Enhance Your Productivity and Software Quality with Techniques from Silicon Valley

Benjamin S. Skrainka
The Harris School of Public Policy
University of Chicago

skrainka@uchicago.edu

April 2, 2012

The Big Picture

Whether you like it or not you are a software engineer:

- Much wisdom we can learn from Silicon Valley
- Much technology we can exploit
- About increasing your productivity
- About reproducible results (scientific method, getting sued)

$\Rightarrow$ much of the cost of software is maintenance!

Good Code

Good code is:

- Easy to maintain
- Easy to extend
- Easy to understand ... even after a six month break!
- Straight-forward and direct ... no side-effects or surprises!
- Reads like English (or some other human language)

When you feel ‘friction’ something is wrong. . .

Some Questions

Before writing a line of code, ask yourself:

- What will this code be used for?
- How often will it be used?
- How might it evolve? How can I isolate myself from possible changes, such as using a different solver?
- What part of this code is generic and what part problem-specific? i.e,
    - What can I reuse?
    - What should I abstract into a library?

Roadmap

Tactical Programming

Designing Better Software

Debugging and Optimization

Software Development Tools

Goals of Tactical Programming

Tactics – aka *programing style* – are about structuring your code so that:

- Easier to read
- Easier to detect bugs
- Easier to understand
- Easier to extend
- i.e., to minimize the costs of working with your code
- In short, you want to minimize (or eliminate) complexity

$\Rightarrow$ increased productivity for free!!!

Use A Coding Convention

A good coding convention makes your code read like a good story and makes your intent clear:

- Naming of functions, variables, and filenames
- Grouping and layout of code such as braces
- Modification history
- Comments
- Respect the local coding convention when working on code

Choose a convention and stick to it!

Structure Your Code

Group logical chunks of code together:

- Separate larger blocks with comments
  - Create horizontal lines of '-', '=', etc. to indicate higher-level groupings
  - Just like books are organized into chapters, sections, subsections, etc.
  - Use vertical space (blank lines) to set off lower-level chunks of code
- Use white space:
  - Put space around operators `=`, `+`, `-`, `*`, `/` and inside of `{}`, `()`, and `[]`
  - Choose a sensible indentation scheme, such as two spaces
  - Beware of tabs ...
- Anything longer than 1-2 screenfuls of code should be a separate function

Choose Good Names

Choose names which describe the role of a function or variable:

- Separate multiple words with CamelCase or '_'
- Function names should start or end with a verb: `CalcMarketShares()`
- Encode type information into variable names: float, int, matrix, vector, etc.
- One variable definition per line + a comment
- Start indexes with `ix`: `ixStart`, `ixStop`
- One 'p' for each level of pointer indirection

Bad Names: `p, x, y, n, i, j, k, l, jfunc1`

Good Names: `dwPriceFood, dwExcessDemand, dwIncome, nGoods, vProb, IntegrateMarketShares(), IsValid(), ix, jx, kx, pHHData`

Braces

There are two main styles for braces:

1TBS/K+R/etc.

```
if( IsBadState() ) {
  fixProblem() ;
}
```

Allman/GNU/etc.

```
if( IsBadState() )
{
  fixProblem() ;
}
```

Write Comments

Comments are important:

- History of changes
- Why you did something, not what you did
- Explain anything tricky – you won't remember why you did something next month...
- Use comments and white space to convey logical structure of code on small, medium, and large scales
- Start any file with a short one line comment explaining purpose of module
- Document function interfaces and any quirks

One Place Only

Strive to minimize duplication:

- Are you writing code with cut and paste? $\Rightarrow$ abstract it into a function ...
- Use constants whenever possible:
  - Define all numbers and constants in one place only
  - Define indexes (with good names) for different columns or rows in a matrix, especially for MATLAB
  - Make arguments `const` when only used for input
  - No hard-coded numbers!!!
- Automate what you can:
  - macros
  - templates
- When you have to make changes, it is easier if you only have to modify it in one place!

Order of Operations

Don't abuse order of operations:

- Only use order of operations for +, -, /, *
- For everything else, use parentheses!
- Avoid clever tricks and side-effects . . . unless necessary for performance in which case you need to document how the trick works

MATLAB Tricks

Here are a couple tricks to improve your MATLAB code:

- Use cells by commenting the start of a section with `%%`:
  - Group a logically-related block of code
  - Rerun the cell with `CTRL + RETURN`
- Handle errors with `keyboard`
- Store column indexes in a structure: `Index.Price`, `Index.Income, ...`
- Wrap related variables into a structure:

```
ChoiceData.X    = mCovariates ;
ChoiceData.Y    = vChoices ;
ChoiceData.nObs = length( vChoices ) ;
```

How to Design Software

Much of good software design is based on:

- Planning ahead for maintenance (one of the biggest costs of most projects) and future extensions
- Writing testable code
- Choosing good abstractions
  - The right data structures
  - The right algorithms
- Designing good interfaces

The goal is to minimize (hide) complexity, reduce friction, and avoid duplicating code

What to Worry About

Questions to ponder:

- Where will my code run?
- What technologies does it depend on?
- How is it likely to change?
- How will it be used?
- How often will it be used?
- How can I test it?

$\Rightarrow$ Write a design document!!! You don't have time not to plan…

Trade-offs

You need to evaluate many trade-offs:

- Speed vs. robustness
- Speed vs. memory usage
- Speed vs. maintainability (e.g. fast code may require unreadable optimizations)
- Development time vs. code quality (performance, maintainability, reusability)
- Quality vs. frequency of use

Interfaces

An interface is a contract:

- Clear and easy to remember
- Use the same interface for similar objects/operations
- Promotes loose coupling and reuse
- Minimizes maintenance headaches by isolating implementation from interface
- Publish the interface in a header file:
  - Separate from the implementation file
  - Protect with include guards if using C preprocessor
  - May need second header file for private information
- Only a few arguments – put any more in a `struct`

Functions

Functions are a key technique to eliminate complexity:

- A function should do one thing and do it well
  - Facilitates composition to solve more complex problems
  - Facilitates reuse, debugging, maintenance, and extension
  - Facilitates understanding
- Follow the Unix model:
  - Write simple commands and functions
  - Easy to test
  - Easy to combine
- Use to express interfaces
- Use to break up any code which exceeds a couple screenfuls

Practice Information Hiding

Hiding information and implementation make your code more robust:

- Put only the minimum amount of information in the public name space
- Make everything else `private` or `static`
- Prevent unintentional access
- Now changing implementation details won't break other code
- Encapsulate state information in a `struct`, not a global if possible
- Avoid global variables!!! They often lead to race conditions...

Reusable Code

Write reusable code:

- Collect general tools and components into a common library
- Reuse for faster development of other projects
- Decrease bugs through use of production code

Corollary: reuse (high quality) existing software libraries and components:

- Don't reinvent the wheel
- Benefit from code which has already been debugged

Defensive Programming I

Write code to facilitate debugging:

- Modularize functionality
- E.g., access shared resources or special facilities only through one library: `splineLib`, `splineCreate`, `splineEval`, `splineDelete`, ...
- If a bug occurs then it is:
  1. In the library
  2. Use of the library

Defensive Programming II

Isolate your code from things which might change:

- Third party software: MPI, solvers, libraries
- Platform-specific technologies: OS-specific APIs
- Buggy code by co-workers ('software condom')

I.e., write a thin layer between your code and volatile resources

Defensive Programming III

Trust but verify:

- Verify that input is sane:
  - When reading in configuration information and data at start of program
  - Inside functions:
    - Are the arguments correct?
    - Did the computation produce a feasible value? E.g., is consumption non-negative?
- Tools:
  - keyboard in MATLAB
  - #include <cassert> in C++
- Automate everything you can:
  - Multiple steps and copying data lead to avoidable errors
  - One to hit one button to produce your paper!

Test Driven Development

TDD uses unit tests and a tight *write-test-debug* cycle to catch bugs early:

- Unit tests are short pieces of code which exercise all (or the key) paths through a function
  - The sooner you find a bug, the cheaper/easier it is to fix
  - Immediately program to an interface to verify design decisions
  - Catch bugs caused by other changes to system
- Many popular unit test frame works are available: `junit`, `cunit`, `boost::test`, etc.
- Interpreted languages provide a similar productivity boost by letting you test code interactively as you develop it.
- TDD is a philosophy for software development
- Refactor code which is unwieldy

Refactoring

Refactor when necessary:

- Refactoring means redesigning and/or rewritting code when it becomes brittle, unwieldy, or starts to rot
- Do in presence of unit tests to ensure that you reimplement code correctly
- Brooks (1995): ‘Plan to throw one away.’
- It is time to refactor when you feel friction and frustration when working on code.
- See Fowler et al (1999) ‘Refactoring’.

Debugging

Unfortunately, you will make mistakes:

- Learn to use the debugger
- Don't sprinkle your code with `printf`, `WRITE`, etc.:
  - Obscures code readability
  - I/O slows code considerably
- Add diagnostic logging to large applications
  - Message logging to files
  - Print messages to screen in debug version only
- Step through your code in the debugger: you might be surprised by how it actually executes...
- Will boost productivity considerably!

Debugging

Use the C preprocessor to facilitate debugging (even in FORTRAN):

```
#ifdef USE_DIAG
#define DIAG_PRINT      PRINT *,
#else
#define DIAG_PRINT      !
#endif
```

Must use correct compiler flags: `-fpp -allow no_fppcomments`

Optimization

Your intuition about what needs optimization is often wrong:

- First, get your code to work correctly
- Then optimize:
  - Measure code with a profiler
  - Optimize what needs optimizing
- MATLAB has a built-in optimizer
- For gcc, use gperf

Vectorization

Write loops which support vectorization (unrolling):

- Use:
  - Straight-line code
  - Vector (array) data only
  - Local variables
  - Assignment statements only
  - Pre-defined (constant) exit condition
- Avoid:
  - Function calls
  - Non-mathematical operations (which are difficult to vectorize)
  - Mixing vectorizable types
  - Memory access patterns which prevent vectorization – i.e. where one statement access future and/or previous array elements

Version Control

Manage all of your code (and $\LaTeX$) with version control:

- Provides a safety net when programming
- Stores code in a repository which tracks changes anyone makes to code
- Synchronize changes across computers
- (Automatically) merge your changes with your co-authors' changes
- Revert to earlier versions
- Manage different branches of code
- Tag key milestones

Popular flavors: Subversion (svn), CVS, git, and hg

Make

Make manages building software:

- Checks dependencies
- Builds only what is necessary
- Allows abstraction of build process:
  - Tools
  - Options
  - Platform specific details
- Promotes portability

Editor and OS

Invest in your tools:

- ‘Choose your editor with more care than you would your spouse because you will spend more time with your editor, even after the spouse is gone.’ – Harry J. Paarsch
  - Learn to use a good programming editor: Vi, Emacs, jEdit, Notepad++, Eclipse, etc.
  - Will increase your productivity
- Same applies to your OS – get some Unix in your life!
- etags, cscope, ctree, etc. make it easy to explore code
- Eclipse, MS Visual Studio have powerful tools as well