Empirical Software Engineering

As researchers investigate how software gets made, a new empire for empirical research opens up

Greg Wilson and Jorge Aranda

Software engineering has long considered itself one of the hard sciences. After all, what could be "harder" than ones and zeroes? In reality, though, the rigorous examination of cause and effect that characterizes science has been much less common in this field than in supposedly soft disciplines like marketing, which long ago traded in the gut-based gambles of "Mad Men" for quantitative, analytic approaches.

A growing number of researchers believe software engineering is now at a turning point comparable to the dawn of evidence-based medicine, when the health-care community began examining its practices and sorting out which interventions actually worked and which were just-so stories. This burgeoning field is known as empirical software engineering and as interest in it has exploded over the past decade, it has begun to borrow and adapt research techniques from fields as diverse as anthropology, psychology, industrial engineering and data mining.

The stakes couldn't be higher. The software industry employs tens of millions of people worldwide; even small increases in their productivity could be worth billions of dollars a year. And with software landing our planes, diagnosing our illnesses and keeping track of the wealth of nations, discovering how to make programs more reliable is hardly an academic question.

Greg Wilson leads the Software Carpentry project, which teaches basic software skills to scientists and engineers. He received his Ph.D. in computer science from the University of Edinburgh. Jorge Aranda received his Ph.D. from the University of Toronto and is currently a post-doctoral fellow at the University of Victoria. Their blog "It Will Never Work in Theory" (http://neverworkintheory.org) is a survey of recent results in empirical software engineering. E-mail addresses: gvwilson@third-bit.com, jarandag@gmail.com

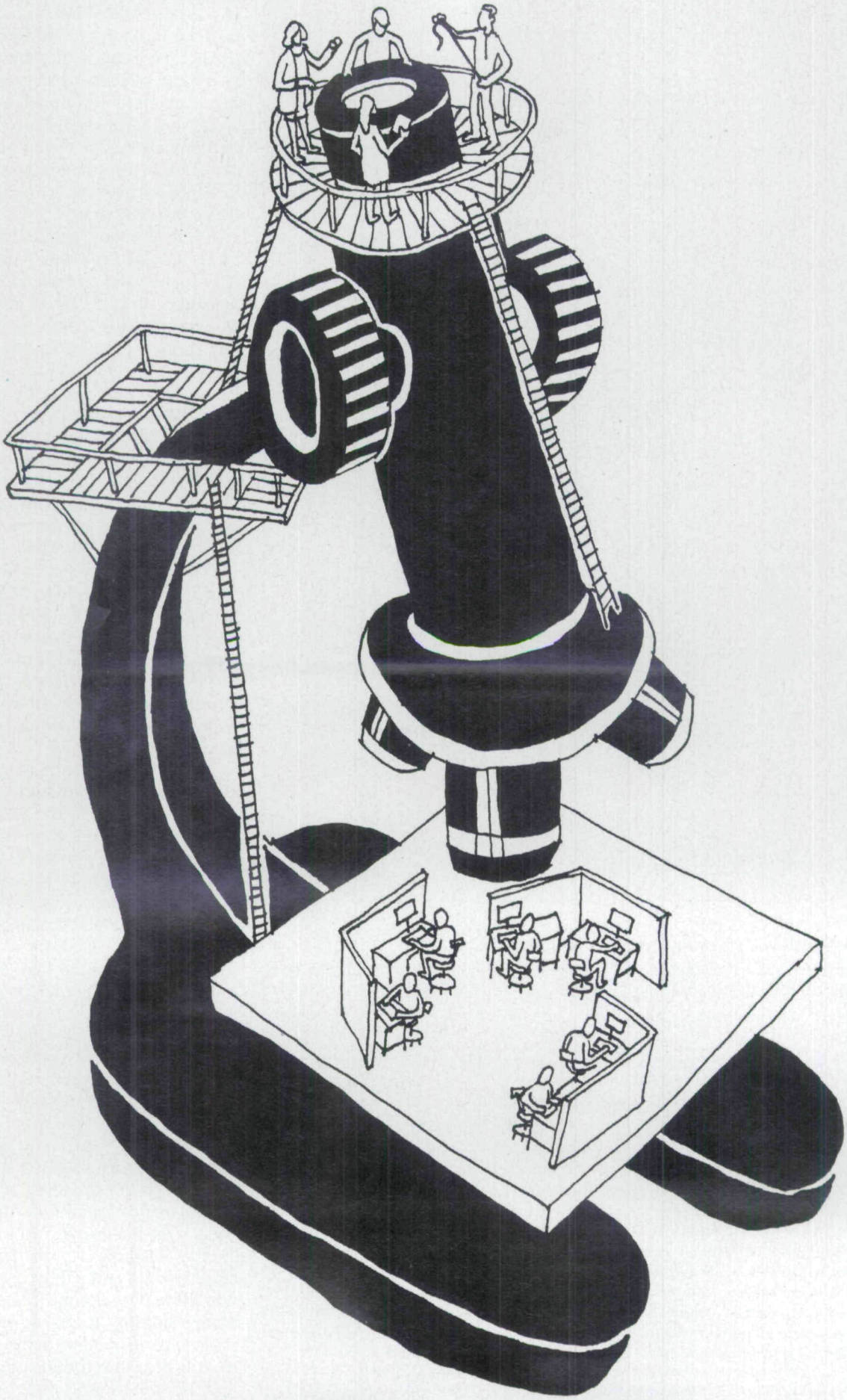
Where We Are

Broadly speaking, people who study programming empirically come at the problem from one of two angles. To some, the phrase *software engineering* has always had a false ring. In practice, very few programmers analyze software mathematically the way that "real" engineers analyze the strength of bridges or the resonant frequency of an electrical circuit. Instead, programming is a skilled craft, more akin to architecture, which makes the human element an important (some would say *the* important) focus of study. Hollywood may think that programmers are all solitary 20-something males hacking in their parents' basement in the wee hours of the morning, but most real programmers work in groups subject to distinctly human patterns of behavior and interaction. Those patterns can and should be examined using the empirical, often qualitative tools developed by the social and psychological sciences.

The other camp typically focuses on the "what" rather than the "who." Along with programs themselves, programmers produce a wealth of other digital artifacts: bug reports, email messages, design sketches and so on. Employing the same kinds of data-mining techniques that Amazon uses to recommend books and that astronomers use to find clusters of galaxies, software engineering researchers inspect these artifacts for patterns. Does the number of changes made to a program correlate with the number of bugs found in it? Does having more people work on a program make it better (because more people have a chance to spot problems) or worse (because of communication stumbles)? One sign of how quickly these approaches are maturing is the number of data repositories that have sprung up, including the University of Nebraska's Software Artifact Infrastructure Repository, the archives of NASA's highly influential Software Engineering Laboratory and the National Science Foundation–funded CeBASE, which organizes project data and lessons learned. All are designed to facilitate data sharing, amplifying the power of individual researchers.

The questions we and our colleagues seek to answer are as wide-ranging as those an anthropologist might ask during first contact with a previously unknown culture. How do people learn to program? Can the future success of a programmer be predicted by personality tests? Does the choice of programming language af-

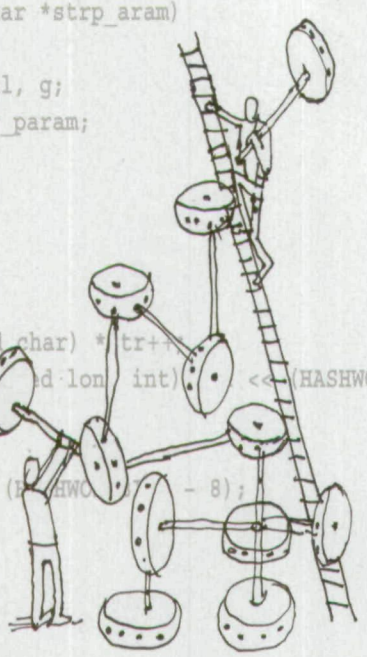
Figure 1. Making software is a signature activity of our era that has produced a canon of beliefs and practices, yet only recently have the tools of social science and empirical investigation been systematically applied on a large scale to the software enterprise. Empirical software engineering is the emerging discipline of acquiring a rigorous, evidence-based understanding of what we know about making software, what we don't know and what we can learn about it using the tools of empirical research.



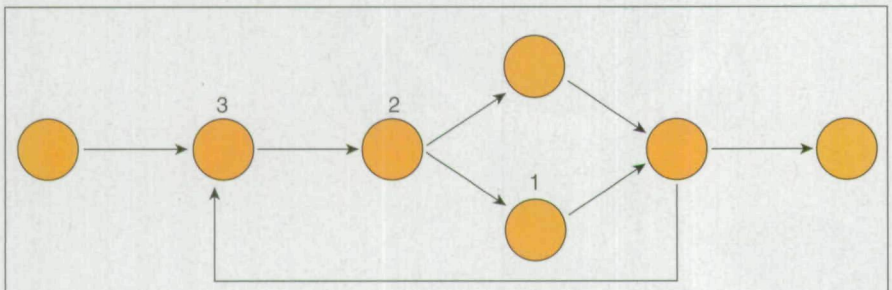
a. original source code

```
unsigned long int
__hash_string (const char *strp_aram)
{
  unsigned long int hval, g;
  const char *str = str_param;

  hval = 0;
  while (*str != '\0')
    {
      hval <<= 4;
      hval += (unsigned char) *str++;
      g = hval & ((unsigned long int) 0xf << (HASHWORDBITS - 4));
      if (g != 0)
        {
          hval ^= g >> (HASHWORDBITS - 8);
          hval ^= g;
        }
    }
  return hval;
```



b. McCabe's cyclomatic complexity of the source code above is equal to 3



c. Halstead's software science metric adds numerous dimensions

length	97
volume	526
level	0.036
number of mental discriminations	14,490

Figure 2. There are many reasons to measure the complexity of computer code. The simplest metric of complexity for the snippet of code in *(a)* above, written in the C language, is the number of lines. Next simplest is often the number of C functions; in the sample code here, one. At a higher level, McCabe's cyclomatic complexity allows us to calculate the number of independent paths in the code and render them in a control-flow graph *(b)*. Halstead's software science metric analyzes complexity based on number and redundant use of textual elements, without reference to program structure. Herraiz and Hassan tested whether any of the higher measures provided more diagnostic information about complexity than the simplest measure, lines of code. (Figure adapted from material in Herraiz, I., and A. E. Hassan. 2011. Beyond lines of code: Do we need more complexity metrics? In *Making Software: What Really Works and Why We Believe It*. Sebastopol, CA: O'Reilly Media, 125–144.)

fect productivity? Can the quality of code be measured? Can data mining predict the location of software bugs? Is it more effective to design code in detail up front or to evolve a design week by week in response to the accretion of earlier code? Convincing data about all of these questions are now in hand, and we are learning how to tackle many others.

Along the way, our field is grappling with the fundamental issues that define any new science. How do we determine the validity of data? When can conclusions from one context—one programming team, or one vast project, like the development of the Windows Vista operating system—be applied elsewhere? And crucially, which techniques are most appropriate for answering different kinds of questions?

Some of the most exciting discoveries are described in a recent book called *Making Software: What Really Works, and Why We Believe It*, edited by Andy Oram and Greg Wilson (O'Reilly Media, 2011), in which more than 40 researchers present the key results of their work and the work of others. We'll visit some of that research to give an overview of progress in this field and to demonstrate the ways in which it is unique terrain for empirical investigation.

Quality by the Numbers

A good example of the harvest that empirical studies are generating relates to one of the holy grails of software engineering: the ability to measure the quality of a program, not by running it and looking for errors, but by automated examination of the source code itself. Any technique that could read a program and predict how reliable it would be before it is delivered to customers would save vast sums of money, and probably lives as well.

One consistent discovery is that, in general, the more lines of code there are in a program, the more defects it probably has. This result may seem obvious, even trivial, but it is a starting point for pursuing deeper questions of code quality. Not all lines of code are equal: One line might add 2 + 2 while another integrates a polynomial in several variables and a third checks to see whether several conditions are true before ringing an alarm. Intuitively, programmers believe that some kinds of code are more complex than others, and that the more complex a piece of

code is, the more likely it is to be buggy. Can we devise some way to measure this complexity? And if so, can the location of complexity hot spots predict where defects will be found?

One of the first attempts to answer this question was developed by Thomas J. McCabe and is known as *cyclomatic complexity*. McCabe realized that any program can be represented as a graph whose arcs show the possible execution paths through the code. The simplest graph is a straight chain, which represents a series of statements with no conditions or loops. Each *if* statement creates a parallel path through the graph; two such statements create four possible paths. Figure 2 shows a snippet of code extracted from a cross-platform download manager called Uget. The graph in part *(b)* shows the paths through the code; each *if* and loop adds one unit of complexity, giving this code an overall complexity score of 3.

Another widely used complexity measure is Maurice Halstead's software science metric, which he first described in 1977. Instead of graph theory, it draws on information theory and is based on four easily measured features of code that depend on the number of distinct operators and operands, their total count, how easy they are to discriminate from one another and so on. Figure 2*(c)* shows the values for the sample piece of code in *(a)*.

Hundreds of other metrics have been developed, published and analyzed over the past 30 years. In their chapter, Herraiz and Hassan use statistical techniques to explore a simple question: Are any of these metrics actually better at predicting errors than simply counting the number lines of source code? Put another way, if a complexity metric is highly correlated with the number of lines of source code, does it actually provide any information that the simpler measure does not?

For a case study, Herraiz and Hassan chose to examine the open-source Arch Linux operating system distribution, which yielded a sample of 338,831 unique source files in the C language. They calculated the measures discussed above, and several others, for each of these files, taking special account of header files (those consisting mainly of declarations that assist in code organization). They found that for nonheader files, where programs actually do their work, all the metrics tested showed a very high degree of correlation with lines of code. Checking for generalizability, the effect held for all but very small files. The authors drew a clear lesson: "Syntactic complexity metrics cannot capture the whole picture of software complexity." Whether based on program structure or textual properties, the metrics do not provide more information than simply "weighing" the code by counting the number of lines.

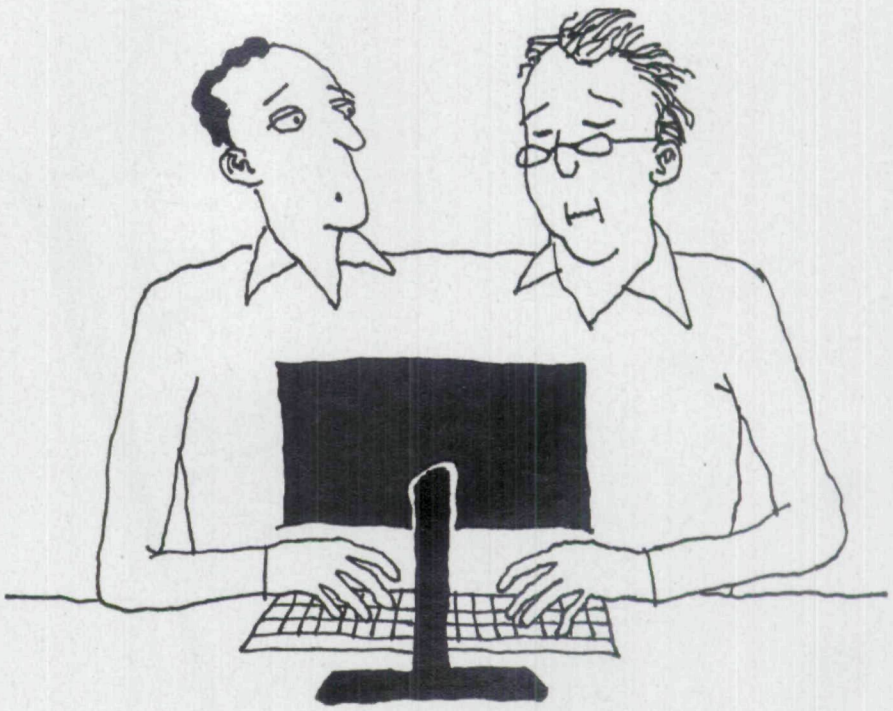


Figure 3. To the uninitiated, pair programming seems an out-there technique. Two programmers work at the same machine, usually with one piloting the keyboard, the other flying in the rear seat of the cockpit. Many innovators in software practices laud pair programming. Whether it should be preferred can be and has been investigated rigorously via a wide range of experiments that test different personality matches, different protocols for determining to whom control of the keyboard is assigned and different types of problems.

Like all negative results, this one is a bit disappointing. However, that does not mean these metrics are useless—for example, McCabe's scheme tells testers how many different execution paths their tests need to cover. Above all, the value here is in the progress of the science itself. The next time someone puts forward a new idea for measuring complexity, a validated, empirical test of its effectiveness will be there waiting for them.

Two-Headed Approach

Metrics research looks at the code that programmers produce, but at least as much research effort has focused on *how* they produce it. An interesting case is *pair programming*, a work style that burst onto the scene in the late 1990s (though people have been using it informally for as long as programming has existed). In pair programming, two programmers sit at a single workstation and create code together. The *driver* handles the keyboard and mouse, while the *navigator* watches, comments and offers corrections. Many programmers have noticed over the years that duos like this seem to produce code more quickly and with fewer bugs. If asked, they would probably say that the benefits arise because different people naturally notice different sorts of things, or because not having to type gives the navigator more time to think, or possibly that having an audience makes the driver think more carefully. But is pair programming actually better, and if so, which of the possible explanations is the reason?

The first empirical study of pair programming, by Temple University professor John Nosek in 1998, studied 15 programmers, 5 working alone (the control group), the other 10 in 5 pairs. A challenging programming problem was given, with a time limit of 45 minutes. The results were statistically significant. Solutions produced by the pairs took 60 percent more to-

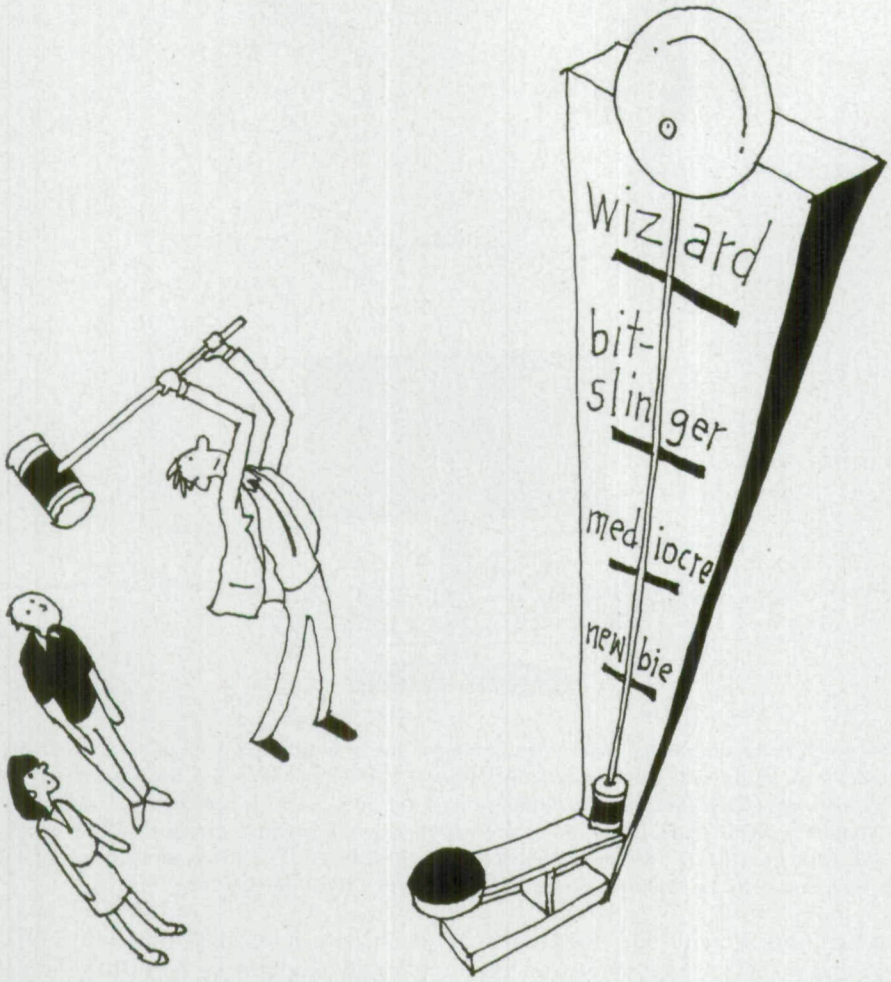


Figure 4. Measuring programmer aptitude and skill is the difficult art of devising a number line from "zero" to "great."

tal time, but dividing the total time by two, they completed the tasks 20 percent faster. And the author of the study reported that the pairs produced higher-quality code.

Subsequent studies with larger groups over longer periods of time, summarized by Laurie Williams at North Carolina State University, have expanded on these early results. Pair programmers tend to produce code that is easier to understand, and they do so with higher morale. Their productivity may fall initially as the programmers adjust to the new work style, but productivity recovers and often surpasses its initial level as programmer teams acquire experience. Related studies have also given insight into what actually happens during pair programming. For example, successful pairs often don't work together for a full day—pairing can be mentally exhausting. Changing partners regularly seems to help, and swapping roles periodically between driver and navigator helps keep programmers engaged.

These results can even influence hardware design. Some teams have two people who both do best when they have control of the keyboard. Why not provide dual keyboards and mice? Jan Chong and Tom Hurlbutt tested this approach at Stanford University in 2007 and found it a worthwhile advance in pair programming, a finding subsequently supported by additional studies by Andreas Höfer at the University of Karlsruhe and by Sallyann Freudenberg, a software coach. Once again, what we're seeing is the science rapidly improving as early studies generate questions and methods for follow-on research.

What we also see is that being right isn't always enough to effect change. Despite the accumulation of evidence in its favor, pair programming is still very much the exception in industry. Managers and programmers alike often brush the data away, clinging to the idea that putting two people on one job must double staffing requirements and therefore cannot deliver efficiency. This is an unfortunate phenomenon in many fields. Some people resist change tenaciously, even in the face of evidence and at the risk of failure.

Getting Good Programmers

Another holy grail of empirical software engineering research is finding a way to tell good programmers from bad. A widely quoted piece of folklore says that good programmers are 28 times better than average (or 40, or 100, or some other large number). As teacher and developer Steve McConnell discusses in *Making Software*, the precise quantification of this assertion may be suspect, given that the definition of "better" is elusive in any knowledge-based work. Yet everyone who has programmed for a living knows that there really are huge differences in productivity and capability among programmers. Other than hiring someone and watching her work for a couple of years, how can we spot the stars?

Jo Erskine Hannay at Simula Research Laboratory has addressed the issues of talent and expertise by breaking the question into parts: Can we actually define what it means to be a good software developer? Can we reliably determine that one developer is better than another? And if we can't, should we surrender on those questions and focus instead on tools and techniques?

Hannay zeroes in on a field called *individual differences* research, in which individuals are classified by characteristics that separate one from another, such as personality. Although some pseudoscientific schemes used by human resources departments and dating websites have given personality testing a bad reputation, modern programs such as the five-factor model have solid scientific foundations (or as scientists might put it, we have construct validity for the concept of measuring personality). The five-factor model and similar protocols address dimensions that include extraversion, agreeableness, conscientiousness, emotional stability, and willingness and ability to learn from experience. We also know that programmers tend to have certain personality types and that they vary less in personality than the average population. Can we use findings like these to discover good programmers?

The short answer is no. Large meta-analyses and further studies by Han-

Figure 5. Two grails of software research are the ability to compare programmers and determine which is "better," and the ability to algorithmically analyze code to determine its quality. The basic goal of both is better software. One route to better software could be credentialing and licensure of software engineers like that imposed on civil and other engineers. Mastery of a body of knowledge would be the gateway, but at present that would be establishing a required body of knowledge before demonstrating empirically that it is the right body of knowledge.

nay and others conclude that a programmer's personality is not a strong predictor of performance. The people who swear by their beliefs about personality and programmer success have now been given reason to assess their position critically, along with methodological support for doing so.

Credible Accreditation

If we can't predict who will become a good programmer, can we at least certify who already is one? Would it help if software developers were required to undergo some sort of certification? In most of the industrialized world, engineering professionals must be licensed; a preliminary exam permits them to practice and gain experience as an engineer-in-training, followed some years later by a deeper exam, often specialized according to their discipline. Successful candidates are then allowed to offer their services to the public. Could such a program be developed for software engineers? And would it actually make anything better?

If done right, such a program would be based on a codified body of necessary knowledge and best practices, just as exams in civil, mechanical, and electrical engineering are. A major effort under way right now is the Software Engineering Body of Knowledge project (SWEBOK), sponsored by the IEEE Computer Society. Their intentions are noble, and conscientious professionals are steadily assembling a potential set of standards.

We are skeptical of this work, for the very reasons that we are committed to empirical software engineering research. We believe that it puts the cart before the horse, that we simply don't yet know enough about what actually works and what doesn't to define such standards. In place of a trustworthy gateway, we fear that a Software Engineering Body of Strong Opinion would create a false belief that we know more than we do at present about how software can and should be made, how the quality of software can be rigorously determined and how programmers should take on the job of programming. That knowledge is the very thing empirical software researchers are stalking.

For example, in the 1990s a group of respected software designers combined forces to create a graphical notation for computer programs called the Unified Modeling Language (UML), which was intended to fill the role of blueprints and circuit diagrams in civil and electrical engineering. Despite a great deal of hype, UML never really caught on: Almost everyone who earns a degree in computer science learns about UML at some point, but very few programmers use it voluntarily (although some are obliged to use it in order to satisfy the terms of government procurement contracts).

In 2007, Mario Cherubini, Gina Venolia, Rob DeLine and Andrew Ko studied what kinds of diagrams programmers actually draw and why they draw them. They found that in almost all cases, programmers' sketches were transient in nature; they were meant to be aids to conversation rather than archival documentation. They also found that the cost of turning hastily drawn whiteboard doodles into formal instructions was greater than the value of the diagrams to the programmers who were creating them. Companies selling UML drawing tools ignore this awkward result, but we are hopeful that a younger generation of software designers will incorporate into their work both find-

Figure 6. Like the rest of social science research, empirical software engineering faces challenges in establishing that results are generalizable. Do findings scale from small shops to large, from small projects to large, from industrial, tightly managed settings to the milieu dubbed the open-source bazaar by Eric Raymond?

ings like these and the research methods behind them.

Life Imitates Code

In 1967, only partly as a wry joke, Melvin Conway coined his eponymous law:

> Any organization that designs a system...will produce a design whose structure is a copy of the organization's communications structure.

In other words, if the people writing a program are divided into four teams, the program they create will have four major parts.

Nachi Nagappan, Christian Bird and others at Microsoft Research evaluated the validity of Conway's law by examining data collected during the construction of Windows Vista. Vista consists of thousands of interrelated libraries and programs called *binaries*. When an error occurs, the breakdown can usually be traced to a fault in a single binary or to a breakdown in the interaction between binaries. Nagappan, Bird, and their team used data mining to explore which aspects of software construction correlated with faults. They found that when work occurred in alignment with Conway's law—that is, when the structure of the team and the structure of the code mirrored each other—code contained fewer bugs, whereas work that crossed team boundaries increased failure-proneness.

Nagappan and his collaborators then used their data to *predict* failure-proneness by locating code produced by multiple groups or at the interface of multiple groups. Contrary to digital folklore, they found that geographic separation between team members didn't have a strong impact on the quality of their work. What did matter was organizational separation: The farther apart team members were in the company organization chart, the great-

er the number of faults in the software they produced. This result is applied science at its best: It is both surprising and actionable.

Open Access

At present, a tremendous enabler of empirical software engineering research is the open-source software movement, which is rapidly generating a freely available accumulation of code along with complete archives of the communications between developers. In an open-source setting, programmers collect around software projects to produce applications that they want to see available for free. The developers are often in different places and time zones, so communication occurs via email and online forums. The code and communication records are accessible to all via websites, so interested developers can join the project at any stage to share expertise, troubleshoot and add to the source code.

These electronic repositories are a software-engineering researcher's paradise. They constitute a historical record of the life of a project, including all of the dead ends and debates, the task assignments, the development of team structure and many other artifacts. With thoughtful and targeted searches, researchers can explore topics such as how newcomers adapt to a software project's culture. They can test prediction engines to assess the validity of theories about project structure and code development. Bug-tracking records and the interpersonal interactions involved in solving software flaws serve as a narrative of the incremental improvement of code quality. Before the open-source community took on its present form, this kind of access to project archives was available only to investigators in corporate research units.

As the open-source movement developed, there was a feeling that researchers should treat it as a special case in the realm of software engineering. Eric Raymond, president of the Open Source Initiative, highlighted the differences between open-source and industrial projects when he compared them to a bazaar and a cathedral. Industry is the cathedral, in which projects are built according to carefully detailed plans, with attendant hierarchy, role divisions, order and dogma. The bazaar is bustling, free-form, organic and shaped by the aggregate actions of the crowd. Researchers bringing results from the open-source world met skepticism about whether their findings could be generalized to the rest of the community. In fact, research has demonstrated that the distinctions between the two worlds are often illusory. There are cathedrals in the open-source sphere and bazaars in the closed-source. Similar social and technical trends can be documented in both, and researchers have come to appreciate the dividends that come from comparing the two.

The work of Guido Schryen at the University of Freiberg and Eliot Rich at the University at Albany, SUNY, is instructive about how to ask and answer questions about the two worlds. In a 2010 paper they addressed a much-debated and critically important issue: Which model leads to better security, open- or closed-source software? Security is a formidable concern for any software that will come within reach of networks. Schryen and Rich examined the security-vulnerability announcements and the release (or nonrelease) of patches (software fixes) for 17 widely deployed software packages. Proponents of open-source software have argued that its characteristically wide developer base must lead to better review and response to security issues. An opposing argument holds that closed-source and industrial projects have more direct motivation to find and fix security flaws. Schryen and Rich sorted the packages they studied within categories such as open- and closed-source, application type (operating system, web server, web browser and so on), and structured or loose organization. They found that security vulnerabilities were equally severe for both open- and closed-source systems, and they further found that patching behavior did not align with an open–versus-closed source divide. In fact, they were able to show that application type is a much better determinant of vulnerability and response to security issues, and that patching behavior is directed by organizational policy without any correlation to the organizational structure that produced the software. Whether open- or closed-source software was more secure turned out to be the wrong question to ask. We do not expect that the lines between the open- and closed-source worlds will be so blurred in every aspect of software engineering, but results like these show how the massive amount of information available as a byproduct of open-source development can be put to scientific use.

As in any applied science, the ultimate measure of success for all of this work will be change—change in the tools used to develop software, change from current practices to those that are provably better and most importantly, change in what is and is not accepted as proof.

Bibliography

Bird, C., D. Pattison, R. D'Souza, V. Filkov and P. Devanbu. 2008. Latent social structure in open-source projects. SIGSOFT '08/FSE-16: *Proceedings of the 16th ACM SIGSOFT Symposium on Foundation of Software Engineering*:24–35.

Chong, J., and T. Hurlbutt. 2007. The social dynamics of pair programming. *Proceedings of the 29th International Conference on Software Engineering*:354–363.

Freudenberg, S., P. Romero and B. du Boulay. 2007. Talking the talk: Is intermediate-level conversation the key to the pair programming success story? *Proceedings of AGILE 2007*:84–91.

Glass, Robert. L. 2002. *Facts and Fallacies of Software Engineering*. Boston: Addison-Wesley.

Halstead, M. 1977. *Elements of Software Science*. North Holland: Elsevier Science Ltd.

Hannay, J. E., E. Arisholm, H. Engvik and D. I. K. Sjøberg. 2010. Personality and pair programming. *IEEE Transactions on Software Engineering* 36:61–80.

Höfer, A. 2008. Video analysis of pair programming. *Proceedings of the 2008 International Workshop on Scrutinizing Agile Practices*:37–41.

Nagappan, N., B. Murphy and V. Basili. 2008. The influence of organizational structure on software quality: an empirical case study. *Proceedings of the 30th International Conference on Software Engineering*:521–530.

Nosek, J. T. 1998. The case for collaborative programming. *Communications of the ACM* 41:105–108.

Oram, A., and G. Wilson. 2011. *Making Software: What Really Works and Why We Believe It*. Sebastopol, CA: O'Reilly Media.

Schryen, G., and E. Rich. 2010. Increasing software security through open source or closed source development? Empirics suggest that we have asked the wrong question. *Proceedings of the 43rd Hawaii International Conference on System Sciences*:1–10.

For relevant Web links, consult this issue of *American Scientist Online:*

http://www.americanscientist.org/issues/id.93/past.aspx