Editors: Andy Hunt and Dave Thomas ■ The Pragmatic Programmers ■ www.pragmaticprogrammer.com

Software Archaeology

Andy Hunt and Dave Thomas

"This isn't programming, this is archaeology!" the programmer complained, wading through the ancient rubble of some particularly crufty piece of code. (One of our favorite jargon words: www.tuxedo.org/~esr/jargon/html/entry/crufty.html.) It's a pretty good analogy, actually. In real archaeology, you're investigating some situation, trying to understand what you're looking at and how it all fits together. To do this, you must be careful to preserve the artifacts you find and respect and understand the cultural forces that produced them.

But we don't have to wait a thousand years to try to comprehend unfathomable artifacts; code becomes legacy code just about as soon as it's written, and suddenly we have exactly the same issues as the archaeologists: What are we looking at? How does it fit in with the rest of the world? And what were they thinking? It seems we're always in the position of reading someone else's code: either as part of a code review, or trying to customize a piece of open source software, or fixing a bug in code that we've inherited.

This analogy is such a compelling and potentially useful one that Dave, Andy, Brian Marick, and Ward Cunningham held a workshop on Software Archaeology at OOPSLA 2001 (the annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications). The participants discussed common problems of trying to understand someone else's code and shared helpful techniques and tips (see the "Tools and Techniques" sidebar).

Roll up your sleeves

What can you do when someone dumps 250k lines of source code on your desk and simply says, "Fix this"? Take your first cue from real archaeologists and inventory the site: make sure you actually have all the source code needed to build the system.

Next, you must make sure the site is secure. On a real dig, you might need to shore up the site with plywood and braces to ensure it doesn't cave in on you. We have some equivalent safety measures: make sure the version control system is stable and accurate (CVS is a popular choice; see www.cvshome.org). Verify that the procedures used to build the software are complete, reliable, and repeatable (see the January/February issue's column for more on this topic).

Be aware of build dependency issues: in many cases, unless you build from scratch, you're never really sure of the results. If you're faced with build time measured in hours, with multiple platforms, then the investment in accurate dependency management might be a necessity, not a luxury.

Draw a map as you begin exploring the

0740-7459/02/$17.00 © 2002 IEEE

code. (Remember playing Colossal Cave? You are in a maze of twisty little passages, all alike….) Keep detailed notes as you discover priceless artifacts and suspicious trapdoors. UML diagrams might be handy (on paper—don't get distracted by a fancy CASE tool unless you're already proficient), but so too are simple notes. If there are more than one of you on the project, consider using a Wiki or similar tool to share your notes (you can find the original Wiki at www.c2.com/cgi/wiki?WikiWikiWeb and a popular implementation atwww.usemod.com/cgi-bin/wiki.pl.

As you look for specific keywords, routine names, and such, use the search capabilities in your integrated development environment (IDE), the Tags feature in some editors, or tools such as Grep from the command line. For larger projects, you'll need larger tools: you can use indexing engines such as Glimpse or SWISH++ (*s*imple *W*eb *i*ndexing *s*ystem for *h*umans) to index a large source code base for fast searching.

The mummy's curse

Many ancient tombs were rumored to be cursed. In the software world, the incantation for many of these curses starts with "we'll fix it later." Later never comes for the original developers, and we're left with the curse. (Of course, we never put things off, do we?)

Another form of curse is found in misleading or incorrect names and comments that help us misunderstand the code we're reading. It's dangerous to assume that the code or comments are being completely truthful. Just because a routine is named readSystem is no guarantee that it isn't writing a megabyte of data to the disk.

Programmers rarely use this sort of cognitive dissonance on purpose; it's usually a result of historical accident. But that can also be a valuable clue: how did the code get this way, and why? Digging beneath these layers of gunk, cruft, and patch upon patch, you might still be able to see the original system's shape and gain insight into the changes that were required over the years.

Of course, unless you can prove otherwise, there's no guarantee that the routine you're examining is even being called. How much of the source contains code put in for a future that never arrived? Static analysis of the code can prove whether a routine is being used in most languages. Some IDEs can help with this task, or you can write ad hoc tools in your favorite scripting language. As always, you should prove assumptions you make about the code. In this case, adding specific unit tests helps prove—and continue to prove—what a routine is doing (see www.junit.org for Java and www.xprogramming.org for other languages).

By now, you've probably started to understand some of the terminology that the original developers used. Wouldn't it be great to stumble across a Rosetta stone for your project that would help you translate its vocabulary? If there isn't one, you can start a glossary yourself as part of your note-taking. One of the first things you might uncover is that there are discrepancies in the meaning of terms from different sources. Which version does the code use?

Duck blinds and aerial views

In some cases, you want to observe the dynamics of the running system without ravaging the source code. One excellent idea from the workshop was to use *aspects* to systematically introduce tracing statements into the code base without editing the code directly (AspectJ for Java is available at www.aspectj.org). For instance, suppose you want to generate a trace log of every database call in the system. Using something like Aspect-J, you could specify what constitutes a database call (such as every method named "`db*`" in a particular directory) and specify the code to insert.

Be careful, though. Introducing any extra code this way might produce a "Heisenbug," a bug introduced by the act of debugging. One solution to deal with this issue is to build in the instrumentation in the first place, when the original developers are first building and testing the software. Of course, this brings its own set of risks. One of the participants described an ancient but still used mainframe program that only works if the tracing statements are left in.

Whether diagnostic tracing and instrumentation are added originally or introduced later via aspects, you might want to pay attention to what you are adding, and where. For instance, say you want to add code that records the start of a transaction. If you find yourself doing that in 17 places, this might indicate a structural problem with the code—and a potential answer to the problem you're trying to solve.

Instead of hiding in a "duck blind" and getting the view on the ground, you might want to consider an aerial view of the site.

Instead of hiding in a "duck blind" and getting the view on the ground, you might want to consider an aerial view of the site. Synoptic, plotting, and visualization tools provide quick, high-level summaries that might visually indicate an anomaly in the code's static structure, in the dynamic trace of its execution, or in the data it handles. For instance, Ward Cunningham's Signature Survey method (http://c2.com/doc/SignatureSurvey) reduces each source file to a single line of the punctuation. It's a surprisingly powerful way of seeing a file's structure. You can also use visualization tools to plot data from the volumes of tracing information languishing in text files.

As with real archaeology, it pays to be meticulous. Maintain a deliberate

Tools and Techniques

The workshop identified these analysis tools and techniques:

- Scripting languages for
  –ad hoc programs to build static reports (`included by` and so on)
  –filtering diagnostic output
- Ongoing documentation in basic HTML pages or Wikis
- Synoptic signature analysis, statistical analysis, and visualization tools
- Reverse-engineering tools such as Together's ControlCenter
- Operating-system-level tracing via `truss` and `strace`
- Web search engines and tools to search for keywords in source files
- IDE file browsing to flatten out deep directory hierarchies of the source
- Test harnesses such as Junit and CPPUnit
- API documentation generation using Javadoc, doxygen, and so on
- Debuggers

Participants also identified these invasive tools:

- Hand-inserted trace statements
- Built-in diagnostic instrumentation, enabled in production code as well
- Instrumentation to log history of data values at interface calls
- Use of AspectJ to introduce otherwise invasive changes safely

pace; keep careful records. Even for a short term, quick patch that doesn't require you to understand the whole code base, keep careful records of what you've learned, what you've tried, what worked, and what didn't.

What were they thinking?

Archaeologists generally don't make wisecracks about how stupid a particular culture was (even if they did throw dead bodies into the only good drinking well). In our industry, we generally don't show such restraint. But it's important when reading code to realize that apparently bone-headed decisions that appear to be straight out of a "Dilbert" cartoon seemed perfectly reasonable to the developers at the time. Understanding "what they were thinking" is critical to understanding how and why they wrote the code the way they did.

If you discover they misunderstood something, you'll likely find that mistake in more than one place. But rather than simply "flipping the bozo bit" on the original authors, try to evaluate their strengths as well as weaknesses. You might find lost treasure—buried domain expertise that's been forgotten.

Also, consider to which "school of programming'" the authors belonged. Regardless of implementation language, West-Coast Smalltalkers will write in a different style from European Modula programmers, for instance. In a way, this approach gets us back to the idea of treating code as literature. What was the house style?

By understanding what the developers were thinking, what influenced them, what techniques they were fond of, and which ones they were unaware of, you will be much better positioned to fully understand the code they produced and take it on as your own.

Leaving a legacy

Given that today's polished code will inevitably become the subject of some future developer's archaeological dig, what can we do to help them? How can we help them comprehend "what we were thinking" and work with our code?

- Ensure the site is secure when you leave. Every file related to the project should be under version control, releases should be appropriately identified, and the build should be automatic and reliable.
- Leave a Rosetta stone. The project glossary was useful for you as you learned the domain jargon; it will be doubly useful for those who come after you.
- Make a simple, high-level treasure map. Honor the "DRY" principle: Don't duplicate information that's in the code in comments or in a design document. Comments in the code explain "why," the code itself shows "how," and the map shows where the landscape's main features are located, how they relate to each other, and where to find more detailed information.
- Build in instrumentation, tracing, and visualization hooks where applicable. This could be as simple as tracing "got here" messages or as intricate as an embedded HTTP server that displays the application's current status (our book, *The Pragmatic Programmer* (Addison-Wesley, 2000) discusses building testable code).
- Use consistent naming conventions to facilitate automatic static code analysis and search tools. It helps us humans, too.
- No evil spells. You know what the incantations sound like. Don't let the mummy's curse come back to haunt programmers later. The longer a curse festers and grows, the worse it is when it strikes—and curses often backfire against their creators first.

Acknowledgments

Many thanks to the other workshop organizers, Brian Marick and Ward Cunningham, and to the attendees: Ken Anderson, Vladimir Degen, Chet Hendrickson, Michael Hewner, Kevin Johnson, Norm Kerth, Dominik Kuropka, Dragos Manolescu, John McIntosh, Walter Risi, Andy Schneider, Glenn Vanderburg, and Charles Weir.

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 co-authors of *The Pragmatic Programmer* and *Programming Ruby*, both from Addison-Wesley. Contact them via www.pragmaticprogrammer.com.