Take Control of Your Code

Essential Software Development Tools for Engineers and Scientists

Steve Eddins
The MathWorks, Inc.

April 25, 2008
Yale University

© 2008 The MathWorks, Inc.

Handout PDF:
http://blogs.mathworks.com/images/steve/2008/handout_yale_20080425.pdf

Engineers and Scientists Write Code

Engineers and scientists in all disciplines rely heavily on software.

Much of the software is custom code, written in a variety of languages: Fortran, C, C++, MATLAB®, Java, HDL, PHP, etc.

Applications include data analysis, simulation, embedded controllers, interfacing with lab instruments, etc.

Software projects vary in size from hundred-line MATLAB scripts to applications requiring tens of thousands of lines.

The dilemma: You're not a full-time programmer, and you don't especially want to become one. But you need a lot of software to get your job done. So when software projects stumble (as they so often do), your productivity suffers.

Typical problems:

• Last month's results can't be reproduced because the software changed.

• Collaborators use different versions of the software, resulting in confusion.

• Software can be *brittle* – difficult to modify without breaking.

Improve Your Approach: Use the Basic Tools

You need not be a master carpenter or plumber to handle basic maintenance and handy work around your home. But wouldn't it be frustrating if you didn't even have a screwdriver?

To improve your approach and take control of your code:

- Use version control
- Sniff out bad code smells
- Reduce function complexity
- Refactor your code

These techniques can all be applied to any programming problem, in any language or environment. No special software engineering expertise is needed. Applying even one of these techniques will benefit your project.

Copyright 2008 The MathWorks, Inc.

Use Version Control

Demo

Have you ever started making a few "simple" changes, spent a day or so working on them, and then changed your mind? Maybe you realized the changes were misguided, or that they were more complicated than you expected.

You should be using VERSION CONTROL!

A version control system tracks every change made to source code and other documents – what the change was, who made it, and when it was made. It can roll back a file to a previous state, show you what has changed since last month, help you reliably reproduce results, and facilitate collaboration.

With a version control system, you can always revert to the latest version in the repository, or to any previous version.

Key vocabulary:

Repository – container of the current version and all previous versions of all files associated with a project; managed by the version control system

Working copy – a copy of project files on your own computer that you can modify

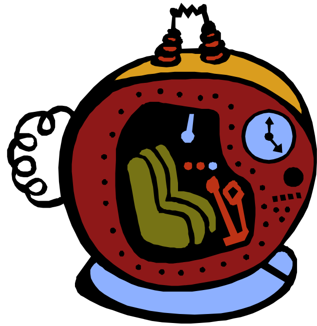
Diff – a display of the differences between a file in your working copy and the version of the file in the repository

Commit – submit modified files to the repository

Log – record of developer notes attached to every change

Copyright 2008 The MathWorks, Inc.

Use Version Control – Go Back in Time



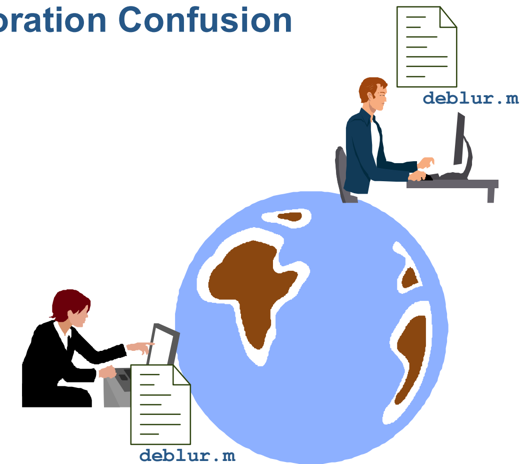
Can you reproduce results from a paper you wrote two years ago?

Reproducibility is an important concept in many scientific and engineering disciplines. But reproducing results based on software can be difficult or impossible without version control.

When you get in the habit of using version control, you'll be able to reproduce past results at will.

Copyright 2008 The MathWorks, Inc.

Use Version Control – Eliminate Collaboration Confusion



"Version control is indispensable on team projects." [McConnell, *Code Complete*, page 668]

Version control resolves the thorny question of "who has the latest version of this file?"

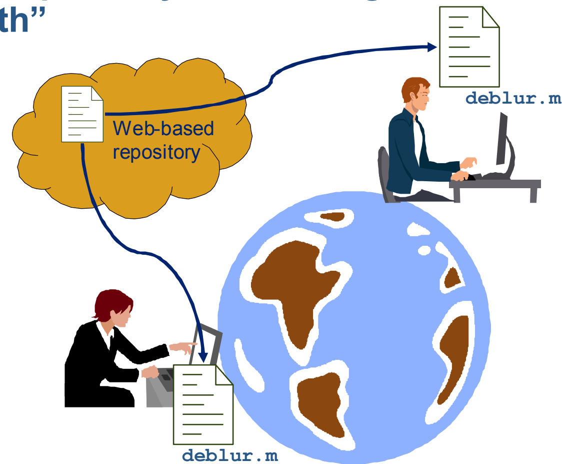
Version control shows you at a glance whether anyone else on the team has changed any of the files you have.

Version control lets you easily update your files to the latest versions.

Version control on a server allows anyone (with permission) to access the latest files at any time.

Copyright 2008 The MathWorks, Inc.

The Repository—Your Single Source of "Truth"



Two collaborators are working on the same set of files. Whose copy of `deblur.m` is the latest? How do you keep track?

A version control system completely and unambiguously answers this question. No individual is the keeper of the latest files. Instead, the repository is the sole source of "truth."

No one changes files in the repository directly. The version control system manages all repository changes.

Collaboration doesn't work unless all collaborators have ready access to the files.

All major version control systems have Internet-based servers and clients.

If you can use a Web browser, you can use a version control client.

If you or someone else can set up a repository server on the Internet, then anyone with the right access permission and a version control client can participate.

If you don't have the Web expertise and a server available, inexpensive commercial services (and some free services as well) can do it for you.

Copyright 2008 The MathWorks, Inc.

Ann and George Collaborate

1. Ann updates her files.
2. Ann edits deblur.m.
3. After testing her changes, Ann commits them and goes to lunch.
4. George updates his files.
5. George edits deblur.m.
6. After testing his changes, George commits them and goes to a meeting.
7. Ann returns from lunch and makes more changes to deblur.m.
8. Ann tests her changes and tries to commit them.
9. The version control system, quiet and happy until now, tells Ann she's out of sync with the changes George made. It helps her to merge George's changes with hers.

Some things to note about this sequence:

- Ann and George do not send each other their files directly.
- They can freely choose to work on their local copy of their files at any time.
- Each of them is in the habit of committing their work often.

Copyright 2008 The MathWorks, Inc.

Use Version Control

Pragmatic Programmer Tip:

Always Use Version Control!

Pragmatic Programmer Tip #23: Always Use Source Code Control

"Always. Even if you are a single-person team on a one-week project. Even if it's a 'throw-away' prototype. […] Even if we're not working on a project, our day-to-day work is secured in a repository." [Hunt and Thomas, page 88]

If you work with one or more people on the same project ... Use version control.

If you write software libraries or applications that other people use ... Use version control.

Even if you're working by yourself on "throw-away" code ... Use version control!

Copyright 2008 The MathWorks, Inc.

Choose Your Version Control Tools

To help you get started, let me recommend some specific tools. These are not the only good choices available to you. Other tools may be a better fit for your particular circumstances. But I've tried these tools, and I'm confident they will work for most people in most situations.

Copyright 2008 The MathWorks, Inc.

Use Subversion

Use Subversion for your version control system.

Subversion is a modern, free, actively maintained, open-source version control system. It works well, and it has easy-to-use clients on all major platforms. The Windows installer, which I have tried, couldn't be easier. There are several good books about Subversion, including one that's free. And there are several commercial and free repository hosting providers.

Compared to CVS (an older, widely used system), Subversion handles directories, file renaming, branching, and binary files more robustly and efficiently.

Obtain Subversion from:

subversion.tigris.org

Other version control system choices:

• CVS

• Commercial offerings such as Perforce

There are many others. For a comprehensive list, see:

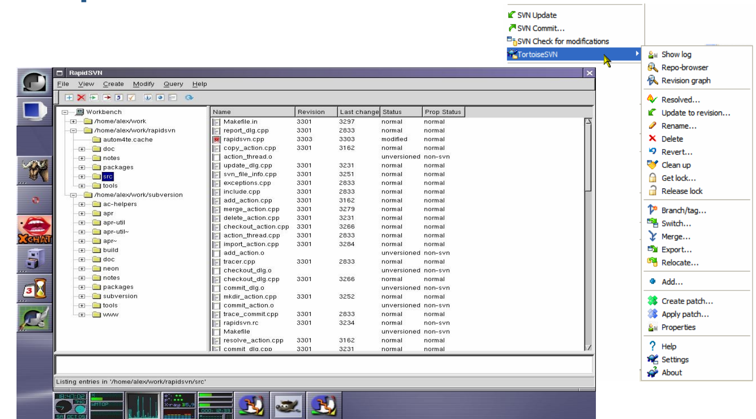
en.wikipedia.org/wiki/List_of_revision_control_software

For a comparison of different version control systems, see:

en.wikipedia.org/wiki/Comparison_of_revision_control_software

Copyright 2008 The MathWorks, Inc.

Use TortoiseSVN on Windows RapidSVN on Linux or MacOS



On all platforms, you can interact with a Subversion repository using the command-line interface. If command-line interfaces make you happy, stop here.

Easy-to-use GUI tools are available on all major platforms.

On Windows, I like TortoiseSVN because it integrates into the Windows Explorer. I've been using TortoiseSVN in my demos. It is available from:

tortoisesvn.tigris.org

Another GUI interface, RapidSVN, is available on most major platforms, including Windows, Linux, and Macintosh. RapidSVN is available from:

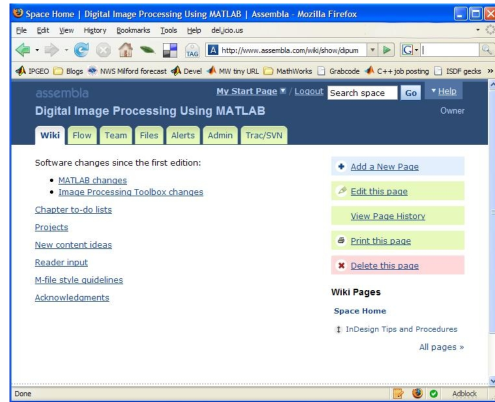
rapidsvn.tigris.org

For Web-based repository browsing, try WebSVN, available from:

websvn.tigris.org

Copyright 2008 The MathWorks, Inc.

Use Web-Based Repository Providers



Not everyone knows how to set up a publicly accessible Web server that's configured properly to host a Subversion repository. Also, you'd need to learn some administrative details, such as how to control who has read and/or write access to the repository. You'd also have to worry about doing regular backups.

Don't let these concerns stop you. There are services out there that will set everything up for you. Some are commercial, some are free.

For the book *Digital Image Processing Using MATLAB*, my coauthors and I have been lately using Assembla.com.

Try a Google search on "subversion hosting" or "software project hosting."

Copyright 2008 The MathWorks, Inc.

Improve Code Quality

Communicate with people, not computers

Minimize complexity

What a challenging topic! It could be an entire course of study. How can we usefully spend a few minutes talking about it?

Here are a few ideas that are widely accepted, effective, practical, and easy to apply:

- Learn to identify bad code smells
- Keep function complexity down
- Refactor your code regularly

These concepts and techniques can be learned and applied individually, but they also reinforce each other.

Along the way, remember these two fundamental principles that underlie most techniques for assessing and improving the quality of code:

- Write code to communicate effectively with people, not computers
- Write and organize code to minimize complexity

When someone tells you a "rule" for writing code, evaluate it against these principles.

Copyright 2008 The MathWorks, Inc.

Recognize “Bad Code Smells”

A *bad code smell* is a characteristic of code that causes an experienced programmer to pause, wrinkle his or her nose, and think, “There’s a good chance something bad is going to happen here.” The odor metaphor resonates strongly because smell is such a powerful memory cue.

Fowler’s *Refactoring* includes a catalog of smells in Chapter 3. Read them. Find two or three that make sense to you and learn to sniff for them.

I’ve picked two of the most widely applicable smells to discuss here:

“Don’t Repeat Yourself”

“Comments”

Both of these commonly apply but are often not well understood.

Copyright 2008 The MathWorks, Inc.

Don’t Repeat Yourself

DRY

Duplicated code is “number one in the stink parade.” [Fowler, page 76] This is such a common theme in the software literature that “don’t repeat yourself” has its own acronym (DRY).

Why? Because anything repeated in two or more places will eventually be wrong in at least one.

Meaningful differences can’t be easily distinguished from accidental ones.

Copyright 2008 The MathWorks, Inc.

Comments Are Good, Right?

```
% I put this comment here because I was taught to comment
% my code.


string_args = {'nearest neighbor', 'linear', 'spline', ...
               'pchip', 'cubic', 'v5cubic', 'ram-lak', ...
               'shepp-logan','cosine','hamming', 'hann', ...
               'none'};
```

This bad code smell is certainly counterintuitive, since we were all taught to comment our code.

Fowler: “Don’t worry, we aren’t saying that people shouldn’t write comments. In our olfactory analogy, comments aren’t a bad smell; indeed they are a sweet smell. The reason we mention comments here is that comments often are used as a deodorant. It’s surprising how often you look at thickly commented code and notice that the comments are there because the code is bad.” [Fowler, page 87]

McConnell quotes Kernighan and Plauger: “Don’t document bad code – rewrite it.” [McConnell, page 568]

Copyright 2008 The MathWorks, Inc.

Comments Are Good, Right?

```
% The interpolation options must be first in this list. If
% the number of interpolation options changes, you have to
% change the string option parsing code below.

string_args = {'nearest neighbor', 'linear', 'spline', ...
               'pchip', 'cubic', 'v5cubic', 'ram-lak', ...
               'shepp-logan','cosine','hamming', 'hann', ...
               'none'};
```

I wrote the comment above when revising the Image Processing Toolbox™ function `iradon.m`. It’s a classic example of using a comment as a deodorant to cover up the fact that the code is bad.

Copyright 2008 The MathWorks, Inc.

Comments Are Good, Right?

```
interp_strings = {'nearest neighbor', 'linear', 'spline', ...
                  'pchip', 'cubic', 'v5cubic'};
filter_strings = {'ram-lak','shepp-logan','cosine',...
                  'hamming', 'hann', 'none'};
string_args = [interp_strings filter_strings];
```

Now the code communicates the programmer's intent. The comment is no longer necessary.

Apply the "Magic Metric" — McCabe Complexity

We move now from the qualitative to the quantitative. The *McCabe complexity* (also called *cyclomatic complexity*) metric assigns a positive integer to each routine (function or method) in a program.

I call it the magic metric because it is:

- Easy to compute
- Easy to understand
- Independent of programming language
- Widely accepted
- Well correlated with program quality

From the Carnegie Mellon Software Engineering Institute (SEI): "Cyclomatic complexity is the most widely used member of a class of static software metrics. Cyclomatic complexity may be considered a broad measure of *soundness* and *confidence* for a program. Introduced by Thomas McCabe in 1976, it measures the number of linearly-independent paths through a program module."

www.sei.cmu.edu/str/descriptions/cyclomatic_body.html

Measure Complexity

```
idxGroupedByLevel = {};
done      = false;
findHole = false; % start with an object boundary
while ~done
  if (findHole)
    I = FindOutermostBoundaries(holes);
    holes = holes(~I); % remove processed boundaries
    idxGroupedByLevel = [ idxGroupedByLevel, {holeIdx(I)} ];
    holeIdx = holeIdx(~I);    % remove indices of processed boundaries
  else
    I = FindOutermostBoundaries(objs);
    objs = objs(~I);
    idxGroupedByLevel = [ idxGroupedByLevel, {objIdx(I)} ];
    objIdx = objIdx(~I);
  end
  if (processHoles)
    findHole = ~findHole;
  end
  if ( isempty(holes) && isempty(objs) )
    done = true;
  end
end
```

Computing cyclomatic complexity for a routine:

“Add one for the straight path through the routine.

Add one for each of the following keywords, or their equivalents: `if` `while` `repeat` `for` `and` `or`

Add one for each case in a [switch] statement.” [McConnell, page 458]

In MATLAB, don’t forget to count the `elseif`.

Cyclomatic complexity is easy to compute, but doing it manually gets old very fast. Fortunately, you can find tools that compute it for you.

In MATLAB –

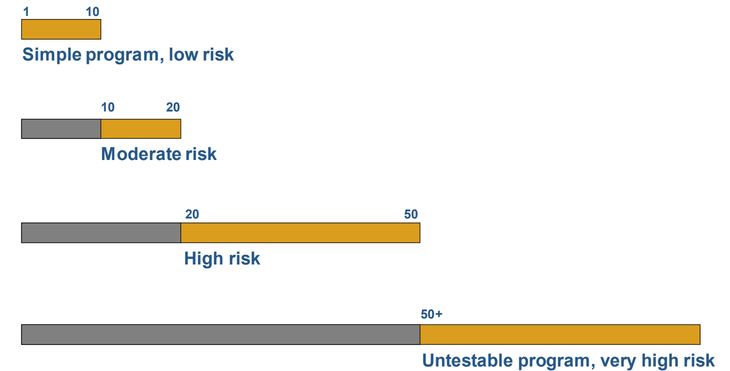
```
mlint -cyc foo
```

For other languages, some teams at The MathWorks use Source Monitor, a free tool that can compute software metrics for C++, C, C#, Java, Delphi, Visual Basic (VB6) or HTML.

www.campwoodsw.com/sourcemonitor.html

Copyright 2008 The MathWorks, Inc.

Complexity Correlates with Bug Risk



Source: Software Engineering Institute (SEI)

www.sei.cmu.edu/str/descriptions/cyclomatic.html

Studies:

- McCabe, Tom. 1976. “A Complexity Measure.” *IEEE Transactions on Software Engineering*, SE-2, no. 4 (December): 308-20.
- Shen, Vincent Y., et al. 1985. “Identifying Error-Prone Software – An Empirical Study.” *IEEE Transactions on Software Engineering*, SE-11, no. 4 (April): 317-24.
- Ward, William T. 1989. “Software Defect Prevention Using McCabe’s Complexity Metric.” *Hewlett-Packard Journal*. April, 64-68.

The Ward study reported significantly higher program reliability produced by using the complexity metric at Hewlett-Packard.

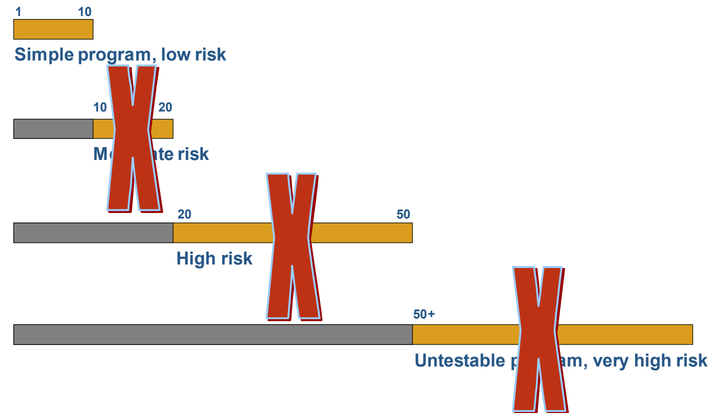
At The MathWorks, we have ample anecdotal evidence that high complexity numbers are correlated with problematic code, and that reducing complexity numbers results in code that’s much easier to understand and maintain.

On the Image and Geospatial Team at The MathWorks, our presubmission code checklist includes this:

> “Have you checked the McCabe complexity metric? If you're modifying existing code, try to keep the metric from going up for routines you touch. If you're creating new code, aim for the target of a complexity beneath 10 for each new routine.”

Copyright 2008 The MathWorks, Inc.

Set Coding Standards Based on Complexity



What coding rules should you follow on your project? If you had to pick just one, make it be this:

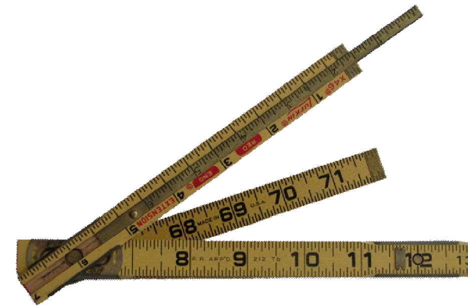
No routines allowed with a cyclomatic complexity higher than 10.

Do you supervise people who write code? Insist that they follow this rule.

Why so much focus on the ordinary routine? It is the fundamental unit of code organization in almost every widely-used programming language. Improve your routines, and you improve all of your code.

Copyright 2008 The MathWorks, Inc.

Measuring Complexity Automatically



Cyclomatic complexity is easy to compute, but doing it manually gets old very fast. Fortunately, you can find tools that compute it for you.

The MATLAB `mlint` function has an option for reporting the cyclomatic complexity of functions within an M-file:

```
mlint -cyc foo
```

(This option was documented only recently, but it has been available since the R2006a release.)

For other languages, some teams at The MathWorks use Source Monitor, a free tool that can compute software metrics for C++, C, C#, Java, Delphi, Visual Basic (VB6) or HTML.

www.campwoodsw.com/sourcemonitor.html

Copyright 2008 The MathWorks, Inc.

Refactoring Your Code

Every change to source code strongly tends to increase code badness

Steve's 2nd law of software thermodynamics:

Every change to source code strongly tends to increase code badness.

(I'm sorry, I don't have a 1st law.)

Most programmers come to realize, sooner or later, that preserving clean design and maintainable code requires constant vigilance.

Programmers also learn, though, that it's dangerous to modify working code, because it's all too easy to introduce new bugs.

So what to do? Use *refactoring* – a set of programming practices designed to allow a programmer to improve the design and structure of existing code regularly and safely.

Resist the temptation to be lazy in terminology – "refactoring" means a specific set of programming practices; it isn't just any ol' big code rewrite.

Copyright 2008 The MathWorks, Inc.

Improving the Design of Existing Code

What is refactoring? I'll let refactoring guru Martin Fowler explain it:

> "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written." [Fowler, page xvi]

Key refactoring concepts:

- Have good tests in place first
- Make only changes that do not alter external behavior
- Make only one refactoring change at a time
- Run tests after each change

Copyright 2008 The MathWorks, Inc.

Refactor Your Code

$$\int u\,dv \Rightarrow uv - \int v\,du$$

Do you remember integration by parts in calculus? We were taught that we could *transform* the expression on the left into the expression on the right without changing its meaning.

To *refactor* code is to *transform* it, to alter its structure, **without changing its behavior**.

Similar to the rule of integration by parts, there are several useful rules for transforming code without changing behavior.

For a comprehensive catalog of refactoring rules, see Martin Fowler's book *Refactoring: Improving the Design of Existing Code*.

Copyright 2008 The MathWorks, Inc.

Extract a Method

```
% Set the mouse pointer to be the Window/Level custom
% pointer.  The custom cdata shape is stored in the file
% cursor_contrast.png in the IPT icon directory.
iconfile = fullfile(ipticondir, 'cursor_contrast.png');
cdata = makeToolbarIconFromPNG(iconfile);
% We just need the first plane; offset it by 1
cdata = cdata(:,:,1) + 1;
set(fig, 'Pointer', 'custom', 'PointerShapeCData', cdata);
```

```
set(fig, 'Pointer', 'custom', 'PointerShapeCData', ...
    getWindowLevelPointer);
```

Extract a method is one of the most common refactorings.

Fowler's notes for extract a method:

- **Summary:** "You have a code fragment that can be grouped together. Turn the fragment into a method whose name explains the purpose of the method."
- **Motivation:** When a method is too long; when a code fragment needs a comment to explain its purpose
- **Issues:** Watch for problems with local and temporary variables.

Copyright 2008 The MathWorks, Inc.

Put It All Together

Version Control

Complexity Metric

Code Smells

Refactoring

Case studies:

scantiff.m – Refactoring my own code to keep function complexity under control

patchm.m – Refactoring someone else's code so that I can figure out how to change it safely.

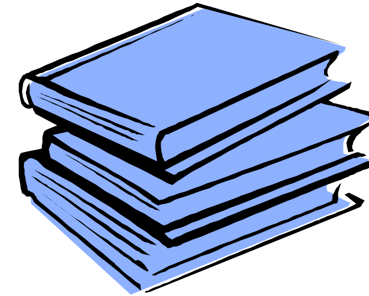
Key refactoring tips:

- Make changes in very small, quick increments.
- Make sure the code still works after each change.
- Check in each small change to version control.
- Drive down function complexity.
- No behavior changes allowed!

General tip for handling "bad" code that someone else left in your lap: If you can't understand the code well enough to make the changes you need to make, start by refactoring the code, using small, cautious steps. Submit each incremental change to version control so that you can easily revert if things start to go wrong.

Copyright 2008 The MathWorks, Inc.

Recommended Reading



There are many excellent books about the craft of software development. I've chosen a very small number to recommend here. These books

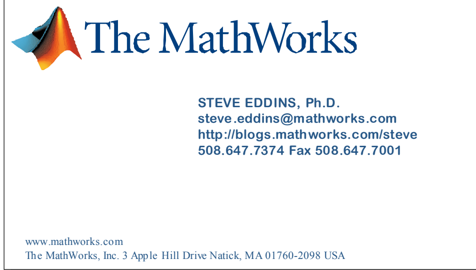
- Are language- and environment-agnostic
- Can be read in part or in whole
- Have good bibliographies and recommended reading lists if you want to learn more

Also, these are the books I relied upon most heavily to prepare this presentation. All three are widely read and studied at The MathWorks.

- Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*, 2nd edition. Microsoft Press. 2004.
- Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley. 2000.
- Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. 1999.

Also, take a look at the course *Software Carpentry* by Greg Wilson. Course materials are available online at http://www.swc.scipy.org/

Copyright 2008 The MathWorks, Inc.



Copyright 2008 The MathWorks, Inc.