

# Taking Control of Your Code: Essential Software Development Tools for Engineers

**Steve Eddins, Ph.D.**

**October 9, 2006**

**International Conference on Image Processing**

© 2006 The MathWorks, Inc.



## Modern Engineering Work Requires Software



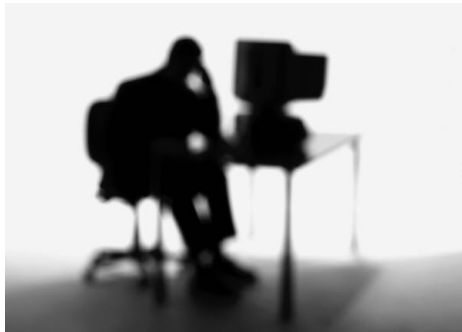
Engineers in all disciplines rely heavily on software.

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.

## Engineers Write Software — or Manage People Who Do



You write software for yourself.

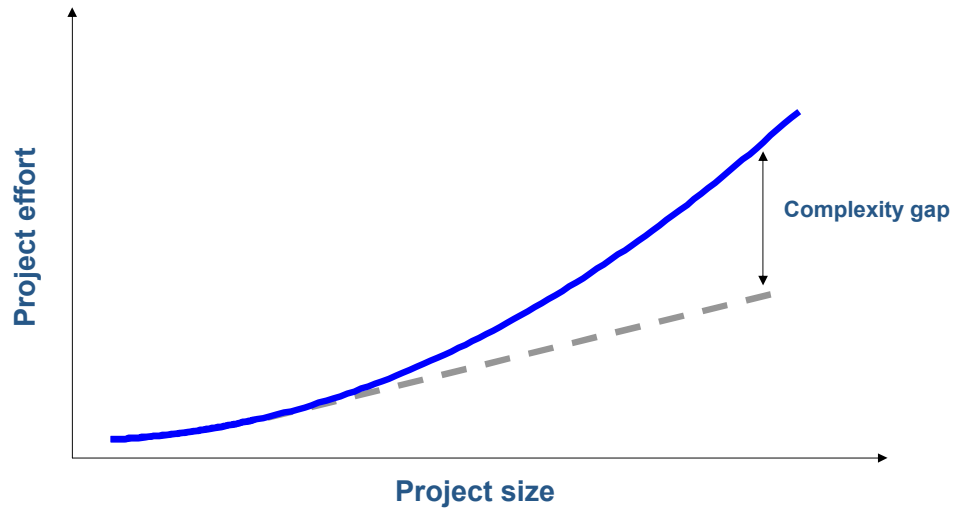
Or...

You write software that other people use.

Or...

You supervise people who write software.

## Everyone Struggles with Complex Software Projects



The engineer's dilemma: You're not a full-time programmer, and you don't especially want to become one. But when the software projects stumble, your engineering productivity suffers.

Typical problems:

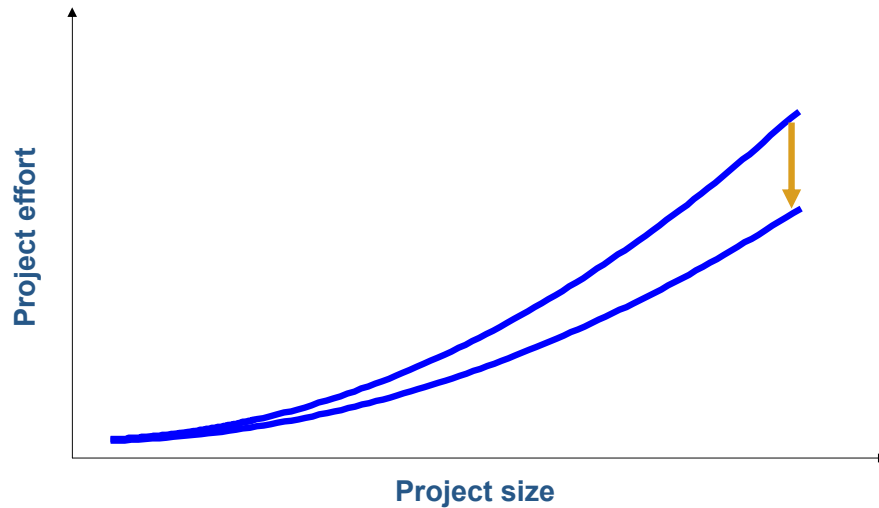
- Last month's results can't be reproduced because the software changed.
- Collaborators use different versions of the software, resulting in confusion.
- Software is "brittle." That is, it is difficult to modify without breaking it.

Why do software problems consistently surprise us?

Answer: The *complexity gap*



## Improving Your Approach to Software Development



The good news: It is possible to get better, incrementally, at this software development business.

And it doesn't require fancy new programming languages or methodologies.

By adopting basic tools and methods, you can increase the quality and reliability of your code, and you can reduce the overall development effort needed.

## Learning and Applying Basic Software 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?

In software development, there are tools and techniques not mentioned in school that software professionals know and rely upon. We'll talk about some of these tools and techniques in this presentation:

- Version control systems
- Unit testing
- Bad code smells
- Code quality metrics
- Refactoring

## Using a Version Control System

### Pragmatic Programmer Tip:

### Always Use 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.

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]

Get a version control system, learn to use it, and use it every day you work with code.

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

*Commit* – use the version control system to put a set of file modifications into the repository

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

## Version Control — Your Project's Undo Button



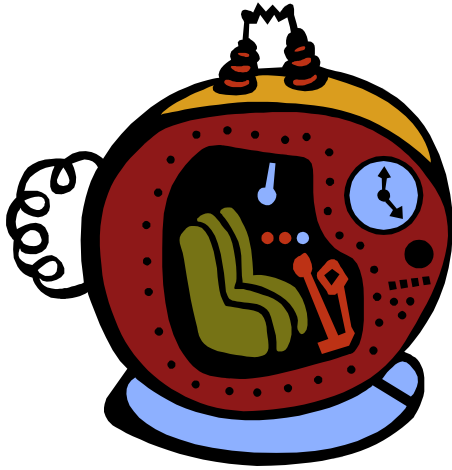
Have you ever started making a few “simple” changes, spent a day or so working on them, and then changed your mind? Maybe you realized the changes were misguided, or that they were more complicated than you expected.

Or – have you ever been caught in the middle of modifying your code by a request to do a demonstration? And you can’t, because the modifications aren’t complete and working yet.

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

[Demo – interacting with a version control system – revert to repository version; change logs; diffs]

## Version Control — Your Project's Time Machine

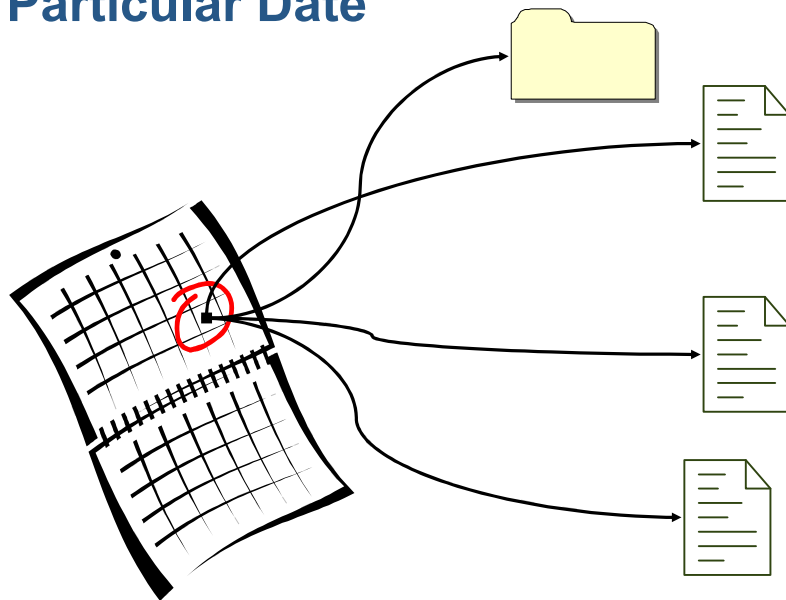


Can you reproduce results from a paper you wrote two years ago?

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.

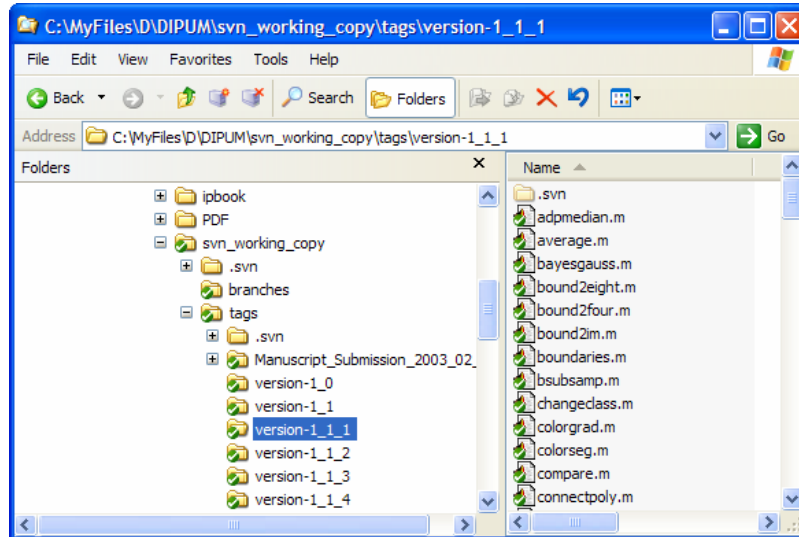
When you get in the habit of using version control, you'll be able to reproduce past results at will.

## Reconstructing Your Files from a Particular Date



[Demo – reconstructing a file based on a date]

## Reconstructing a Particular Release



Do you write code for others to use? For example, have you written software tools used by everyone in your lab? Then start thinking in terms of “releases.” Whenever you distribute updated files, you are distributing a new release, and you should give it a name or number.

When someone reports a problem with your tools, your first question should be “What release of the tools are you using?” Then reconstruct your own working copy of that release, so you can reproduce and then diagnose the problem.

[Demo – “tagging” a set of files as a release]

## Version Control Facilitates Collaboration



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

I would refuse to work on any sort of collaborative software development project that doesn’t use version control.

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

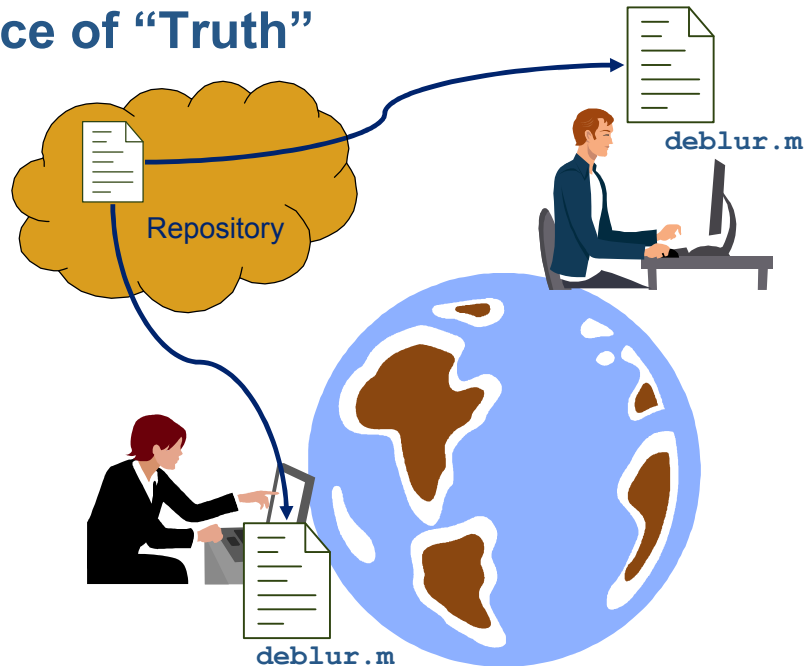
Version control shows you at a glance whether anyone else on the team has changed any of the files you have.

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.



## Version Control Repository—Your Single Source of “Truth”

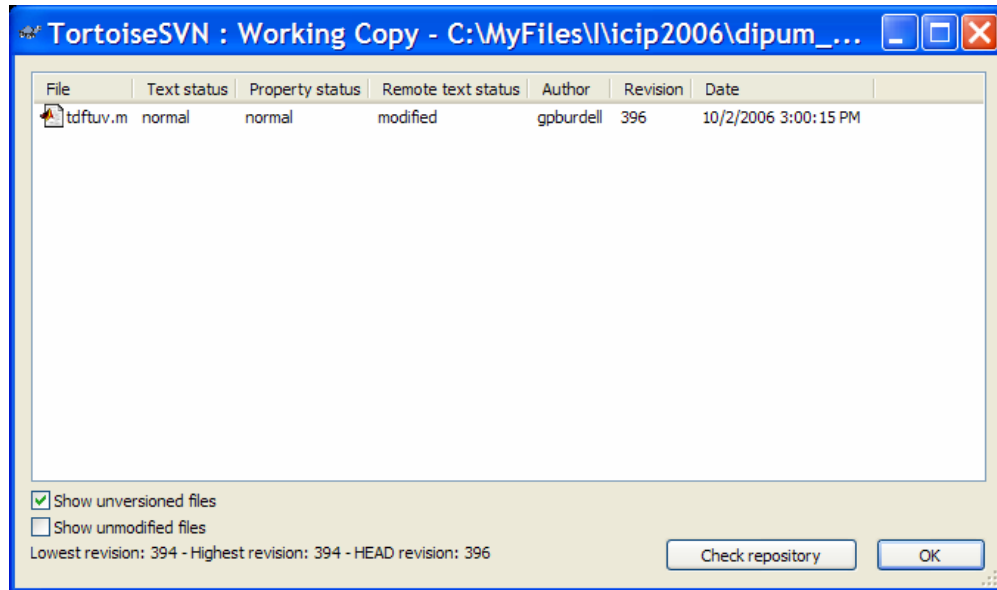


Two collaborators are working on the same set of files. Whose copy of `deblur.m` is the latest? How do you keep track?

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.

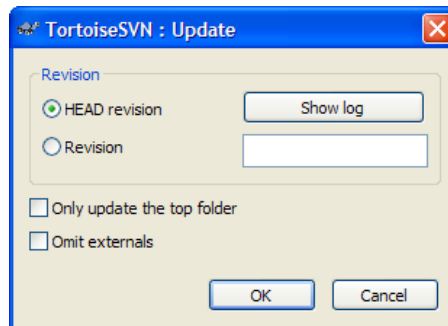
## Determining If Your Files Have Been Changed



Version control lets you see at a glance if anyone else has changed the files you are working on.

[Demo]

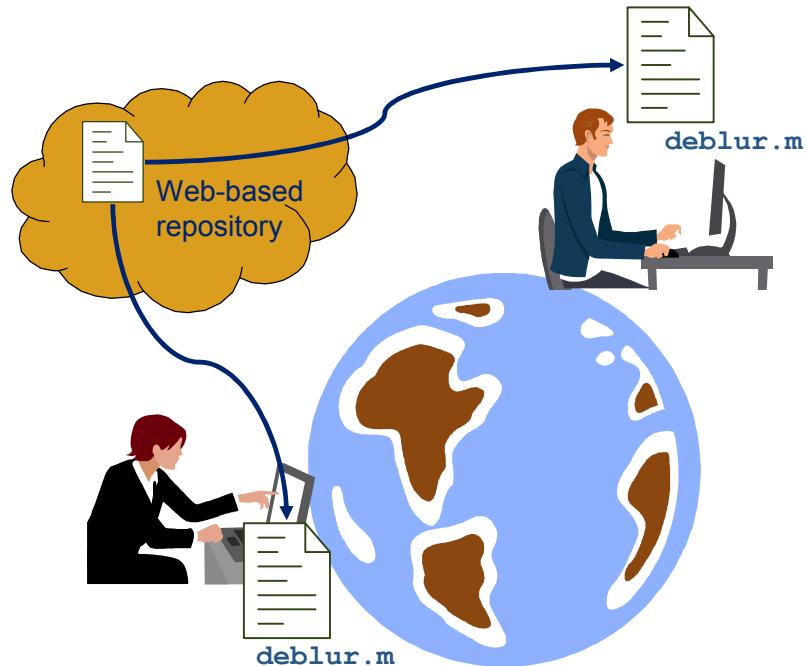
## Getting the Latest Changes



Version control lets you quickly update your working copy with the latest changes.

[Demo]

## Using a Web-Based Repository



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.

## Choosing 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.

## Use Subversion



Use Subversion for your version control system.

Subversion is a modern, freely available, 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.

Obtain Subversion from:

[subversion.tigris.org/](http://subversion.tigris.org/)

Other version control system choices:

- CVS
- Commercial offerings such as Perforce

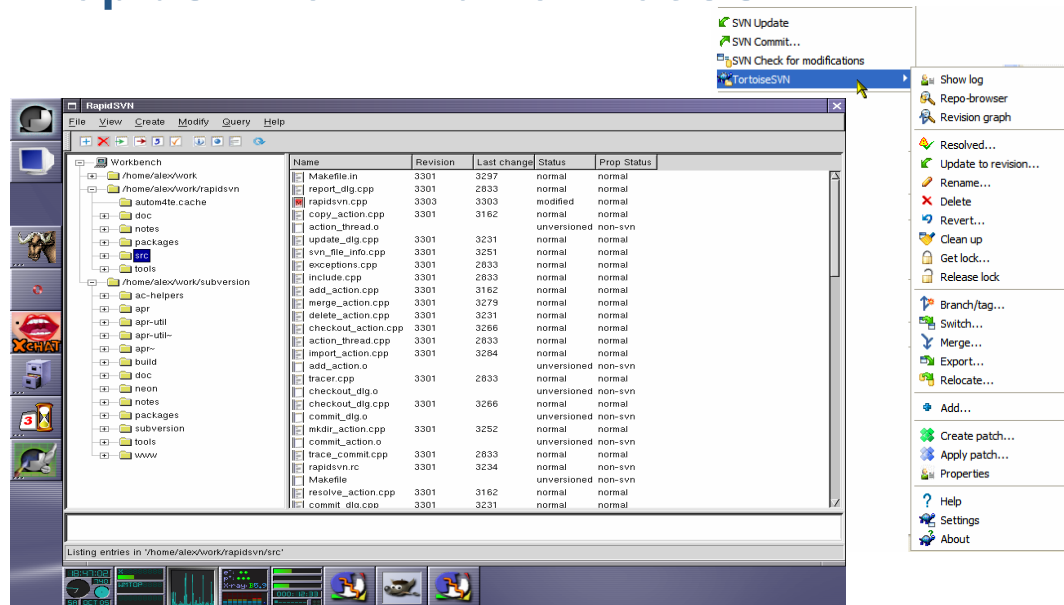
There are many others. For a comprehensive list, see:

[en.wikipedia.org/wiki/List\\_of\\_revision\\_control\\_software](http://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](http://en.wikipedia.org/wiki/Comparison_of_revision_control_software)

# 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.

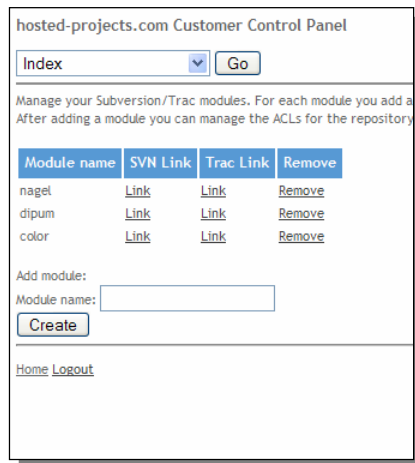
Easy-to-use GUI tools are available on all major platforms.

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/](http://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/](http://rapidsvn.tigris.org/)

For Web-based repository browsing, try WebSVN, available from: [websvn.tigris.org/](http://websvn.tigris.org/)

## Use Commercial Web-Based Repository Providers



hosted-projects.com Customer Control Panel

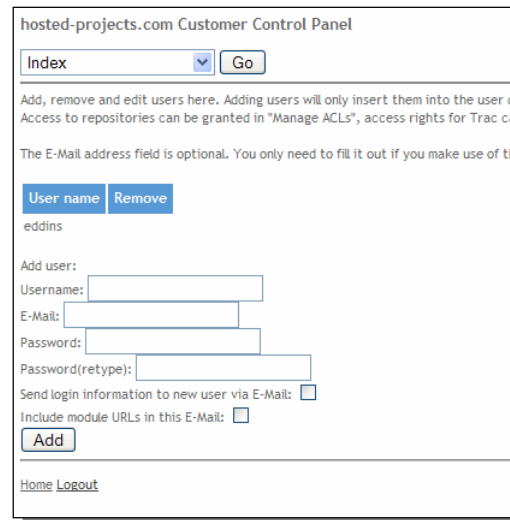
Index

Manage your Subversion/Trac modules. For each module you add a  
After adding a module you can manage the ACLs for the repository.

Module name	SVN Link	Trac Link	Remove
nagel	<a href="#">Link</a>	<a href="#">Link</a>	<a href="#">Remove</a>
dipum	<a href="#">Link</a>	<a href="#">Link</a>	<a href="#">Remove</a>
color	<a href="#">Link</a>	<a href="#">Link</a>	<a href="#">Remove</a>

Add module:  
Module name:

[Home](#) [Logout](#)



hosted-projects.com Customer Control Panel

Index

Add, remove and edit users here. Adding users will only insert them into the user database. Access to repositories can be granted in "Manage ACLs", access rights for Trac can be granted in "Manage Trac ACLs".  
The E-Mail address field is optional. You only need to fill it out if you make use of the "Send login information to new user via E-Mail" checkbox.

User name	Remove
eddis	<a href="#">Remove</a>

Add user:  
Username:   
E-Mail:   
Password:   
Password(retype):   
Send login information to new user via E-Mail: ☐  
Include module URLs in this E-Mail: ☐

[Home](#) [Logout](#)

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. I prefer to use an inexpensive commercial service, so that I don't feel bad asking for support if I need it.

[Demo – if Internet access is available – commercial Subversion hosting provider; setting up repositories; browsing repositories; controlling access]



## Improving 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 in practice, and don't require specialized training to learn and apply:

- Unit testing
- Bad code smells
- Code quality metric
- Refactoring

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.

## Write Unit Tests — Run Them Often

```
function test_results = tpBasic(test_results)
[U,V] = dftuv(4, 5);
act1 = U;
exp1 = [ 0  0  0  0  0
         1  1  1  1  1
         2  2  2  2  2
        -1 -1 -1 -1 -1];

act2 = V;
exp2 = [0  1  2 -2 -1
        0  1  2 -2 -1
        0  1  2 -2 -1
        0  1  2 -2 -1];

test_results = CheckTestPoint(test_results, ...
                              {act1, exp1}, ...
                              {act2, exp2});
```

There are many types of test types, such as unit, component, integration, regression, and system tests. [McConnell, page 500]

A *unit test* verifies the execution of a single routine, class, or small program. In professional software organizations, unit tests are often written by software developers instead of quality engineers.

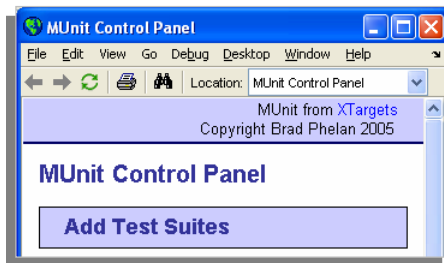
To encourage frequent use of unit tests, they should be easy to run, easy to write, and execute in just a few seconds. If everything is OK, they should be silent, or almost silent.

Remember that writing a unit test is a form of communication – you are recording in a very explicit fashion exactly how you expect your routine to be called, and exactly what you expect your routine to do.

Experienced software developers know that time saved by skipping test writing is more than lost later by time spent debugging. Many (most?) developers learn this the hard way. I did.

[Examples – Image Processing Toolbox tests; *Digital Image Processing Using MATLAB* tests]

## Beg, Borrow, or Steal a Test Harness



```
>> RunAllTests .
Test: tbound2eight      passed
Test: tchapter10        passed
Test: tchapter11        passed
Test: tchapter12        passed
Test: tchapter2         passed
Test: tchapter3         passed
```

A test harness makes it easy for you to add and run tests. Do use a test harness, but don't create one if you don't need to. See this page for a list of at least 40 test harnesses for different languages and environments, including MATLAB:

[en.wikipedia.org/wiki/List\\_of\\_unit\\_testing\\_frameworks](http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks)

Or you can search for "unit test" on the MATLAB Central File Exchange to find even more test harnesses for MATLAB:

[www.mathworks.com/matlabcentral/fileexchange](http://www.mathworks.com/matlabcentral/fileexchange)

[Example – Test harness for *Digital Image Processing Using MATLAB*]

## Exercise all Lines, all Branches

```
297 % for empty theta, choose an intelligent default delta-theta
30 298 if isempty(theta)
299     theta = pi / size(p,2);
300 end
301
302 % If the user passed in delta-theta, build the vector of theta values
30 303 if numel(theta)==1
304     theta = (0:(size(p,2)-1))* theta;
305 end
```

What makes for a good unit test?

First, I have to emphasize that just having a unit test, any unit test, is so much better than having none at all!

If you are aiming higher, though, the minimal criterion is that the test should exercise every line of code at least once. This is called the “code coverage” criterion.

The next step is to achieve complete “branch coverage.” That is, every predicate term is tested for at least one true and one false value.

As McConnell says, because “exhaustive testing is impossible, practically speaking, the art of testing is that of picking the test cases most likely to find errors.” [McConnell, page 505] There is a lot of existing science and experience behind this “art.” To learn more, see section 22.3, “Bag of Testing Tricks,” in [McConnell].

## Limitations of Testing



“Testing by itself does not improve software quality. Test results are an indicator of quality, but in and of themselves they don’t improve it. Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often. ... If you want to improve your software, don’t just test more; develop better.” [McConnell, page 501]

## 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.

Duplicated code	Long method	Large class
Long parameter list	Divergent change	Shotgun surgery
Feature envy	Data clumps	Primitive obsession
Switch statements	Parallel inheritance hierarchies	Lazy class
Speculative generality	Temporary field	Message chains
Middle man	Inappropriate intimacy	Alternative class with different interfaces
Incomplete library class	Data class	Refused bequest
Comments		

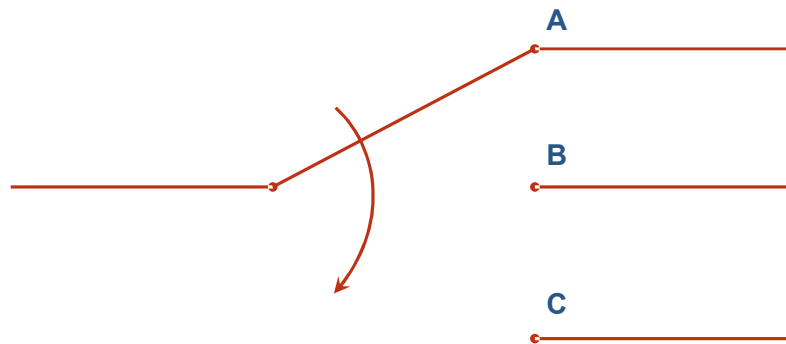
## Don't Repeat Yourself

# 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.

## Suspect Switch Statements



Switch statements aren't bad on their own; it's just that they have a strong tendency to get copied from place to place.

In an object-oriented language, replace a switch statement with some form of polymorphism. In a procedural language, extract the switch statements into a single routine that hides the details.

[Example: `imread`, `imwrite`, and `imfinfo` in MATLAB 5]



## 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]

## 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 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.

## 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” — Cyclomatic Complexity

**For each function (method, routine, procedure):**  
**The number of decision points plus 1**

We move now from the qualitative to the quantitative. The *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](http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html)


Cyclomatic complexity is often called *McCabe complexity*.

## Cyclomatic Complexity — Simple to Use, Understand

```

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:

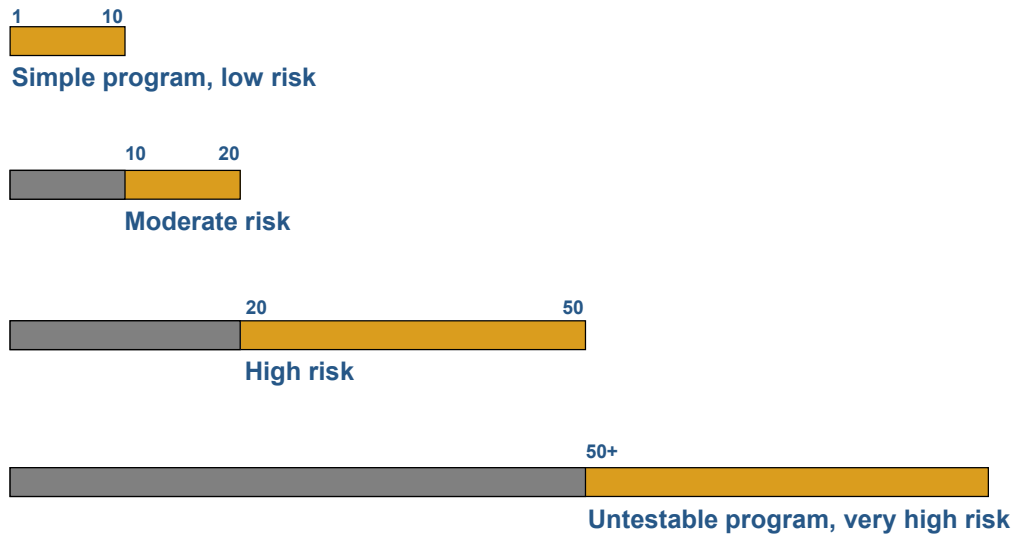
“Add one for the straight path through the routine.

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 — Correlated with Bug Risk



Source: Software Engineering Institute (SEI)

[www.sei.cmu.edu/str/descriptions/cyclomatic.html](http://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.

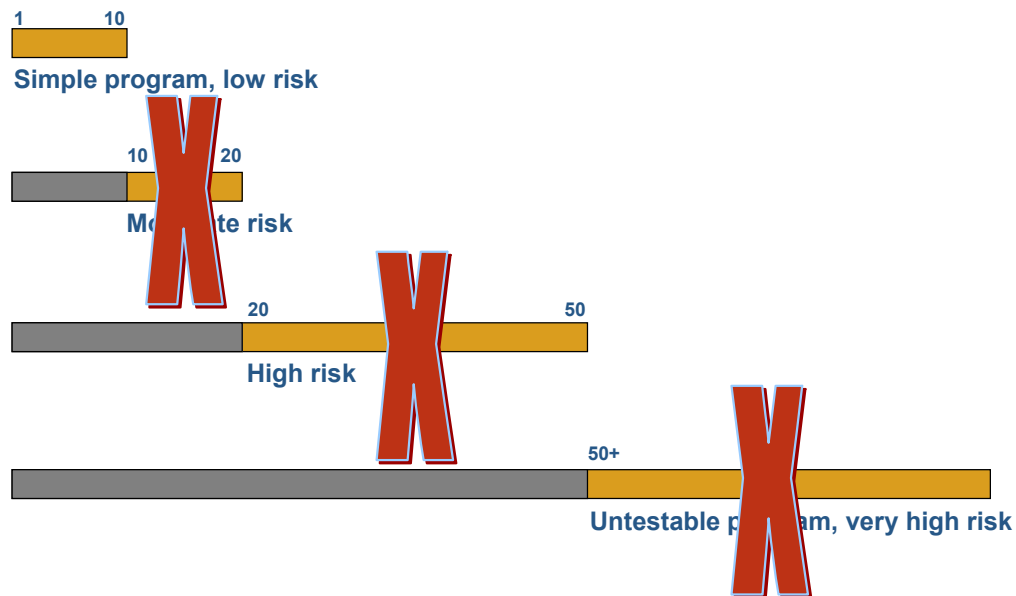
The Ward study reported significantly higher program reliability produced by using the complexity metric at Hewlett-Packard.

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."

## 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.

## 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.

In recent releases of MATLAB (R2006a and R2006b), the `mlint` function has an undocumented option for reporting the cyclomatic complexity of functions within an M-file:

```
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](http://www.campwoodsw.com/sourcemonitor.html)



## Refactoring Your Code

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

Steve's 2<sup>nd</sup> law of software thermodynamics:

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

(I'm sorry, I don't have a 1<sup>st</sup> law.)

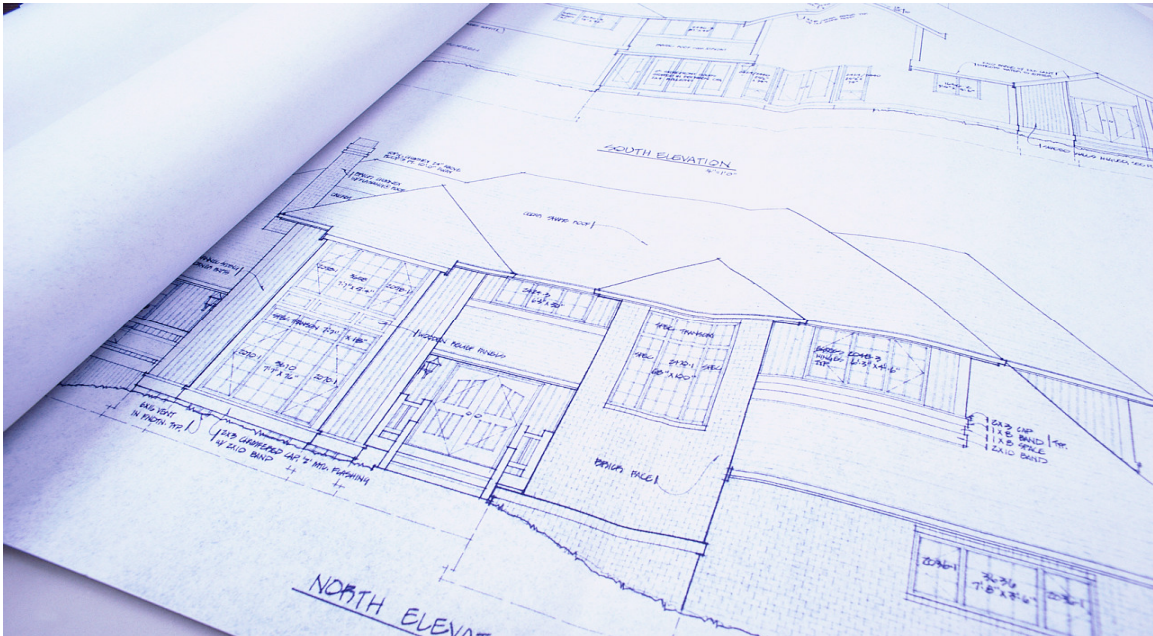
Most programmers come to realize, sooner or later, that preserving clean design and maintainable code requires constant vigilance.

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.

## 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

## Check Everything In — Write Tests



If you have no tests, and if the code isn't checked into a version control system, then STOP!

You may proceed no further with refactoring until you have good tests and everything is checked in.

## Extracting 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);
```

Fowler's refactoring catalog includes several common elements for each item:

- A brief 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
- Procedure: Detailed, mechanical steps for transforming existing code into the new form. In extract method, watch for problems with local and temporary variables. Use other refactorings to solve these problems if necessary. The procedures are cautious at every step. For example, Fowler has you make sure the extracted method compiles successfully before modifying the old code to call the new method.

## Using a Guard Clause

```
function out = watershed(in)
if ~isempty(in)
    if ~any(isnan(in(:)))
        watershed alg code
        ...
    else
        error('NaN input')
    end
else
    out = [];
end
```

```
function out = watershed(in)
if isempty(in)
    out = [];
    return;
end

if any(isnan(in(:)))
    error('NaN input')
end

watershed alg code
...
...
```

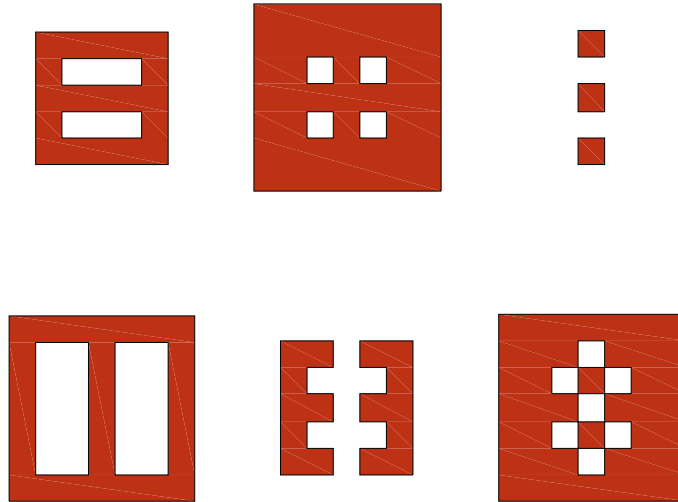
“A method has conditional behavior that does not make clear the normal path of execution. *Use guard clauses for all the special cases.*” [Fowler, page 250]

Many people have learned a “rule” that functions should have only one exit point, which should be at the end. This rule originates from the early days of structured programming. There’s a good reason for the rule: Early exits can lead to errors if the programmer isn’t careful with them.

But remember, the more fundamental rules are to communicate with other people, and to reduce complexity. Sometimes, sticking to no-early-returns leads to awkward code that goes against the spirit of the fundamental routines.

Using a guard clause with an early exit can sometimes clarify the principal purpose of a routine.

## Mapping Toolbox Case Study



Here's a case study from my own experience working on a function in Mapping Toolbox. It pulls together version control, unit testing, cyclomatic complexity, and refactoring.

## Struggling with Software Development



You're an engineer. You rely on software to do your job. Maybe you write that software, or maybe you supervise someone else on your team who does.

Recurring problems with the software frustrate you and make you less effective. But you really don't want programming to take over as your main job or concern.

What do you do?

## Learning and Applying Basic Software Tools



There are simple tools and techniques considered essential by all professional software developers. They have nothing to do with the latest fashions in object-oriented this-or-that language. They aren't taught in the engineering programming course (or maybe not in any course).

But they're not that complicated, and you can learn them. Learn one technique, and it'll help. Learn several, and regularly apply them to your work, and you may soon feel like you are controlling your software project instead of the other way 'round.



# Taking Control of Your Code

**Version Control**

**Unit Tests**

**Quality Metrics**

**Code Smells**

**Refactoring**

## 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*, 2<sup>nd</sup> 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.

You might also want to look at *Software Carpentry*, the open-source lecture materials for an extensive course covering these topics and more:

[www.swc.scipy.org/](http://www.swc.scipy.org/)



STEVE EDDINS, Ph.D.  
steve.eddins@mathworks.com  
<http://blogs.mathworks.com/steve>  
508.647.7374 Fax 508.647.7001

[www.mathworks.com](http://www.mathworks.com)  
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098 USA