Programming Best Practices
Statistics 650/750
Week 2 Thursday

Christopher Genovese and Alex Reinhart

07 Sep 2017

Good Programming Practices: Key Concepts

Programming is a form of communication to *two* audiences: the computer and human readers (including future you).

As long as your code is syntactically correct, the computer will run it, for better or worse. But your code will be checked, studied, tested, documented, debugged, used, modified, generalized, and reused by humans. To get the most value from your time spent programming, you need to pay attention to how *humans* process your code.

Indeed, the features that characterize good code are very similar in spirit to the features that characterize good writing.

A Few Central Principles

There are many details to manage in a complex piece of code, and consequently there are many detailed choices to consider in practice. These include matters of

- style
- *naming*
- documentation
- organization
- design
- dependence
- error handling
- tooling

See the practices and design checklists in the `documents` repository or the reserve book *Code Complete* by Steve McConnell. These are worth reading and studying.

But for our purposes today, we can cover a lot of ground with only a few basic principles.

Write code to be read

Code *communicates ideas* and *describes abstractions*, often complicated ones. Its execution is like an unfolding story, with characters traveling along their own narrative arcs.

Try to maximize the ease and clarity with which the reader can process the code. Help them to

- understand the ideas/abstractions behind the code
- identify the characters/entities involved, and
- follow the story.

And remember to *prepare your reader for the information you are about to give.*

Here are a few implications of this principle.

- Format your code to make it easy to read
- Use meaningful, concrete, and descriptive names
- Arrange your code to bring out the centraal idea in each chunk
- Make critical relationships salient
- Structure your interfaces to present a clean and consistent abstraction
- Avoid hidden side effects and obscure features
- Use documentation to supplement code not mimic it

Be consistent unless you have a good reason not to

A *foolish* consistency may be the hobgoblin of little minds, but for programming, a practical consistency is helpful to in many ways.

Here are a few implications of this principle.

- Use consistent *formatting, spacing, and style*
- Use consistent *naming schemes* for variables, functions, classes, and files
- Use consistent documentation formatting, style, and scope
- Use consistent *interfaces* to functions and classes
- Use consistent error handling

Don't Repeat Yourself

Seriously, don't repeat yourself. It's inefficient to repeat yourself. So don't do it. Really.

Keep your code DRY! (Not WET – wasting everyone's time!)

Each piece of knowledge embodied in the code should have one unambiguous and authoritative representation.

Here are a few implications of this principle.

- If you find yourself repeating a piece of code, put it in a function.
- If you find yourself using a number or other literal, make it a *named constant.* (Besides a few basic cases such as 0, 1.)
- Documentation should not merely repeat what the code does but should add value For instance: why, who, when

It's easier to chew small pieces

Any stretch of code focuses on a few key ideas. Organiziing your code to bring out one idea at a time, rearranging as needed.

- Organize your code modularly (paragraphs, functions, files)
- Prefer functions that do *one* thing well
- Prefer orthogonality (decoupling)
- Prefer classes with a distinct purpose and identity

Keep the contract clear

Each function or class has an explicit contract behind it. *"I give you this, you give me that."*

Make that contract salient in your code, your tests, and your documentation.

An idea we will discuss: consider using assertions and pre/post conditions to check/enforce this contract.

Keep information on a need to know basis

Each function, class, and module in your code needs some information to do its job.

Give it the information it needs but no more.

Giving too much information couples parts of the code that should be independent, making them harder to test, debug, and reason about.

Objects in particular should "$_{\text{encapsulate}}$_" the information they contain quite jealously.

Make it run, make it right, make it fast – in that order.

Only Optimize the bottlenecks!

A Demonstration

- In your local copy of the `documents` repository, do a pull.
- Open the file `ClassFiles/week2/shift-the-mean-1.r` in an editor or in RStudio.

A few initial questions:

1. What does this code do? How might you figure it out?
2. What are the intended inputs?
3. What is the intended output?
4. Can you explain *why* anything is done the way it is?

We will think about this code with respect to the principles and consider some modifications to improve it.

First, look over the code for five minutes and consider a few initial questions:

1. What does this code do? How might you figure it out?
2. What are the intended inputs?
3. What is the intended output?

4. Can you explain *why* anything is done the way it is?

Second, a few modifications. See the files:

- README
- shift-the-mean-2.r
- shift-the-mean-3.r
- shift-the-mean-3-script.r

in the `ClassFiles/week2` directory of the `documents` repository.

An Interactive Exercise

- Create a local branch `week2T` off `master` in your assignments repository and check out that branch.
- Copy one of the files `ClassFiles/week2/nnk.py` or `ClassFiles/week2/nnk.r` into the respository directory.

We will think about this code and make a series of modifications, in light of the principles we have discussed today. At each stage, commit your changes within your local branch. We will **not** be pushing this local branch to github.

A few initial questions to consider as you examine the code:

1. What does this code do? How might you figure it out?
2. What are the intended inputs?
3. What is the intended output?
4. Can you explain *why* anything is done the way it is?
5. What works well here for clarity and readability? What does not?
6. Where is the code consistent or inconsistent?
7. Is there repeated code? What should you do about that?
8. Are the concepts within the code separated into meaningful chunks?
9. Is information properly encapsulated?

Rough activity time: 30 minutes

You are encouraged to discuss this with your neighbors as you work, but you should enter your own changes.

Debrief

Resources

- The practices and design checklists in the `documents` repository.
- The reserve book *Code Complete* by Steve McConnell.
- *The Pragmatic Programmer* by Andy Hunt and Dave Thomas

- Community style guides
  - Advanced R Style Guide by Hadley Wickham
  - PEP8 Style Guide for Python Code
  - Google Style Guides for many languages (including R, Python, C, C++, Java, and Lisp)

A Few Comments on Editors and IDEs

An **editor** is a general-purpose tool for editing files and many other tasks. An **IDE**, or Integrated Development Environment, provides an interface and tools for managing projects in a specific language or framework. Both have advantages as environments within which to construct your software, and in reality you will probably use some mixture of the two.

Examples of editors:

- Emacs
- Vim (see also Spacemacs)
- Sublime Text
- Atom
- nano
- vi (older but widely available)

Examples of IDEs include Rstudio (R), Ipython (python), Eclipse (Java), LightTable and Cursive (clojure), IntelliJ (JVM languages), VisualStudio (Javascript), and WebStorm (Javascript).

In addition, a generalized *notebook* style interaction is offered by Jupyter.

These are all good tools, and they will pay back the time spent mastering them. I would argue that even if you use an IDE for some languages or projects, **mastering a good editor** will make you more efficient.

Good editors are radically customizable, broadly functional, and surprisingly powerful.

I use Emacs, an editor that got its start in the 1960s and has been continuously updated since. (The Emacs git repository has history that stretches back over 30 years and has more than 130,000 commits and 600 committers.)

Emacs is "tinkerer's editor." It can be customized in almost every way conceivable to allow you to optimize your environment and workflow to your needs. It has a learning curve, but that repays the effort spent on it.

Getting Started with Emacs:

- Emacs Prelude: `https://github.com/bbatsov/prelude`
- Emacs Starter Kit: `https://github.com/technomancy/emacs-starter-kit`
- Spacemacs: `http://spacemacs.org/`
- Steve Purcell's .emacs.d: `https://github.com/purcell/emacs.d`
- Mastering Emacs by Mickey Petersen