# Clean Code / Code Optimization / Refactoring

## How to spend more time doing the important stuff

Programming Practices for Economics Research

Department of Economics, Univeristy of Zurich

Fall 2017

# Learning Objectives

- At the end of the session you will be able to:
  1. Highlight the importance of clean code
  2. Increase the understanding of computational bottlenecks
  3. Know what profiling means
  4. Know what debugging means
  5. Know where to read up advanced stuff

# Clean Code?

# What does clean code mean

* Code is clean, when it is understood easily by you
* your team mates
* your professor
* your grandmother

* in the contrary:
    * When you have to comment your comments you're lost

# Cost of Change

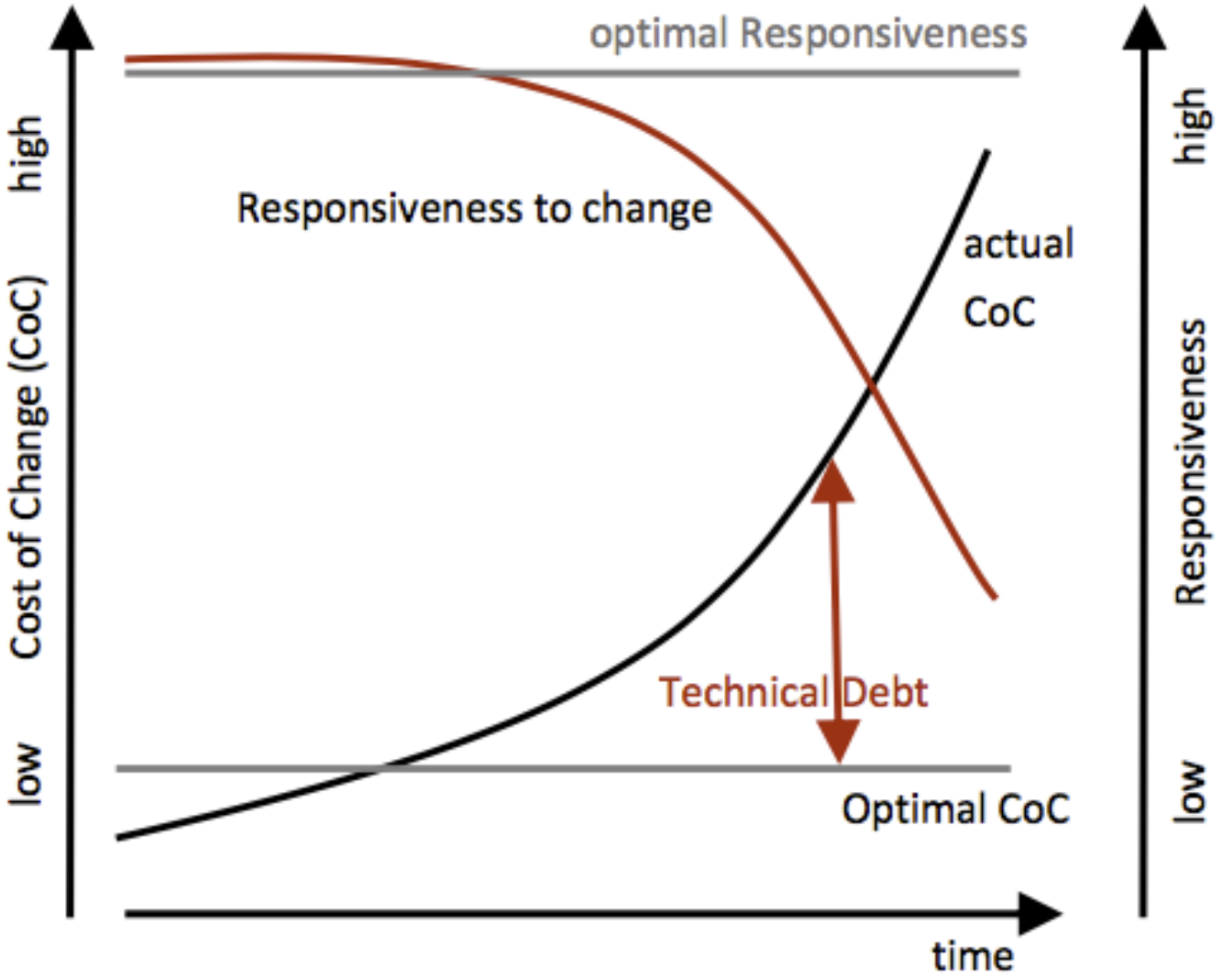


**Figure 1:** Cost of Change

# The newspaper Metaphor

*Think of a well-written newspaper article. You read it vertically. At the top you expect a headline that will tell you what the story is about and allows you to decide whether it is something you want to read. The first paragraph gives you a synopsis of the whole story, hiding all the details while giving you the broad-brush concepts. As you continue downward, the details increase until you have all the dates, names, quotes, claims, and other minutia. We would like a source file to be like a newspaper article. The name should be simple but explanatory. The name, by itself, should be sufficient to tell us whether we are in the right module or not. The topmost parts of the source file should provide the high-level downward, until at the end we find the lowest level functions and details in the source file.*

# The newspaper Metaphor

*A newspaper is composed of many articles; most are very small. Some are a bit larger. Very few contain as much text as a page can hold. This makes the newspaper usable. If the newspaper were just one long story containing a disorganized agglomeration of facts, dates, and names, then we simply would not read it.*

- from: `Clean Code: A Handbook of Agile Software Craftsmanship`

# Guiding Principles

# Rule 1: Write Programs for People Not Computers

- Code that a computer can understand $\neq$ code a human understands
- Does that difficult to understand code do what it's suppose to?
- Makes it hard for other collaborators and researchers to use your code
  - Future you *is an 'other' collaborator*

# Rule 1a: Write Many Short Scripts / Code

- Short-term memory can hold $7 \pm 2$ items
- Write short, readable functions, each taking only a few parameters
  - **Rule of thumb**: code looking very complex, you're probably doing it wrong.
  - Think of a function/script like a paragraph: limit it to one idea.
- *What not to do:* 5000 line scripts that execute an entire project
  - What does the variable on line 4100 mean?
  - How does it relate to it's initial defintion on line 1350?

# Rule 1b: Use meaningful variable names

- `p` less useful for short term memory than `price`
- `i`, `j` are (almost) OK for indices in small scopes
  - `ixx` and `jyy` might be better
  - `iDescription` is even better
- Be careful with the use of ambiguous names like `temp`

# Rule 1c: Make code style and formatting consistent

- Which rules don't matter — having rules does
- Brain assumes all differences are significant
  - Inconsistency slows comprehension
- Good ideas:
  - Keep each line of code within 80 characters
  - Consistent naming convention
  - Whitespace is your friend

# Rule 2: Let the Computer Do the Work

- Computers exist to repeat things quickly and accurately
- 99% accuracy vs 63% percent chance of error in *simple* tasks

# Rule 2a: Let the computer repeat and execute tasks

- Write little programs for everything
  - Even if they're called scripts, macros, or aliases
- Easy to do with text-based programming compared to GUIs
- We will search for the 'magic button'
  - One command that will execute your entire project
  - ... after you have written ordered instructions

# Rule 2b: Save recent commands when developing new code

- Most text-based interfaces do this automatically
  - Repeat recent operations using history
  - "Reproducibility in the small"
- Saving history supports "reproducibility in the large"
  - An accurate record of how a result was produced
- Have a 'sandbox' where you explore interactively and save your commands
  - Move it across to your main scripts once the pilot excercise works

# Rule 2c: Use a build tool to automate workflows

- Build tools originally developed for compiling programs
  - Adapted for managing code dependencies
- Workflow becomes explicit
- Will become your ‘magic button’

# Rule 3: Make Incremental Changes

- Most researchers don't have "requirements" when making changes to code
  - 'change it until it works how I want it to', then save it
- Code evolves in tandem with research
  - potentially in lumpy increments
- *Agile development*...

# Rule 3a: Work in small steps

- Frequent feedback and ability to correct the course when things go awry.
- People can concentrate for 45-90 minutes without a break
  - So size each burst of work to fit that
  - within a burst, save history of changes as you make (small) progress
- Longer cycle should be a week or two
- Design Issues and To Dos with these timelines in mind

# Rule 3b: Use a version control system

- Tracks changes
- Allows them to be undone
- Supports independent parallel development
- Essential for collaboration

***Email is not version control!***

# Rule 3c: Put everything that has been created manually in version control

- Not just software: papers, raw images, ...
  - Not gigabytes...
  - ...but metadata about those gigabytes
- Leave out things generated by the computer
  - Use build tools to reproduce those instead
  - Unless they take a very long time to create

# Rule 4: Don't Repeat Yourself (or Others)

- Anything repeated in two or more places will eventually be wrong in at least one
- If it's faster to 're-create and duplicate' than to discover or understand, fix what you're doing
  - Usual solution: re-modularize and import the "replicated" part

# Rule 4a: Define things once, and only once

- Every input must have a single authoritative representation in the system.
- Define something *exactly once*
  - Make calls to that input each time you need to reference it
  - Example: Define important parameters in a dictionary, import into each script

# Rule 4b: Modularize code rather than copying and pasting

- Reducing code cloning reduces error rates
  - Decreases testing needed to ensure your code does the right thing
  - . . . and increases comprehension
- a parameter dictionary is an example of modularization

# Rule 4c: Re-use (good) code instead of rewriting it

- Takes years to build high-quality numerical or statistical software
- Your time is better spent doing research on top of that, not trying to rewrite it to suit your needs,
  - Or avoid the errors we are getting from the software
- Code your advisor sends you may not be *good code*
  - Sometimes you can be better off rewriting, than comprehending
    - Recall in these situations we wanted to fix the problem
  - There is a generation gap

# Rule 5: Plan for Mistakes

- No single practice catches everything
- Practice defense in depth
- *Note: improving quality increases productivity*

# Rule 5a: Add assertions to programs to check their operation

- "This must be true here or there is an error"
  - Like diagnostic circuits in hardware
- No point proceeding if the program is broken...
  - ...and they serve as executable documentation
- Error messages are implemented and expected by other programmers
  - Indicators to improve your code

# Aside: Testing is Hard

- "If I knew what the right answer was, I'd have published by now."
- How to test your code works? Compare to
  1. Experimental or synthetic data
  2. Analytic solutions of simple problems
  3. Old (trusted) programs
- Documents what errors are acceptable

# Rule 5c: Turn bugs into test cases

- Write a test that fails when the bug is present
  - Work on the code until that test passes...
  - ...and no others are failing
- Write tests parallel to code
  - Otherwise you won't write them

# Rule 6: Optimize Software Only After It Works Correctly

- Experts find it hard to predict performance bottlenecks
  - Small changes can have dramatic impact on performance
- Get it right, then make it fast
- Don't be scared to ask questions about how you could further improve
  - e.g. the engineering team

# Rule 6a: Use a profiler to identify bottlenecks

- Reports how much time is spent on each line of code
- Re-check on new computers or when switching libraries
- Summarize across multiple tests

# Rule 6b: Write code in the highest-level language possible

- People write the same number of lines of code per hour regardless of language
- Use the most expressive language available to get the "right" version...
  - (Potentially) Rewrite core pieces in lower-level language to get the "fast" version
  - General trade-off: expressiveness vs speed
  - We don't explore: High level, fast langauge - Julia

# Rule 7: Document Design and Purpose, not Mechanics

- Documentation: make the next person's life easier
- Focus on what the code doesn't say
- Or doesn't say clearly
  - E.g., file formats
  - An example is worth a thousand words...

# Rule 7a: Document interfaces and reasons, not implementations

- Interfaces and reasons change more slowly than implementation details,
  - Documenting them is more efficient
- Most people care about using code more than understanding it

# Rule 7b: Refactor code instead of explaining

- Good code can be understood when read aloud
- Good programmers build libraries so that solving their problem is straightforward
- Too many comments?
  - You probably violated rule 1b
- No comments?
  - You *will* forget
- Commenting the comments?
  - Your lost!

# Rule 7c: Embed the documentation in the code

- Most languages have specially-formatted comments or strings
  - Likely to be kept up to date
  - More accessible than opening a supplementary file
- General movement towards "code in documentation" rather than vice versa
  - `Jupyter` and `Rmarkdown` are relatively extreme examples

# Rule 8: Collaborate

- Computers were invented to calculate
- The web was invented to collaborate
- Research is more fun when it's shared
  - Unfortunately you will likely be obligated to write one solo authored paper during your PhD
  - But *it really isn't a solo effort*

# Rule 8a: Use pre-merge code reviews

- Review changes before merging in version control
  - Develop on different branches or forks
- Significantly reduces errors
  - Good way to share knowledge
- It's what makes open source possible

# Rule 8b: Use pair programming

- When bringing someone new up to speed and when tackling particularly tricky problems useful to do it together
- Two people, one keyboard, one screen
  - An extreme form of code review
  - Puts you on the same page
  - Rarely done . . . probably our fault
- Can get a bit tiring if done tpp often

# Rule 8c: Use an issue tracking tool

- A shared to-do list
- Items can be assigned to people
- Supports comments, links to code and papers, etc.
- "Version control is where we've been, the issue tracker is where we're going"
- **Email is not an issue tracker!**
  - Although my advisor would seemingly disagree ;)

# Clean Code

# DRY

- Do it once - do it right

```
A = B * C * D
E = B * C * F
```

```
BC = B * C
A = BC * D
E = BC * F
```

# KISS

- Keep it simple - stupid!
- reduce complexity as much as possible

# Code Openness

## high density

```
C3 = @(a,c) ((1-F(a))*q*(1-func_tau(tau_eps(c),a,a_bar,a_ti
    /((1-func_tau(tau_eps(c),a,a_bar,a_tilde))^(1/eta)*(exp
```

## low density

```
c_vec = (1:cmax).';
tau = 1 - func_tau(tau_eps(c_vec), a_vec, a_bar, a_tilde);
tau_eta = tau .^(1 / eta);
norm_eta = (1 - eta)/eta;
tau_eta_exp = tau_eta .* (exp(norm_eta .* a_tilde) - 1);
exponent = exp(norm_eta .* (a_bar - a_vec .* a_tilde));

dividend = (1 - F) .* q .* tau_eta_exp + c_bar .*  exponent

C3 =  dividend ./ tau_eta_exp;
```

# Boy Scout Rule

- Leave the campground cleaner than you found it

# Debug

# Debug Vocabulary

- Breakpoint
- Callstack
- Step / Step In / Step Out
- Continue

# Matlab

# Javascript

# Python

# Profiling

# Matlab

# Javascript

# Python

# Can you summarize all of that?!

1. Use text-based interfaces
2. Turn history into scripts
3. Put everything in version control
4. Use test-driven development
5. Use debugger where needed
6. Profile your code
7. Optimize Bottlenecks ## Acknowledgements

- This course is designed after and borrows a lot from:
  - Effective Programming Practices for Economists, a course by Hans-Martin von Gaudecker
  - Software Carpentry and Data Carpentry designed by Greg Wilson
- The course material from above sources is made available under a Creative Commons Attribution License, as is this courses material.

# Programming Practices Team

Programming Practices for Economics Research was created by

* Lachlan Deer
* Adrian Etter
* Julian Langer
* Max Winkler

at the Department of Economics, University of Zurich. These slides are from the 2017 edition.