, but also articles!); description of I/O; description of internal variable/object; comments; interrelation graphs for modules; structuration of source code; test description, including input and reference files

Different points of view: users, developers, maintainers, installers, testers ...

**Software development: the waterfall model**

A model that is notoriously insufficient! A start for further understanding ...

1. Requirements
2. Design
3. Coding
4. Module testing
5. Integration
6. System testing
7. Installation
8. Maintenance
Maintenance / Software maintenance

Maintenance: the act of keeping an entity in an existing state of efficiency, validity, to preserve from failure or decline.

e.g. Car maintenance

Software maintenance: modification of a software product after delivery in order to
- improve performance,
- correct faults,
- adapt the product to a modified environment

In most of the cases, for atomic-scale software, we restart from existing software to add new functionalities. Maintenance is essential. Waterfall model is inadequate.

Software evolution

Changes:
- adaptive changes ... made in order to become suited to different conditions
- corrective changes ... made in order to remove defects
- perfective changes ... made in order to improve the software
- preventive changes ... made in order to reverse deterioration

Defects (bugs) result from: design errors; logic errors; coding errors

Ripple effect:
Consequences of an action in one place, occurring elsewhere
“A badly structured program is like a plate of spaghetti: if one strand is pulled, then the ramifications can be seen at the other side of the plate”
Better, but still, there is only one thread ...
Lehman’s eight laws of software evolution
(1974-1996 Here, only four of them)

I. Law of continuing change
*Systems must be continuously adapted or they become progressively less satisfactory to use. The variance between the system and its operational context leads to feedback pressure forcing change in the system.*

VI. Law of continuous growth
*Functional capability must be continually increased over a system’s lifetime to maintain user satisfaction. In any system implementation, requirements have to be constrained. Attributes will be omitted, these will become the irritants that trigger future demand for change. Feedback from the users.*

II. Law of increasing complexity
*As a system evolves, its complexity increases unless work is done to maintain or reduce it. If changes are made with no thought to system structure, complexity will increase and make future change harder. On the other hand, if resource is expended on work to combat complexity, less is available to system change. No matter how is balance is reconciled, the rate of system growth inevitably slows.*

VII. Law of declining quality
*Unless rigorously adapted to meet changes in the operational environment, system quality will appear to decline. A system is built on a set of assumptions, and however valid these are at the time, the changing world will tend to invalidate them. Unless steps are taken to identify and rectify this, system quality will appear to decline, especially in relation to alternative products that will come onto the market based on more recently formulated assumptions.*
What really takes time?

Ref.: The mythical man-month. Essays on software engineering.
Frederick P. Brooks, Jr. Addison-Wesley

“The mythical Man-Month” (I)

Essays on software engineering, by F. Brooks

✓ Basic, easy to read. Some recipes are just organisation recipes.
✓ “Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty <...> but the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it”.

Questions:
✓ What really takes time?
✓ How to make a group have better productivity?
✓ Can (software) tools improve the productivity?
“The mythical Man-Month” (II)

What takes time?

- Program module $x$ 2
- Integrated Program $x$ 4
- Product $x$ 2
- Integrated product $x$ 4

Documentation
Testing, portability
Maintenance

Clean I/Os
File formats
Interfaces

This is what we want to rely on, for our long-term research!

“The mythical Man-Month” (III)

How to make a group have a better productivity?

(hearing of a child takes nine months, no matter how many women are assigned)
- “The man-month as a unit for measuring the size of a job is a dangerous and deceptive myth”
- First, each person need training
- Then, software construction is a system effort

=> communication effort can dominate the decrease in individual task time brought by partitioning.
=> need : division of labor + specialisation of function

Also, in our case, each person has his own agenda, his own strengths and weaknesses ... Large productivity variations.
“The mythical Man-Month” (IV)

Conceptual integrity

- According to Brooks: “Conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.”
- How is conceptual integrity to be achieved?
- The small team concept (or even the surgical team)
- Solution: Disentangle system architecture and component implementation => centers of decision

“Cathedral building”: a large team and integrity!

“The mythical Man-Month” (V)

Additional: specificities of software engineering

- “Representation is the essence of programming” (meaning datastructures, file formats). Not flowcharts! => towards object-oriented.
- Ways to keep conceptual integrity: Manual - Documentation - Rules
- Self-documentation: the documentation is in the program. Adjustment to humans’ brain limited content!
“The mythical Man-Month” (VI)

Further ideas:

- A redesign is inevitable, for all components: the only constancy is change itself. So, plan the system for change... And have tools for version maintenance ...
- Program maintenance: unlike for a car, no cleaning, lubricating, repair of deterioration. The needed changes repair design "defects". These appear because of new functionalities to be implemented. Moreover, fixing a defect has a substantial (20%-50%) chance of introducing another! Importance of automatic testing.
- Adiabatic changes: quick debugging; references for tests; adding one component at a time.

The essence of programming

Frederick P. Brooks, Jr. Addison-Wesley
“No silver bullet” (1986) by Brooks (I)

Essence / Accidents in software development (refers to Aristotle categories)
- Essence: the fashioning of the complex conceptual structures that compose the abstract software entity
- Accident: the implementation process itself, actual typing, with hardware and software problems, loss of concentration by the programmer, ...

What is the ratio between them?
- Brooks argued that (in 1986) essence is more than 10% of development time, that it is inherently complex, and that it is not addressed by emerging software engineering concepts.

“No silver bullet” (II)

Essence of software development
- Construct of interlocking concepts: data sets, relationships among data items, algorithms and invocation of functions; need specification, design, testing, refinement
- Independent of the representation (language)
- “A scaling-up of a software entity is not merely a repetition of the same elements in larger size; it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly”
“No silver bullet” (III)

How to address the “essence” bottleneck?

- Use already existing software! **Software re-use.**
  - Conceptual work already done
  - Debugged, tested, I/O set-up!
  - Add “integrated product” to the system (need adequate licence)
  - Can be completely external (e.g. in our case, ROBODOC)
  - Can be internal re-use: need modularity!

- Rapid prototyping, then organically grow the software
  - Iterative extraction of product requirement: the prototype make real the conceptual structure specified, and allow adjusted set-up of “details”
  - Grow, not build software: incremental development, top-down design. Also psychological: one has something that works.

“No silver bullet” (IV)

Overall: no silver bullet to kill the werewolf

Building software takes some uncompressible human time, even if we eliminate the accidental difficulties, and attack the essential difficulties in an efficient way.

This principle was said to be, for software engineering, similar to Heisenberg’s principle, or Gödel’s theorem: a useful information!

Still, the question of group effort occurs ...
The linux experience

Ref.: The cathedral and the bazaar
http://www.tuxedo.org/~esr/writings/cathedral-bazaar

“The cathedral and the bazaar” (I)

Eric S. Raymond (1997). Early contributor to GNU. Analysis of LINUX project. Enthousiastic : read the intro !
Anatomize another open source project. Single out two dozens of propositions related to the process of software development, most of which related to the LINUX experience. Bazaar-like style of development ?!
Again : “software re-use”, “grow, not build”, “rapid prototyping”
LINUX used the GNU General Public Licence. “Free software” or “Open Source Software”. Key concept.
“The cathedral and the bazaar” (II)

1. Every good work of software starts by scratching a developer’s personal itch (Motivation)
5. When you lose interest in a program, your last duty is to hand it off to a competent successor
6. Treating your users as co-developers is your least-hassle route to rapid code improvements and effective debugging
8. Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone (Linus’ law)

“The cathedral and the bazaar” (III)

9. Smart data structures and dumb code works a lot better than the other way around.
11. The next best thing to having good ideas is recognizing good ideas from others. Sometimes the latter is better.
13. Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away.
19. Provided the development coordinator has a medium at least as good as the Internet, and known how to lead without coercion, many heads are inevitably better than one
The “Free” or “Open Source” software concept

Free for freedom, not price
- freedom 1: unlimited use for any purpose
- freedom 2: study and modify for your needs (need source access!)
- freedom 3: copy
- freedom 4: distribute modifications

From copyright to freedom (“copyleft”)
- copyright allows licensing
- licenses grants freedom

Terminology: Free software=Open source=Libre software

Free software licences...

Many types
- GNU General Public Licence
- GNU Lesser General Public License (links are possible)
- BSD licence
- X11 license, Perl license, Python license...
- public domain release

GNU General Public License (GPL)
“The licences for most softwares are designed to take away your freedom to share and change it. By contrast, the GNU GPL is intended to guarantee your freedom to share and change free software - to make sure the software is free for all its users” (http://www.gnu.org/copyleft/gpl.html)
- grants four freedoms
- protection of freedom
- « vaccination »
Coding Rules

NQ/ETSF coding rules

Really basic (and consensual) rules !

Documentation

- Choose carefully the names
- Write comments (in english, one line for each 10-20 line of codes at least)
- Describe the purpose, options and arguments of routines and functions
- Keep notes
- Do not try to produce “clever” coding, unless it is very well documented

Adhere to standards
Try to re-use code, try to use libraries
Modularity
Use a versioning tool
**Need of code-specific coding rules**

NQ/ETSF are valid for all projects in NQ/ETSF

- Irrespective of the language
- Irrespective of the software application
- Irrespective of the software tools chosen to manage the software, as well as their complexity (Bzr/SVN/... ? Make / Autotools ? Robodoc ?)

For one specific project, the language is know, the existing status of the software is known, with the accompanying choices

Coding rules might be implicit (the “style” of a project), but better be explicit: easier for newcomers to have the same style as the veterans.

Coding rules:
- helps reverse software engineering (= easier to understand code)
- helps to produce documentation
- helps conceptual integrity
- helps software re-use

**Summary**

Software engineering: a human science

Concepts of software engineering:
- Components
- Models for development / maintenance / evolution
- Lehmann’s law: continuing change, continuing growth, increasing complexity, decreasing quality
- Re-use, not because one saves coding, but because one saves much more
- Software construction is a system effort, conceptual integrity is needed
- Plan for changes: self-testing is mandatory
- Versioning tools
- Licence scheme might favour development
- Users might be made co-developers

Coding rules, needed for style integrity:
- helps reverse software engineering (= easier to understand code)
- helps to produce documentation
- helps conceptual integrity
- helps software re-use