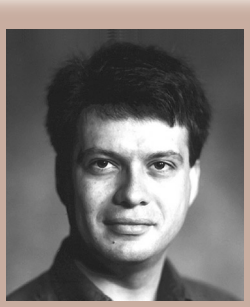
Editors: Andy Hunt and Dave Thomas ■ The Pragmatic Programmers
andy@pragmaticprogrammer.com ■ dave@pragmaticprogrammer.com

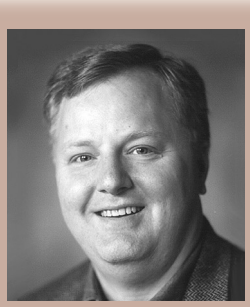
Ubiquitous Automation

Civilization advances by extending the number of important operations we can perform without thinking.
—Alfred North Whitehead

Welcome to our new column on software construction. We hope that you'll enjoy following this series as we explore the practical nuts-and-bolts issues of building today's software.

Before we start, though, we need to talk about the column's title. In some ways, "construction" is an unfortunate term, in that the most immediate (and often used) example involves building construction, with the attendant idea of an initial and inviolate architecture from which springs forth design and code.





Of course, nothing could be further from the truth. Software development, including its construction, is utterly unlike any other human endeavor. (Software development is also exactly the same as all other human endeavors, but that's a topic for another time.) Software is unique in both its malleability and its ephemerality; it is (to borrow the title of a Thomas Disch book) the dreams our stuff is made of. Yet we cannot simply wish a software system into being. We must create it using some semblance of engineering practice. This tension between the nonrepeatable, ill-defined, chaotic creative process and the scientific, repeatable, well-defined aspects of engineering is what causes so much heartburn in practitioners, authors, and scholars.

Is software engineering? Is it art? Is it craft? Many authors and pundits have compelling arguments for each of these viewpoints.[1–3] This is understandable because all the views have merit—software development is all these things, and this plurality is what causes so much misunderstanding. When should we act like engineers, and when shouldn't we?

Many organizations still view coding as merely a mechanical activity. They take the position that design and architecture are of paramount importance and that coding should be a rigorous, repeatable, mechanistic process that can be relegated to inexpensive, inexperienced novices. We wish these organizations the best of luck.

We'd like to think we know a better way. The SWEBOK (Software Engineering Body of Knowledge—view an online draft at www.swebok.org), for instance, states unequivocally that coding is far from a mechanistic translation of good design into code, but instead is one of those messy, imprecise human activities that requires creativity, insight, and good communication skills. Good software is grown organically and evolves; it is not built slavishly and rigidly. Design and coding must be flexible. We should not unduly constrain it in a misguided attempt to turn coders into robots.

However, the way we construct software should not be arbitrary. It must be perfectly consistent, reliable, and repeatable, time after time. And that's the topic of this first column.

—Andy Hunt and Dave Thomas

References

1. P. McBreen, *Software Craftsmanship: The New Imperative*, Addison-Wesley, Reading, Mass., 2001.
2. T. Bollinger, "The Interplay of Art and Science in Software," *Computer*, vol. 30, no. 10, Oct. 1997, pp. 128, 125–126.
3. W. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, Mass., 1989.

0740-7459/02/$17.00 © 2002 IEEE

If software construction is to involve engineering, the process must be *consistent* and *repeatable*. Without consistency, knowing how to build, test, or ship someone else's software (or even your own software two years later) is hard, if not impossible. Without repeatability, how can you guarantee the results of a build or a test run—how do you know the 100,000 CDs you just burned contain the same software you think you just tested?

The simple answer is discipline: software construction must be disciplined if it is to succeed. But people don't seem to come that way; in general, folks find discipline hard to take and even harder to maintain. Fortunately, there's a pragmatic solution: automate everything you can, so that the development environment itself provides the disciplined consistency and repeatability needed to make the process run smoothly and reliably. This approach leaves developers free to work on the more creative (and fun) side of software construction, which makes the programmers happier. At the same time, it makes the accountants happier by creating a development organization that is more efficient and less prone to costly human-induced errors and omissions.

Compilation automation

If you do nothing else, make sure that every developer on a project compiles his or her software the same way, using the same tools, against the same set of dependencies. Make sure this compilation process is automated.

This advice might seem obvious, but it's surprising how often it's ignored. We know teams where some developers compile using the automatic dependencies calculated by their integrated development environment, others use a second IDE, and still others use command-line build tools. The result? Hours wasted every week tracking down problems that aren't really problems, as each developer tests against subtly different executables generated by his or her individual compilation system.

Nowadays, automated compilations are pretty straightforward. Most IDEs offer a single-key "compile the project" command. If you're using Java from the command line, there's the Ant tool (http://jakarta.apache.org/ant/index.html). In other environments, the "make" system, or less common variants such as Aegis (www.pcug.org.au/~millerp/aegis/aegis.html), do the same job. Whatever the tool, the ground rules are the same: provide a single command that works out what needs to be done, does it, and reports any errors encountered.

Testing automation

During construction, we use *unit tests* to try to find holes in our software. (There are many other, equally important, reasons for using unit tests, but that's the subject of another article.) However, most developers we've seen skip unit testing or at best do it in an ad hoc way. The standard technique goes something like this:

1. Write a wad of code.
2. Get scared enough about some aspect of it to feel the need to try it.
3. Write some kind of driver that invokes the code just written. Add a few print statements to the code under test to verify it's doing what you thought it should.
4. Run the test, eyeball the output, and then delete (or comment out) the prints.
5. Go back to Step 1.

Let us be clear. This is not unit testing. This is appeasing the gods. Why invest in building tests only to throw them away after you've run them once? And why rely on simply scanning the results when you can have the computer check them for you?

Fortunately, easy-to-use automated testing frameworks are available for most common programming languages. A popular choice is the set of xUnit frameworks, based on the Gamma/Beck JUnit and SUnit systems (www.xprogramming.com/software.htm). We'll look at these frameworks

in later articles. For now, it's enough to point out that they are simple to use, composable (so you can build suites of tests), and fully automated.

Remember when we talked about making the build environment consistent? Well, the same applies to testing: you need repeatable test environments. However, that's not to say they should all be the same. Test in a wide variety of situations—some bizarre, some commonplace. Expose your code to the true range of target environments. Do it as early as possible during development. And do it as automatically as you can.

Once you can compile code with a button press, and test it soup-to-nuts, what's next? Integration, reviews, release, perhaps—all fertile ground for the seeds of automation. And then there's the lowly shipping process. Here automation is especially crucial. If you can't reliably go from source in the repository to code on a CD, how do you know what you're shipping? Every delivery is a stressful situation, where each manual step must be completed correctly if the four-billion-odd bits on the CD are to be the correct ones. Too often, though, transferring and replicating completed software is a heavily manual process, one that allows the introduction of all sorts of needless risks. What this part of the whole process needs is an *automated delivery build*.

What is an automated delivery build? At a minimum, it involves clean-room compilation, automated running of the unit tests, automated running of whatever functional tests can be automated, and automated collection of the deliverable components (not forgetting documentation) into some staging area. (Speaking of documentation, how much do you currently produce automatically? How much could you? After all, the more documentation you automate, the more likely it will be up-to-date.)

Once you've generated your shippable software, take it to the next level, and automate the testing of the result. Can the pile of bits you're proposing to deliver be installed in typical customer environments? Does it work when it gets there? Think about how much of this process you can automate, allowing the quality assurance staff to concentrate on the hard stuff.

Never underestimate the effort required to do all this. The cycle times of testing the process are large. Every small problem means starting again, compiling and testing and so on. However, there's a trick: don't leave build automation until the end. Instead, make it part of the very first project iteration. At this stage it will be simple (if incomplete). Then arrange for it to run at least daily. This has two advantages. First, the build's products give your quality assurance folks something to work with: anyone can take a build snapshot at any time. Second, you get to find out that the delivery build process failed the day it failed. The fixes will be immediate and incremental.

Other uses of automation

We've barely scratched this topic's surface. Consider that you needn't restrict automation to the build or even to just development. Perhaps your project has strict reporting requirements, review and approval processes, or other onerous secretarial chores. Where possible, try to automate these bits of drudgery as well. There's more to constructing software than just constructing software, and we can leverage automation throughout the process.

The cobbler's children

As we said in *The Pragmatic Programmer* (Addison-Wesley, 2000), the cobbler's children have no shoes. Often, people who develop software use the poorest tools to do the job. You can do better than that. Learn how to automate your environment—write macros if your IDE supports it; grab your favorite scripting language if it doesn't (Ruby is a good choice; see www.ruby-lang.org). Get your computer to do as much of the repetitive and mundane work as you can. Then you can move on to the really hard stuff. See you next issue.

Andy Hunt and **Dave Thomas** are partners in The Pragmatic Programmers, LLC. They feel that software consultants who can't program shouldn't be consulting, so they keep current by developing complex software systems for their clients. They also offer training in modern development techniques to programmers and their management. They are coauthors of *The Pragmatic Programmer* and *Programming Ruby*, both from Addison-Wesley, and speak on development practices at conferences around the world. Contact them via www.pragmaticprogrammer.com.