ALVA L. COUCH

programming with technological ritual and alchemy

Alva Couch is an associate professor of computer science at Tufts University, where he and his students study the theory and practice of network and system administration. He served as program chair of LISA '02 and was a recipient of the 2003 SAGE Outstanding Achievement Award for contributions to the theory of system administration. He currently serves as Secretary of the USENIX Board of Directors.

couch@cs.tufts.edu

I HAVE TAUGHT PROGRAMMING-intensive courses to college students for over 20 years, and I cannot help but notice that the nature of programming has drastically changed in recent years. I teach a generation of students unfettered by my compulsion to understand and, as a substitute, fettered by their own compulsion to do and experience. Unlike the simple languages I employed to learn programming, my students employ complex frameworks, exploit primitive crowdsourcing to come up with solutions, and engage in a shamanistic ritual of dance and pattern to produce software that works, but whose complete function they do not and cannot fully understand. So? As they might say, "What's wrong with that?"

Technological Shamanism

I call the practice of creating software systems from ritual *technological shamanism*. Programming via socially derived ritual makes a surprising amount of sense and is often productive. But to understand its nature, we need to carefully draw some important distinctions between the concepts of "pattern," "example," and "ritual."

A *design pattern* is a proven and reusable approach to solving a specific kind of programming problem [1]. It is called a "pattern" because it specifies the nature of a programming solution without the details. The original description of patterns expressed a pattern as an object-oriented program that operates on objects that implement specified interfaces. These objects represent details the programmer must supply. For example, a "sorting" pattern works on objects that implement a comparison interface. One uses a pattern by implementing the interfaces it requires.

More recently, "design patterns" have acquired a general popular meaning outside the strictures of object-oriented programming, as reusable program fragments with some details missing, in which the method whereby a programmer fills in details is well documented. A pattern is a code fragment that has been proven to work through widespread application, and where its applicability, proven uses, and limits are carefully documented [2]. There are several Internet repositories in which one can search for patterns to solve various problems.

Patterns are a powerful way of archiving programming knowledge in an accessible form. One key part of a pattern—independent of the multiple ways that one can be described—is a statement of its limits, i.e., "how far you can stretch it without breaking it." Applying a pattern—in principle—substitutes for having the knowledge to create it. Using patterns creates code that is, in many cases, better than code written from scratch and based upon full knowledge of system function.

However, the contemporary programmer is not always fortunate enough to have true design patterns in hand. Documenting a pattern is a labor-intensive task. Thus, programmers turn to Internet sources containing "examples" that substitute for patterns, in the sense that the programmer can twist an example to accomplish a goal and/or combine examples toward a new purpose. But *examples are not patterns*. There is no documentation of what can be changed. There is no documentation as to applicability, scope, or limits; there is no claim of broad applicability. There is not even the implicit claim that an example has been rigorously tested.

But an example that has proven useful in a broad variety of contexts—even in a limited way—is not quite an example anymore. It is not yet a pattern and lacks many kinds of documentation. But it is a "ritual" that "usually works." There is some value in distinguishing between "examples," as untried, versus "rituals," as partly validated.

One might compare patterns, examples, and rituals with pharmaceuticals, folk medicines, and folk remedies. A pharmaceutical—like a pattern—has well-documented effects. A folk medicine, like a programming example, might create some effects when, e.g., made into tea. A folk remedy—like a programming ritual—is contrived from a folk medicine through some experience of success, but *without complete understanding of its effects*. In other words, a programming "ritual" is somewhat like a new drug without FDA approval. A ritual "just works," for some partly understood reason.

One weakness of using rituals is that finding new rituals requires some mix of guesswork, experimentation, and/or deep knowledge. Modifications to existing rituals—however minor—are *not* guaranteed to work, any more than modifications of gourmet recipes are; only the master chef knows what can be substituted.

As an example of this, I assigned my students to write a simple document search program in Hadoop, thinking that this was a straightforward program. Not! We got caught in the netherworld between Hadoop 0.19 and Hadoop 0.20, in which the syntax changed enough in 0.20 to invalidate all of the 0.19 tutorials. The tutorial examples said nothing about how to propagate the search terms to the mapper processes. Worse, the *only* candidate variable that we knew enables that kind of propagation had a type that was marked as deprecated! Through some educated guesswork and experimentation, we found out how to do it, though others before us did not fare as well,and we found one Internet lament that text search—which Hadoop was designed to do well—was impractical in Hadoop!

How did we succeed? Well, it is difficult to admit it, but we succeeded by locating a shamanistic ritual with a closely related outcome, searched the Web for related rituals, and then guessed what to do and verified the guess through experimentation, *thus creating our own personalized ritual*. I call this "ritual" and not "pattern," because in the process of making the program work, *we did not obtain a comprehensive idea of why it works, or its limits!*

A little play with modern Web frameworks such as Ruby on Rails, Symfony, and Cake will demonstrate why modern programmers think this

way: one cannot deviate from predefined rituals without courting disaster and/or inconceivable behavior. Frameworks have reached the complexity point where documenting their complete function to a programmer is impractical, or perhaps I should call it "unempowering." The total workings of modern programming frameworks are not that useful to know for someone using them. So we resort to shamanism and employ rituals that the creators of the frameworks kindly provide for us, or we guess and, from experimentation, create our own rituals to taste.

Engaging in ritual rather than understanding is not just exhibited by programmers. System administrators often crowdsource their configurations of software, for desktops or servers, based upon what rituals are found to work by others. The job of the system administrator is *not* to understand but, rather, just to repair the problem. Time pressure often limits personal exploration, so that successful repairs by others are important to know. Often, the quickest approach to troubleshooting is to mine "rituals that work" from the Internet. The mailing lists are full of these simple—but generally effective—uses of crowdsourcing.

From Shamanism to Alchemy

By this time, the reader may think I am advocating a return to barbarism, throwing away the knowledge of the past. I am instead pointing out that we are *already* barbaric, in the sense that our technology has already vastly surpassed our understanding. How empowering is it, for example, to take apart a cell phone? The physical structure of a cell phone is not empowering to someone looking to understand its function, which is mostly hidden.

And the barbarians are winning, because they can produce programs faster than the civilized programmers!

The modern technological shaman, like the primitive shaman, trusts his or her senses and engages in a kind of science. The difference between primitive shamanism and technological shamanism lies in what the shaman's senses include. The technological shaman has the observational powers of the Internet-connected world available and can crowdsource a solution to a mystifying problem simply by querying the proper mailing lists. The appropriate response to "Doctor, it hurts if I do this" is usually "Don't do that; do this"; a non-working ritual is countered with a working ritual, but without a satisfying explanation of why one ritual does not work while the other does.

Like a folk remedy, a modern ritual gains validity through direct observation of when it does and doesn't work. Thus its validity grows with application and observation, including observation of what requirements it does not meet. One severe downside is that there is no such thing as a "secure ritual"; a reasonably complete security analysis would transform it into a pattern!

Crowdsourced solutions are laced with shamanistic rituals that might do nothing, but were part of an initial pattern. I had a student who always put the statement "X=X" into his Matlab program before using a variable "X." This was ritual rather than knowledge (the statement does nothing); but it was extremely difficult to convince him—even with objective evidence to the contrary—that the statement was not needed. This shamanistic ritual reminded me of the rituals of aboriginal tribes who feed wooden birds because it seems to help the crops. Why do it? Because it might help and does not hurt!

One thing that greatly influenced my thinking on social ritual in technology was Jim Waldo's Project Darkstar at Sun Microsystems. Darkstar attempted

to analyze the requirements for interactive role-playing games [3]. To me, the most surprising finding from Darkstar is that young people do not approach interactive RPGs as adults do; bugs are "features," workarounds are "rituals," and software quality is defined in terms of not losing long-term state (although losing short-term state is acceptable). In other words, if you engage in a ritual, the game should not erase your character (a matter of weeks of development), but erasing your character's development for the past day is relatively acceptable! The quality of the RPG system is relative to how it reacts to ritual and whether its reactions to ritual are appropriately bounded. Some Web frameworks could learn from this definition of quality!

So, what is my advice to young students of programming? I do not advise them to program as a "civilized" adult like myself; I am less facile than they are! I do not advise them to reject shamanism and ritual; technological ritual is a basic part of modern survival. I do tell them to develop their own observational skills to a high art, so that they can track a "personal alchemy" that describes *how their rituals interact*. The result of crowdsourcing this "personal alchemy" is a shared "technological alchemy" describing how rituals can be combined to achieve new goals. This social emergence of order—and not the traditional practice of reading and understanding the source code—is already the key to productive programming in the new world of large-scale frameworks.

From Alchemy to Chemistry

It is with some poetic justice that I—having shown no aptitude for undergraduate chemistry—am now put in the position of advocating its value! In the same way that alchemy became chemistry, we need a new "chemistry of programming" that describes how to combine "elements" (our rituals) into "molecules" (more involved rituals) that achieve our ends.

By chemistry, I am not referring to the precise rules of component-based programming or object-oriented programming but, instead, to the fuzzily defined rules by which rituals are combined into new rituals, without full knowledge of the structure behind the rituals. "Programming chemistry" is a matter of classifying what rituals are safe to combine, what combinations are questionable, and what combinations are likely to explode!

Am I advocating throwing away detailed knowledge? Certainly not. But this kind of knowledge—while valuable to the developers of a framework—is not empowering to the average programmer of a framework. Or, rather, it is as empowering as knowing the physics of electrons is to the chemical engineer. This is not a matter of throwing away knowledge but, instead, packaging it in a much more digestible form, in terms of what programmers should and should not do.

Generations

The difference between me and my students is quite generational. I was taught to be compulsive about knowing, in the sense that I will not stop peeling away layers of a thing until I know everything about how it works. This compulsion was empowering for me but is not empowering for my students. Or, it would be better to say, it is about as empowering for them as taking apart their cell phones to see how they work! To my students, I am somewhat of a dinosaur, and to me, my students are the new shamans of technology, engaging in dance and ritual to produce working code.

But my students are not lacking in ambition; they engage in their own unique flavor of lifelong learning. They learn the rituals that work, and the alchemy between rituals: their own descriptions of how to transmute base software components into "gold." But, somehow, it all works, and often it does result in "gold."

REFERENCES

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, illustrated edition, 1994.

[2] See, e.g., Chapter 12 of Roger Pressman, *Software Engineering, a Practitioners' Approach*, 7th edition, McGraw-Hill, 2009.

[3] Project Darkstar: http://www.projectdarkstar.com.