



SOFTWARE CARPENTRY

Getting Scientists to Write Better Code by Making Them More Productive

By Greg Wilson

IN 2005, I TAUGHT A COURSE ON BASIC SOFTWARE DEVELOPMENT SKILLS TO 37 GRADUATE STUDENTS AT THE UNIVERSITY OF TORONTO. FOURTEEN WERE STUDYING COMPUTER SCIENCE; THE REST WERE IN PHYSICS, THE LIFE SCIENCES, mechanical and civil engineering, and other disciplines.

Only four of the students used a version-control system when the course started, and only two of those were from computer science. Only one (a physicist working on a multinational collaboration) tested his software as a matter of routine; none used any kind of code-checking tools.

I wish this were unusual, but it's not. Since 1997, I've taught software engineering to scientists and engineers across the US and Canada. Most of my students have been intelligent, hard-working people with advanced degrees, but only a handful have had any formal training in programming beyond a general freshman course in Java or C and a later course in numerical methods or bioinformatics tools. As a result, for every high-energy physicist whose test suite runs automatically every time she checks changes into her Subversion repository, dozens of other scientists are debugging with print statements and backing up their work in directories called something like `working_sep9_01d`.

Yet, it doesn't have to be like this. Over the past nine years, my colleagues and I have developed a one-semester course that teaches scientists and engineers the "common core" of modern

software development. Our experience shows that an investment of 150 hours—25 of lectures and the rest of practical work—can improve productivity by roughly 20 percent. That's one day a week, one less semester in a master's degree, or one less year for a typical PhD.

The course is called *software carpentry*, rather than software engineering, to emphasize the fact that it focuses on small-scale and immediately practical issues. All of the material is freely available under an open-source license at www.swc.scipy.org and can be used both for self-study and in the classroom. This article describes what the course contains, and why.

Quality Is Free

The software carpentry course aims to teach computational scientists how to meet the standards that experimental scientists have taken for granted for almost 200 years. If experimentalists don't calibrate their equipment, check their reagents' purity, and take careful notes, what they're doing isn't considered science. In contrast, computationalists don't even learn how to assess their software's quality in any systematic way, and very few would be able to recreate and rerun the programs they

used to produce last year's papers. As a result, most computational science is irreproducible and of unknown quality.

The reason most computational scientists don't care about quality is that there's no incentive for them to do so: journal or conference reviewers rarely ask how they tested the code used to produce the results in a paper, and no one gets points toward tenure for shaking the last few bugs out of a simulation.

The course therefore doesn't try to "sell" quality directly. Instead, it starts from the fact that the only way to improve productivity is to improve quality. As in manufacturing and medicine, investments in quality repay themselves several times over because mistakes are more expensive to fix than to prevent.

This realization is at the heart of all modern software development methodologies. At one end of the spectrum, design-heavy approaches such as the Rational Unified Process (RUP) try to prevent bugs entirely. At the other, agile methodologies such as extreme programming (XP) tighten feedback loops to squash bugs before they can do any harm.

Fewer differences exist in practice between these approaches than in theory (or in textbooks). Most professional software development teams use the same basic tools in the same way, regardless of what ideology they officially espouse. Those tools and practices are the core of our course.

Version Control

Version control is the one thing that every project must have; not using it is

the moral and practical equivalent of not keeping notes while doing a physical experiment. My favorite version-control system is Perforce (www.perforce.com), but Subversion (<http://subversion.tigris.org>) is a very good open-source alternative.

We initially tried to sell version control on the basis of its ability to undo mistakes and facilitate teamwork. We found, however, that it was better to introduce version control as a way to synchronize files between home, work, the lab, a laptop, and the machine you borrowed to give your talk after Air Canada lost your luggage (again). Even a brief exposure can convince students that it's more reliable than copying files onto a USB stick or sending themselves email.

Once students have mastered the basic check-out/edit/check-in cycle, they're ready to start working in pairs or small teams. We cover branching and merging at the end of the course, if at all, because they can go wrong in too many ways, and students rarely understand the need for these operations until they've worked on large projects.

Repeatable Build

Once students know how to manage their source code, the next step is to show them how to build their programs. We do this with GNU Make (www.gnu.org/software/make/), rather than a more modern tool such as Ant (<http://ant.apache.org>), for three reasons:

- GNU Make is well-documented and runs everywhere.
- C/C++ and Fortran programmers still outnumber Java programmers in the course.
- Most of our students have never worked with XML, which Ant uses for its configuration files.

GNU Make does have problems, the

biggest of which is its lack of support for debugging—if anything goes wrong, the only way to fix it is to tweak and pray. However, Ant is just as bad, if not worse, and its support for languages such as Fortran is still shaky.

Make's second problem is its reliance on the nearly orphaned skill of shell programming. Programmers can now use scripting languages such as Python to tackle most of the tasks they used the shell for 10 years ago. Students might not mind learning enough about shell programming, but as an instructor, I resent the hours it takes to teach them a

know how to choose sensible variable names, modularize their programs, or use any data structure more sophisticated than a rectangular array.

The course therefore spends a few lectures teaching them the basics of Python (www.python.org), a modern open-source scripting language with good support for numerical programming, data manipulation, and visualization. To avoid giving the impression that the course is about a language, we've found that it's important to focus on the things scripting languages are good for. We therefore teach Python as

*If they thought of themselves as software engineers,
the next couple of lectures would be about testing.*

second, inferior, way to do simple tasks.

In an ideal world, the solution would be to adopt something like SCons (www.scons.org), which lets developers write build files in scripting languages. This solves the debugging and extensibility problems that afflict both Make and Ant; in fact, Ant's creator has said that if he had it to do over again, he'd have taken this route (http://weblogs.java.net/blog/duncan/archive/2003/06/ant_dotnext.html). But SCons and its kin are still young and infrequently encountered in the wild, so Make remains the lesser of many evils.

Scripting

By this point, students have their source code under control and can rebuild their programs with a single command. If they thought of themselves as software engineers, the next couple of lectures would be about testing. Experience shows, however, that it's hard to do this until they're more proficient programmers. They've all seen loops, conditionals, and functions, but most still don't

a way to reformat legacy data files, reset the laser at 3:00 a.m., and so on.

Debugging

Along the way, we spend at least one full lecture on the art of debugging. This is always a hard topic to teach because everything you can say in a lecture is either a bland platitude ("check your assumptions") or uselessly specific ("so we check that the sum of the indices is less than the length of the storage vector, and..."). The course notes do what they can, but going through a few problems live in the classroom always works best.

This is also the point at which we try to wean students off dumb text editors and get them to use an integrated development environment (IDE). They might not see the need for a class browser—they might not even be writing classes at all yet—but a few demos of how painful programming isn't when you use a symbolic debugger can be very compelling. Some holdouts might insist on pretending that it's still 1975

SUGGESTED READINGS FOR COMPUTATIONAL SCIENTISTS

One of the problems computational scientists face is finding information that is neither too shallow nor too detailed, and doesn't assume too much background knowledge. The following books are the most useful I've encountered in my years as an editor and reviewer for *Dr. Dobb's Journal*:

- Scott Berkun, *The Art of Project Management* (O'Reilly, 2005). Berkun talks about the really important parts of management—team building, trust, and communication—without any kayaking anecdotes or mention of Zen. Don't be misled by the soft-edged diagrams; this is very much a get-it-done book.
- Mike Clark, *Pragmatic Project Automation* (Pragmatic Bookshelf, 2004). This book focuses on getting your project to build itself, and (more important) tell you how the build went. Clark doesn't confine himself to running Make at 3 a.m.; he also covers ways to automatically rerun tests, build and test installers, monitor applications, and more.
- Matthew B. Doar, *Practical Development Environments* (O'Reilly, 2005). This is a guide to what should be in every team's toolbox, how competing entries stack up, and how they ought to be used. It names names, provides links, and treats free and commercial offerings on equal terms.
- Michael C. Feathers, *Working Effectively with Legacy Code* (Prentice-Hall PTR, 2005). Want to know three different ways to inject a test into a C++ class without changing the code? Want to know which classes or methods to focus testing on? Need to break interclass dependencies in Java so that you can test one module without having to configure the entire application? It's all in here, along with dozens of other useful bits of information.
- Karl Fogel, *Producing Open Source Software* (O'Reilly, 2005). This excellent guide to the open-source community's "rules of the road" describes how to earn commit privileges on a project, what to do about irreconcilable differences, how to get your project more attention, and more.
- Robert L. Glass, *Facts and Fallacies of Software Engineering* (Addison-Wesley Professional, 2002). Most of us know that maintenance consumes 40 to 80 percent of software costs, but did you know that roughly 60 percent of that is enhancements

(adding print statements to their code with vi or Emacs, for example), but an hour-long demo can convince even such dinosaurs to grow feathers and fly.

Of course, "which IDE" is always a hard question to answer. IDLE comes with Python, but it has a clunky user interface. Eclipse (www.eclipse.org) is widely used in the open-source world and has plug-in support for a wide variety of languages and tools, but its size is intimidating. This winter, we plan to experiment with Wing (www.wingware.com), a commercial IDE that offers a free-for-students version. However, we firmly believe that which IDE students see is much less important than the fact that they see one.

Testing

If all has gone well, students will now believe that it's worth learning how to test software systematically. If they've seen object-oriented programming (which is optional in Python), we use the `unittest` library that comes in the standard Python distribution; if they haven't, we use an object-free work-alike called Nose (www.somethingabout

orange.com/mrl/projects/nose/). Both are modeled on Java's JUnit library and encourage students to write lots of simple tests as they're developing their code.

But testing scientific programs is hard, and it's important that students understand why. The cause is simple: floating-point numbers don't behave the way their grade-three teacher told them numbers should. Even if they were introduced to round-off, underflow, and the like in their general freshman coursework (which is by no means guaranteed), most students don't realize that changing the order of a summation will change the sum's value.

What's worse, testing tools designed for business applications don't realize this either. The fact is that no one knows how to unit test floating-point code: simple "==" comparisons are useless, and saying, "the test passes if the absolute relative error is less than 10^{-6} " is superstition, not science. I believe that fixing this is the only Grand Challenge in supercomputing that matters.

Continuous Integration

Students now have all the pieces they

need to complete the productivity jigsaw. The final step is *continuous integration*—every time the students check in their code, the server rebuilds the application, reruns the test suite, and posts the results to the project mailing list or Web site. Many open-source tools can do this—CruiseControl (<http://cruisecontrol.sf.net>) is probably the best-documented—but having students write something simple themselves serves as a good "graduation exercise."

If your students want bonus marks, have them add coverage analysis and performance profiling to the mix, so that they know how much of their code is being tested on any given day, and whether their latest "optimization" actually slowed things down. Having them make the build results available as blogs, so that colleagues and supervisors can monitor their progress along with the latest hockey results, is a further extension that introduces students to the basics of Web programming.

Pick a Process, Any Process

The course ends with a couple of lectures on development processes that are

rather than bug fixes? Or that rewriting from scratch is more efficient if you need to modify more than 20 to 25 percent of a component? Glass backs up each of his 50-odd statements with copious references to the primary literature.

- Mike Gunderloy, *Coder to Developer* (Sybex, 2004). This book lives up to its subtitle: “Tools and Strategies for Delivering Your Software.” Project planning, source-code control, unit testing, logging, and build management are all here. I particularly recommend the chapter on working in small teams.
- Andrew Hunt and David Thomas, *The Pragmatic Programmer* (Addison-Wesley, 2000). A best-seller on Amazon since it was first published, this book covers the things that make the difference between typing in code that compiles and writing software that reliably does what it’s supposed to. Topics range from gathering requirements through design to the mechanics of coding, testing, and delivering a finished product.
- Hans Langtangen, *Python Scripting for Computational Science* (Springer-Verlag, 2004). Regular expressions, numerical arrays, persistence, the basics of GUI and Web programming, and interfacing to C, C++, and Fortran are all here, along with hundreds of short example programs. The book’s weight and

dense page layout might put off some readers, but what made me blink was that I didn’t find a single typo or error.

- Mike Mason, *Pragmatic Version Control Using Subversion* (Pragmatic Bookshelf, 2005). Mason brings together everything you’ll ever need to know about Subversion, which has become the version-control system of choice for open-source development.
- Steve McConnell, *Code Complete* (Microsoft Press, 2004). This classic handbook of do’s and don’ts for working programmers covers everything from how to avoid common mistakes in C to setting up a testing framework, organizing multiplatform builds, and coordinating teams.
- Diomidis Spinellis, *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). The author’s earlier *Code Reading* explained how to navigate large software projects; this book looks at their quality. Reliability, security, time performance, space performance, portability, maintainability, and floating-point arithmetic each get a chapter, with hundreds of examples from well-known open-source projects.
- Greg Wilson, *Data Crunching* (Pragmatic Bookshelf, 2005). My book shows how to deal with text, XML, relational databases, binary formats, and other data formats.

commonly used in medium-sized projects in industry and the open-source community. These lectures skip over material such as outsourcing, formalizing requirements and design with UML, or managing quality assurance, which are irrelevant to most research scientists. Instead, we focus on prioritization, estimation, and tracking—what do we really need to build, how long is it going to take, and where are we now?

The third of these topics introduces students to the last tool they meet in the course: a Web-based project management portal that combines a version-control repository browser with a bug-tracking system, mailing lists, a wiki, and other collaborative tools. The most famous such system is undoubtedly SourceForge (www.sf.net), but many lighter-weight alternatives exist, including one called DrProject (www.drproject.org), which my students at the University of Toronto are developing. As with IDEs, the most important thing isn’t which one the students see but rather that they understand how such tools can make their lives better.

As I said at the outset, improving quality improves productivity. Thus, the things that can help computational scientists get something done will also help to turn “computation” into “science.” But many obstacles remain—most of them social rather than technical. Consider, for example, this quote from Microsoft Research’s *Towards 2020 Science* report:

Software engineering for science has to address three fundamental issues: (i) dealing with datasets that are large in size, number, and variations; (ii) constructing new algorithms to perform novel analyses and syntheses; and (iii) sharing of assets across wide and diverse communities.¹

“Getting the right answer” didn’t make this list because it isn’t part of the computational science culture. I look forward to the day when it doesn’t make the list because it no longer needs to be said.



Acknowledgments

This work was originally supported by

the Advanced Computing Laboratory at Los Alamos National Laboratory, and more recently by the University of Toronto and the Python Software Foundation. I’m particularly grateful to Brent Gorda (Lawrence Livermore National Laboratory), Paul Dubois (now enjoying a well-earned retirement from the same institution), and Adam Goucher (who proofread my course notes as carefully as he used to test my code).

Reference

1. S. Emmott et al., *Towards 2020 Science*, tech. report, Microsoft, 2006; <http://research.microsoft.com/towards2020science/>.

Greg Wilson is an adjunct professor at the University of Toronto, and a contributing editor with *Dr. Dobb’s Journal* (www.ddj.com). His interests include scientific computing, software engineering, and computer science education. Wilson has a PhD in computer science from the University of Edinburgh. His most recent book is *Data Crunching* (Pragmatic Bookshelf, 2005). Contact him at gwwilson@cs.utoronto.ca.