Phillip G. Armour

The Five Orders of Ignorance

Viewing software development as knowledge acquisition and ignorance reduction.

In my first column (Aug. 2000, p. 19), I argued that software is not a product, but rather a medium for the storage of knowledge. In fact, it is the fifth such medium that has existed since the beginning of time. The other knowledge storage media being, in historical order: DNA, brains, hardware, and books. The reason software has become the storage medium of choice is that knowledge in software has been made *active*. It has escaped the confinement and volatility of knowledge in brains; it avoids the passivity of knowledge in books; it has the flexibility and speed of change missing from knowledge in DNA or hardware.

If software is not a product, then what is the product of our efforts to produce software? It is the knowledge contained in the software. It's rather easy to produce software. It's much more difficult to produce software that works, because we have to understand the meaning of "works." It's easy to produce simple software because it doesn't contain much knowledge. Software is easier to produce using an application generator, because much of the knowledge is already stored in the application generator. Software is easy to produce if I've already produced this type of system before, because I have already obtained the necessary knowledge.

So, the hard part of building systems is not building them, it's knowing what to build—it's in acquiring the necessary knowledge. This leads us to another observation: if software is not a product but a medium for storing knowledge, then software development is not a product-producing activity, it is a knowledge-acquiring activity.

Hacking

It is quite easy to show that software development is a knowledge-acquisition activity using a slightly exaggerated example. Imagine a hacking project. With hacking, there is no real attempt to acquire the knowledge first, the project just hacks code. As the code is written and executed (testing may be too strong a word), there comes a point where validity of the knowledge in the code is checked somehow. This accomplishes two things: it identifies what in the code is "correct" (the knowledge) and what in the code is "incorrect" (what I call "unknowledge"). This unknowledge is often—somewhat incorrectly—considered to be defects. From a knowledge perspective, "unknowledge" is still knowledge; it just doesn't

PAUL ZWOLAK

The problem arises when we think the code, rather than the knowledge in the code, is the product.

apply to this particular system.

Coding continues by stripping out the "unknowledge" code and building on the "knowledge" code. This continues until the next validation point. This whole activity repeats until the system is built. Note that the activity of coding is simply the *mechanism* that is being used to capture the knowledge (and unknowledge). Most time is actually spent in deriving these two forms of knowledge, and then identifying and separating them.

At the end of this hacking activity, having written a lot of code, we are left mostly with "knowledge" code. One could legitimately argue that if this system can successfully pass all tests we can throw at it, it does, *ipso facto*, contain all the necessary knowledge. However, there are a few further observations we can make:

• Unless great care is taken to remove all traces of the "unknowledge" from the code, some legacy will remain. That is, the program will be contaminated by the footprints of the "unknowledge." These footprints will be extra states, switches, declarations, loops, and so forth, that supported the incorrect assumptions, but were not fully removed. The code may work, but it's not "good" code. So, while the final product does contain the necessary knowledge, it also contains the remains of the journey to find that knowledge.

• While we are acquiring two different kinds of knowledge (what works and what doesn't), we are only saving one kind. The "what doesn't" is simply thrown away.

• The hacking approach does not work well if there is a significant likelihood of later knowledge invalidating earlier knowledge. When this happens, there will be enormous amounts of backtracking in order to rework the system.

The problem is that the final product is contaminated with the legacy of the process used to build it. Perhaps the developer knows this and understands what should have been redesigned. But no one else does, and in a year's time, the developer will have forgotten why it looks the way it does. For this reason, code is a *write-only* knowledge store.

It's evident from this example that the real job is not writing the code, or even building the system—it is acquiring the necessary knowledge to build the system. When hacking, we use the activity of building the system (or rather attempting to build the system) as our mechanism for understanding what the system has to do. Code is simply a by-product of this activity. The problem arises when we think the code, rather than the knowledge in the code, *is* the product. Then we are tempted to ship the code as is. What we should do, of course, is rewrite the code so it cleanly represents the knowledge *after the hacking stage*. If we have done a good job of capturing what we have learned by hacking the code, writing it again should be straightforward and rather quick. The act of doing this intentionally is called *prototyping*.

As a development life-cycle model, prototyping acknowledges that our job is not to build a system, but to acquire knowledge. We don't expect to get a functioning system the first time out when prototyping. What we do expect to get is (some of) the knowledge needed to build the system. And we use prototyping particularly when we don't know in advance what kind of knowledge we might need or there is a likelihood of later knowledge modifying earlier knowledge.

So if our job is to acquire knowledge, what can we assert about the knowledge we must gain? For everything we know, we also have a certain amount of ignorance. Ignorance being simply the other side of the knowledge coin. If we view systems development as the acquisition of knowledge, we can also view it as the reduction or elimination of ignorance. We would hope that, at the end of the project, we are less ignorant than we are at the start. So what kinds of ignorance might we exhibit?

The Five Orders of Ignorance

Based upon what we know and what we don't know, we can classify our ignorance into strata or layers. These I call the "Five Orders of Ignorance." They can be

helpful in understanding what is needed to reduce our ignorance and build a system that works. They also help explain some of the artifacts of the software development environment, and some of our behaviors working in this environment.

Since we are computer folk, we start counting from zero, rather than one. And so we'll apply this to the Orders of Ignorance:

0th Order Ignorance (0OI)—Lack of Ignorance. I have 0OI when I know something and can demonstrate my lack of ignorance in some tangible form, such as by building a system that satisfies the user. 0OI is knowledge. As an example, since it has been a hobby of mine for many years, I have 0OI about the activity of sailing, which, given a lake and a boat, is easily verified.

1st Order Ignorance (1OI)—Lack of Knowledge. I have 1OI when I don't know something and can readily identify that fact. 1OI is basic ignorance. Example: I do not know how to speak the Russian language—a deficiency I could readily remedy by taking lessons, reading books, listening to the appropriate audiotapes, or moving to Russia for an extended period of time.

2nd Order Ignorance (2OI)—Lack of Awareness. I have 2OI when I don't know that I don't know something. That is to say, not only am I ignorant of something (for instance I have 1OI), I am unaware of this fact. I don't know enough to know that I don't know enough. Example: I cannot give a good example of 2OI (of course).

3rd Order Ignorance (3OI)—Lack of Process. I have 3OI when I don't know a suitably efficient way to find out I don't know that I don't know something. This is lack of process, and it presents me with a major problem: If I have 3OI, I don't know of a way to find out there are things I don't know that I don't know. Therefore, I can't change those things I don't know that I don't know into either things that I know, or at least things I know that I don't know, as a step toward converting the things I know that I don't know into things I know. For system development, the "suitably efficient" proviso must be added, since there is always a default 3OI process available: try and build the system. Whereupon the customer can be relied on to inform me of all the things I did not know.

4th Order Ignorance (4OI)—Meta Ignorance. I have 4OI when I don't know about the Five Orders of Ignorance. I no longer have this kind of ignorance, and now, neither, dear reader, do you.

The Five Orders of Ignorance in System Development

Each of the Five Orders of Ignorance plays a significant role in building systems:

- *0OI*: Since 0OI is knowledge, this is the correctly functioning element of the system that I understood and successfully incorporated into the system. When I have 0OI, I have the *answer* to the problem.
- *1OI*: These are the known variables, where the presence of the variables is known, but not their values. When I have 1OI, I have the *question*. Usually, having a good question makes it fairly easy to find the answer.
- *2OI*: This is the real problem. Not only do I not have the answer I need, I don't even have the question. This is where we start many projects. When we begin projects, we know, from experience there are many things we have to learn. We just don't know what they are. 2OI explains, for instance, most variation in project estimates and the concept of "contingency" (to allow for things we haven't thought of). 2OI also explains the famous "90% complete program syndrome," where a programmer asserts with conviction that he or she is 90% complete, sometimes for months on end. The programmer is not "lying," but certainly is not correct. Basically the programmer doesn't know how complete he or she is. Why? Because of 2OI, the programmer doesn't know what the programmer doesn't know. 2OI also accounts for rework cycles, late-phase "gotchas," and what project manager and author Fred Brooks calls "second system effect."
- *3OI*: Coupled with 2OI, 3OI presents a real danger (I don't have a way to resolve my lack of knowledge during the available time I have). Personally, I think all software development methodologies are actually 3OI processes, whose main job is to show the areas of the product or process, or where I lack knowledge. There is an important thing to note—the answer I am looking for *cannot* be

in the methodology. With few exceptions, methodology simply gives me a syntax in which to frame the question and a discipline for identifying those areas where I might have 2OI. But it can't know what I'm trying to do. The answer must come from elsewhere.

• *4OI*: This is probably not too much of an issue, though I've found thinking of process this way helps. We've found ways to compensate for our orders of ignorance, like the use of contingency for 2OI. I added 4OI to this model mostly because knowledge is inherently recursive.

The critical levels seem to be 2OI and 3OI. I view most of our work to be the reduction of 2OI, and the development and use of all software and systems methodologies as being 3OI processes. The job of a 3OI process is to illuminate our 2OI. The application of 3OI to 2OI generates either 1OI or more rarely 0OI—the process either gives us the answer (0OI) or more commonly, it gives us the question (1OI). This is one of the major purposes of process and is the major role of methodologies and modeling.

The critical point here is that the application of 3OI processes (methodologies and process) does *not* give the answer; it gives me the question. As a business model, we have been looking to these tools for the wrong thing. We expect them to provide us with answers, and that's not what they do. It is very frustrating to expect an answer and instead get a question; to use a methodology to reduce the amount of work, and apparently increase it; or to conscientiously apply a process only to have it tell us just how little we actually know. But that is the reality of the software development business. A functioning system is the by-product of the activity of finding things out. The working system is the *proof* that I have the knowledge. Looked at pragmatically, the goal is to resolve our orders of ignorance to 0OI. We spend most of our energies acquiring knowledge. Since finding the answer to 1OI is straightforward, we must be spending most of our production energy on 2OI, and most of our process energy on 3OI. We can use the Orders of Ignorance to categorize what we know, and what we don't know, to estimate the likelihood of what we don't know we don't know, and to assign to process and methodology their true place in the order of things. C

Phil Armour (armour@corvusintl.com) is a vice president and senior consultant at Corvus International Inc, Deer Park, IL.