Why Software Engineering?

(Lectures on High-performance Computing for Economists II)

Jesús Fernández-Villaverde[1] and Pablo Guerrón[2]

November 21, 2021

[1]University of Pennsylvania

[2]Boston College

Bill Bryson

"A" computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match.

Motivation

- You are taking a class on computational methods.
- Even if only because you need to complete your homework, you just became a software engineer (and not just a simple coder/developer!).
- Coding is, in part, an art ($\tau\acute{\epsilon}\chi\nu\eta$).
- But, in an even larger part, coding is about having good knowledge ($\grave{\epsilon}\pi\iota\sigma\tau\acute{\eta}\mu\eta$) of proven procedures.
- You can and should learn and use these procedures. This is not just to make things look pretty.
- You should neither reinvent the wheel nor refuse to use it!

The goal I

- To produce code that is:

1. *Correct*: we are scientist and we pursue correct answers.
2. *Efficient*: you want to get your Ph.D., to get tenure, to become an influential research economist in FINITE time.
   - 2.1 Coding time must be minimized.
   - 2.2 Running time must be minimized.
   - 2.3 Trade-off between coding and running time.
3. *Maintainable*: revise and resubmits, extensions of existing papers.

The goal II

4. *Reproducible*: other researchers (and your future selves; beware of `bit-rot`!) must be able to replicate your results.

5. *Documented*: other researchers (and your future selves) must be able to understand how it works.

6. *Scalable*: code that can be used by you and by other researchers as a base for further development.

7. *Portable*: code that can work across a reasonable range of machines.

The means

- Knowledge accumulated over decades in computational-intensive fields and by the industry.
- Software engineering⇒discipline that aims at developing reliable, efficient, correct, and maintainable software.
- Historical origin: NATO group in 1967, conferences in 1968 and 1969 about the *software crisis*.
- Standard part of a CS curriculum.

This class I

- We will cover some of the basics of software engineering (theory and tools).
- Adapted, though, to the requirements of an economist (at least, as determined by our own experience).
- For instance, you will probably not have different "releases" of a code, UML and design patterns will not be important, testing will be done in differently.
- At the same time, speed and reproducibility will be key.
- Also, we will cover material that it is taught in some basic courses on CS but that economists may be less familiar with (IDEs, Profilers, OOP,...).
- We will emphasize the idea that you want to use well-tested tools that give *you* as much control as possible within a reasonable cost.

This class II

- Brief introduction that cannot substitute:
  1. A real course on software engineering (and other techniques) in your local CS department.
  2. Standard books:
     - *Software Engineering (10th ed.)*, by Ian Sommerville.
     - *The Mythical Man-Month: Essays on Software Engineering (2nd ed.)*, by Fred Brooks.
     - *Code Complete: A Practical Handbook of Software Construction (2nd ed.)*, by Steve McConnell.
     - Other books we will mention throughout the lectures.
  3. Reading the technical documentation (RTFM).

This class III

- Additional resources:

  1. Own experience.
  2. Searching the internet (GIYF).
  3. Software carpentry: `http://software-carpentry.org/index.html`.
  4. Stack Overflow: `http://stackoverflow.com/`
  5. O'Reilly web page: `http://oreilly.com/`.
  6. Slant: `http://www.slant.co`
  7. `Youtube`.

Tools

1. Editors.
2. IDEs.
3. Report generators (`Jupyter` notebooks, `Markdown`, `Pandoc`,...).
4. Compilers.
5. Libraries (modules, toolboxes,...).
6. Make.
7. Lint and other static code analyzers.
8. Debuggers.
9. Profilers.

Techniques

1. Programming approaches (structured, OOP, functional,...).
2. Coding style.
3. Version control.
4. Prototyping.
5. Testing.
6. Performance optimization.
7. Paralellization (`OpenMP`, `MPI`, `OpenACC`,...)
8. Multilanguage programming (`Rcpp`,...).

Some final comments

- None of the contents of this class is a substitute for common sense, self-discipline, and hard work.
- Moreover, experience is more important than anything else.
- There is no silver bullet out there.
- Beware of the temptation of: "If I just update my OS/computer/app everything would be fine."

Swimming without water