Enhance Your Productivity and Software Quality with Techniques from Silicon Valley

Benjamin S. Skrainka
University College London
Institute for Fiscal Studies

August 28, 2009

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)

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?
- What can I reuse?
- What should be a reusable library or toolbox?

Roadmap

Tactical Programming

Designing Better Software

Debugging and Optimization

Software Development Tools

Goals of Tactical Programming

Tactics are about structuring your code so that:

- Easier to read
- Easier to detect bugs
- Easier to understand
- Easier to extend

$\Rightarrow$ increased productivity for free!!!

Use A Coding Convention

A good coding convention makes your code read like a good storey:

- Choose good names for functions and variables which clearly convey their purpose
- Group logical blocks of code with space and comments
- Separate tokens with space
- 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 ...
- Put anything longer than 1-2 screenfuls of code in 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
- 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 create 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 only one place
  - Define indexes (with good names) for different columns or rows in a matrix
  - 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

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

Questions to ponder:

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

Design Document

'This code is too complicated to have a design document' – engineer at a major Internet portal

- You don't have time not to plan
- The more complicated your project, the more important it is to get the design right
- Think about use cases:
  - What are key parts of application?
  - How do they interact?
  - Draw a picture with Visio or Dia

Cultivate Good Habits

Practice OO Principles:

- Encapsulation
- Polymorphism
- Inheritance
- Interfaces: Open to extension, but closed to modification

Practice 'genericity', i.e., Templates
$\Rightarrow$ OO forces you to follow good programming practices: information hiding, loose coupling, and reuse

Interfaces

An interface is a contract:

- Clear and easy to remember
- 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`

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`
- Prevents unintentional access
- Now changing implementation details won't break other code

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

Reentrancy

Good code is reentrant:

- Reentrant code $\equiv$ code which is thread-safe, i.e. it can be executed by multiple threads at once with the same result:
  - Race condition: when order of execution affects correctness
  - Appears as an intermittent bug
- Uses local storage (arguments, stack variables) or pointer to a control object (heap)
- Facilitates parallelization
- Avoids race conditions
- Global variables are evil evil evil.

State Information

Some times you must pass around state information:

- Encapsulate it in an object
- Pass around a pointer to that object
- Do not use global variables:
  - Error prone
  - Hard to debug
  - Can lead to race conditions when modified inconsistently in multiple locations

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

Make the compiler work for you:

- The sooner you catch an error, the cheaper it is to fix
- Enable strictest compiler warnings (e.g., `% gcc -Wall -pedantic ...`)
- Try to eliminate all compiler warnings from your code!
- Program so compiler catches errors, e.g.:

```
if( 0 == nRead )
  handleError() ;
```

- Use `const`
- Compile on multiple compilers
- Compile C with a C++ compiler

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

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

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

Version Control is a safety net for programmers:

- Manages every version of your code
- Supports distributed software development
- Supports multiple developers
- Keeps everything synchronized
- Automatically merges different changes to the same code
- Common examples: SVN, CVS, hg, ClearCase, Perforce, ...

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:

- 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