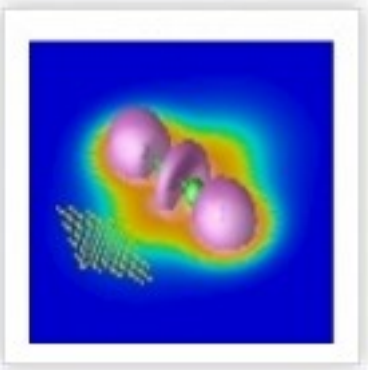
5th School on Time-Dependent Density-Functional Theory: Practical Sessions



E. K. U. Gross, M. A. L. Marques, F. Nogueira, A. Rubio, and A. Castro

cecam
Centre Européen de Calcul Atomique et Moléculaire
EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

ZCAM
ZARAGOZA SCIENTIFIC CENTER FOR ADVANCED MODELING

cecam-MMIP.de

CENTRO DE CIENCIAS DE BENASQUE PEDRO PASCUAL

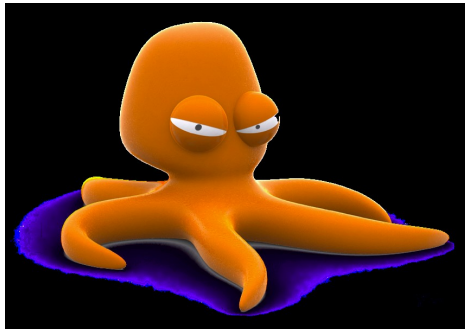
EUROPEAN SCIENCE FOUNDATION

GOBIERNO DE ESPAÑA
MINISTERIO DE ECONOMÍA Y COMPETITIVIDAD

Practical Sessions

1) Software development: construction of a basic TDDFT code.

2) Tutorial I: octopus



3) Tutorial II: yambo

Some Basic Concepts About Scientific Software Development

Source: “Basic concepts of software maintenance”, introductory talk given by Xavier Gonze at the CECAM 2010 Tutorial "Basic techniques and tools for the development and maintenance of atomic-scale software” (Zaragoza, Spain, June 21-25 2010)

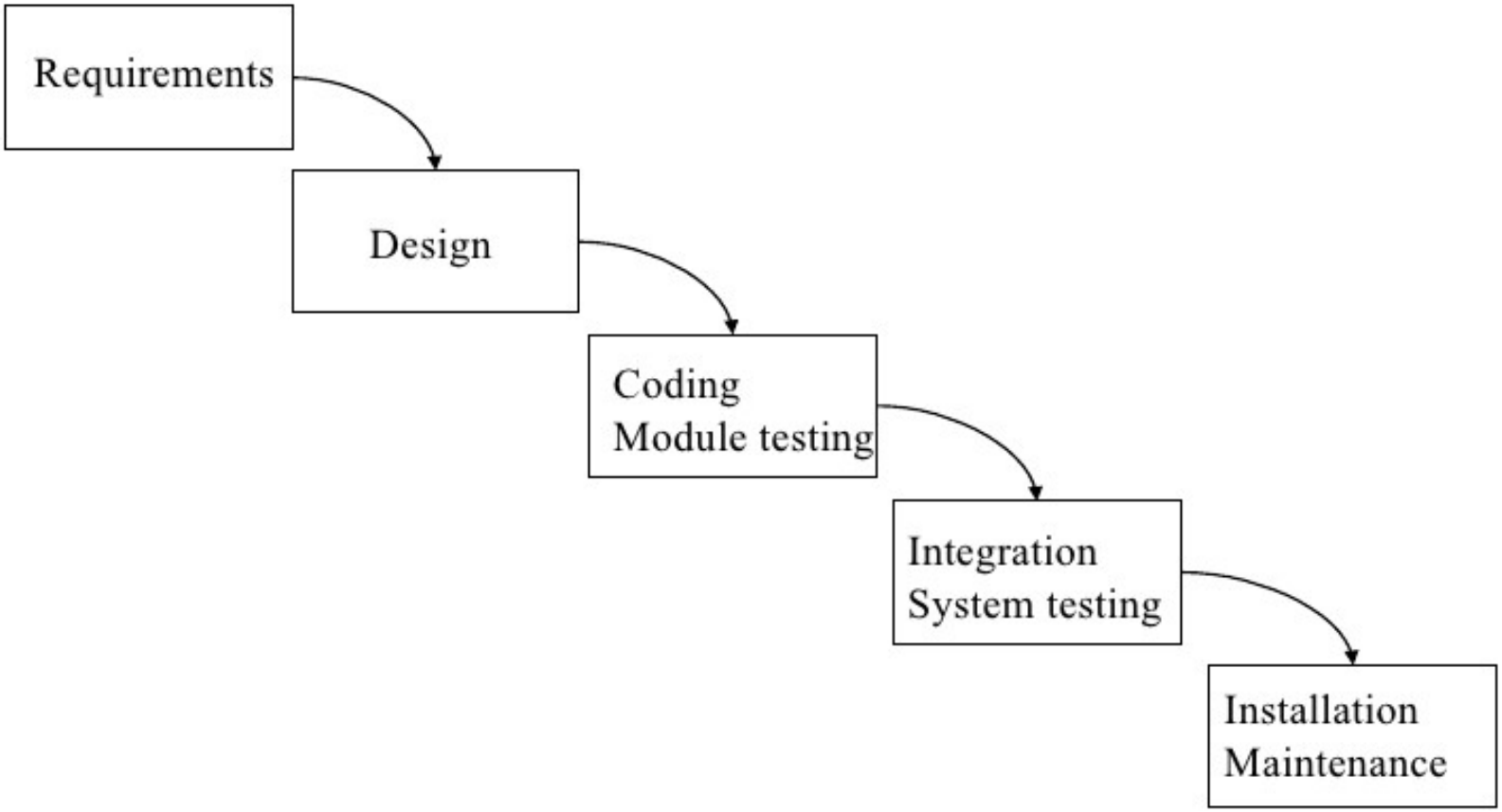
Software engineering for scientists

- The expertise of scientists is not software engineering (unless they are computer scientists...)
- Software engineering is best understood as a human science: it is about improving the productivity of the developers, who are human beings. It is not only about studying the software itself.
- “No single software engineering development will produce an order of magnitude improvement in programming productivity within ten years” (F. Brooks, *No Silver Bullet*, 1986).

Software

- Software is not *just code*.
- Software should be understood as the set of programs, documentation, and operating procedures by which computers can be made useful to people.
  - Program: source code, object code.
  - Documentation: articles, specifications, manuals, tutorials, description of I/O, internal description of data structures in any format,
  - Operating procedures: instructions or scripts to set up and use the program, instructions on how to treat failures, instructions or scripts to test the program.

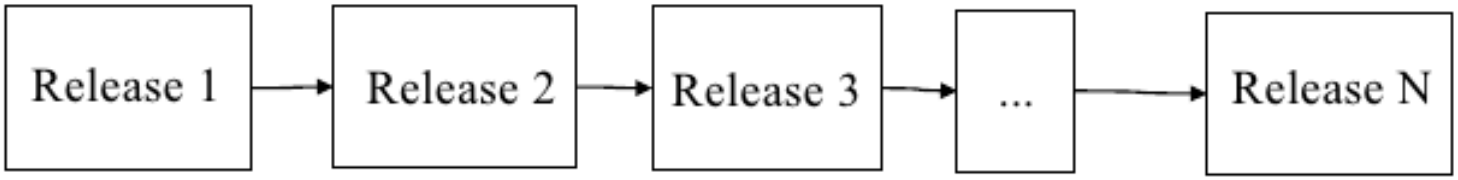
Software development: the waterfall model



Software maintenance

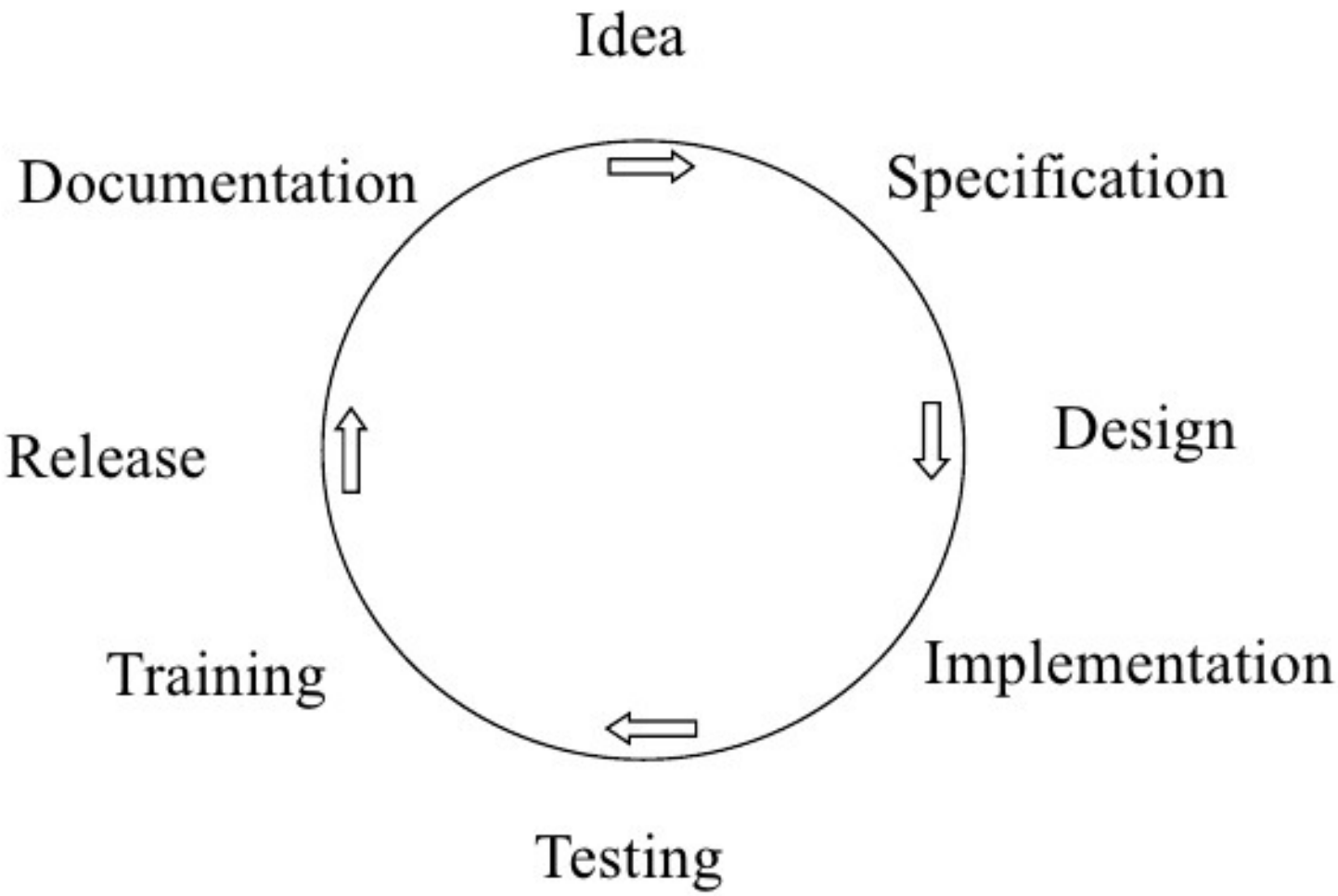
- Software maintenance: the continuous modification of a software product after delivery in order to:
  - Improve performance,
  - Add features,
  - Correct faults,
  - Adapt the product to other environments.
- In most of the cases, for atomic-scale software, we restart from existing software to add new functionalities. Maintenance is essential. Waterfall model is inadequate.

Software evolution

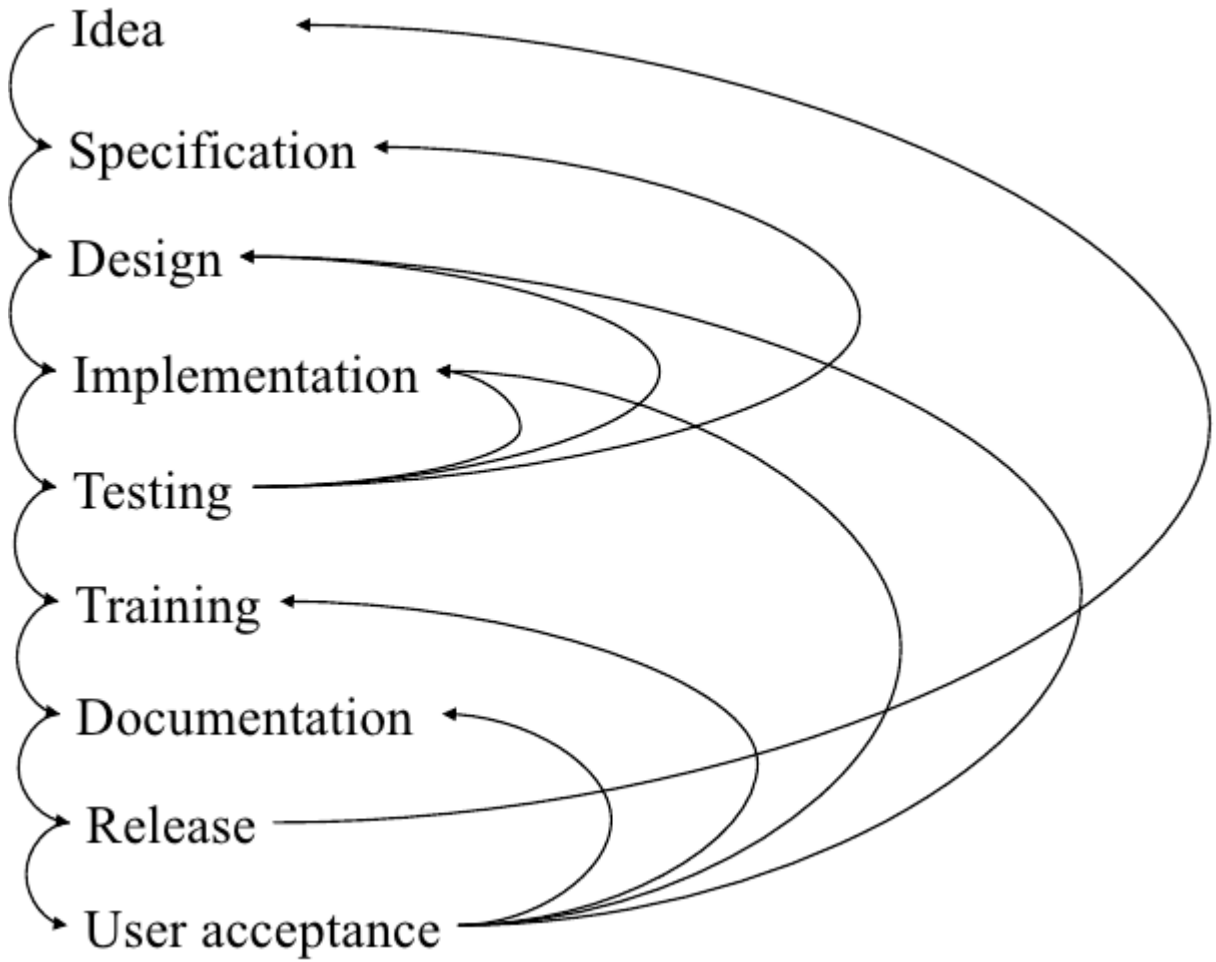


- Changes:
  - Adaptive changes.
  - Corrective changes.
  - Perfective changes.
  - Preventive changes.
- The problem with changes: “A badly structured program is like a plate of spaghetti: if one strand is pulled, then the ramifications can be seen at the other side of the plate.

A Maintenance Conscious Model



The reality of the development/maintenace



Some of “Lehman's Laws” of Software Evolution

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.

Some of “Lehman's Laws” of Software Evolution

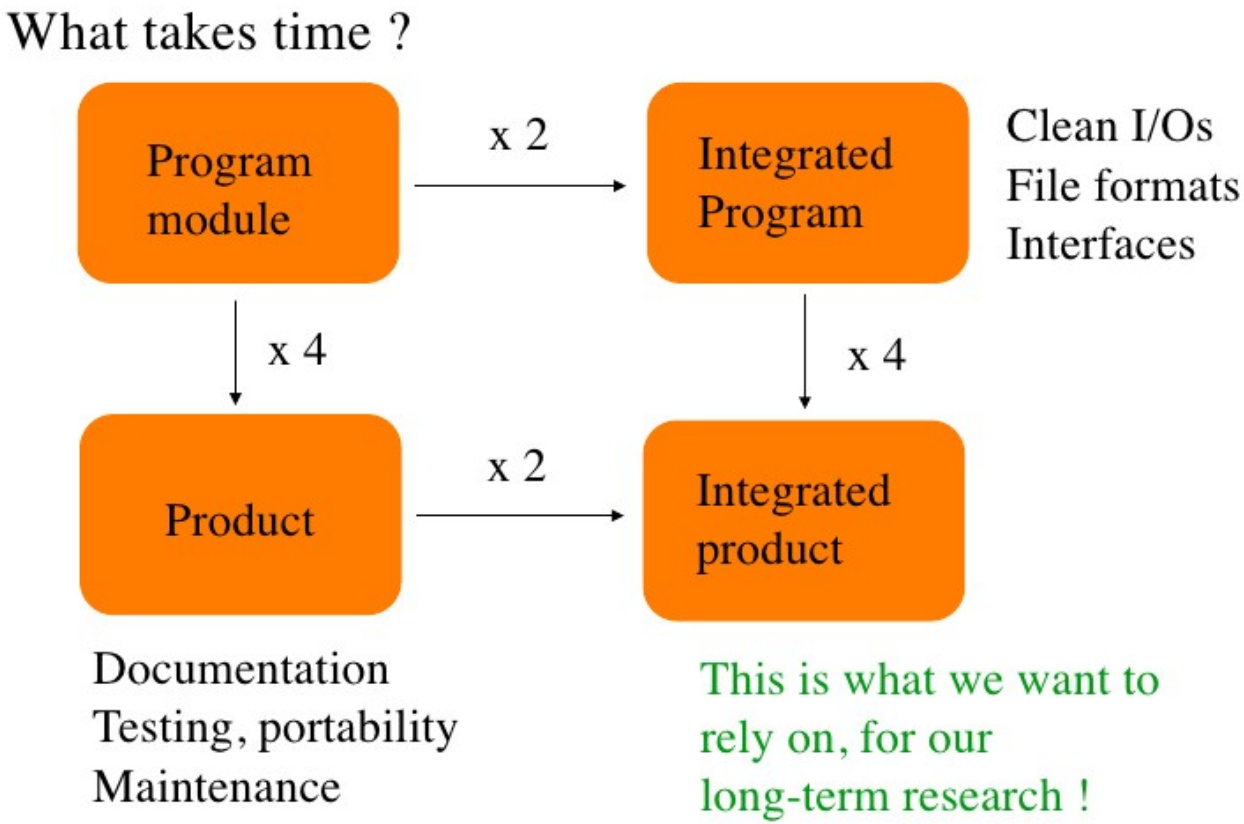
II. Law of increasing complexity

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 takes time?



“Essays on software engineering”, by F. Brooks

What takes time?

- Usually, scientific software development is a group effort.
- However, in Science each person has a different agenda, and different strengths and weaknesses.
- The training of a person should also be included in the total computation of time.
- Division of labor. But: in a similar manner to code parallelization, a bad “communication” can reduce the total productivity.

Conceptual Integrity

- All the parts of the code should reflect one clear set of design ideas. 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.”
- The key is to disentangle the system architecture from the component implementation.

Some (often forgotten) “standard” advices

1. A good data representation is the essence of programming.
2. Self-documentation: the program should, somehow, be self-explanatory (it is not just about adding comments!)
3. Bad comments are worse than no comments.
4. Version maintenance: svn, git, whatever. Plan the system for change.
5. Automatic testing.

Some (often forgotten) “standard” advices

6. Fixing one defect has a large probability of introducing another one.
7. Make “adiabatic changes”
8. Use already existing software! Even if later you plan to build your own because the existing one does not perform as well as you think it should.
9. Choose carefully the variable names.
10. Describe the purpose, options, and arguments of each procedure.

Some (often forgotten) “standard” advices

11. Adhere to standards.
12. Do not try to produce “clever” code, unless it is very well documented.
13. Modularity.

The Cathedral and the Bazaar

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

Eric S. Raymond

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
7. Release early. Release often. And listen to your customers.
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

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

Eric S. Raymond

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