Phillip G. Armour

The Laws of Software Process

Getting from here to there: Putting a boundary around process.

"Just look at this," Chris said, dropping a heavy three-ring binder onto the desk. It made a resounding thud. "This is our new software development process."

"Looks kinda big," Ethel said, thumbing through the several hundred pages. "Do we need all this?"

"Now this is what I call documentation," Chris cracked. "Do they honestly think people will *read* it, let alone use it?"

IN SOME CIRCLES, SOFTWARE process is considered to be *the* issue that needs to be resolved to fix "the software crisis." Improving process has become an article of faith in some corners, while avoiding it has assumed the status of guerrilla warfare in others.

Why is this? Why have some companies allocated enormous resources to defining a process for the construction of software, while all too often the supposed users of the process—the developers themselves—pay lip service to it, or shun it altogether? Is the process bad? Is the process we use to define the process flawed? Should we not have process at all? Or should we have more?

MICHAEL SCHRÖTER

Well, maybe both. But perhaps we should take a look at what process is used for. Perhaps our problem isn't process, it's what we are asking process to do, and when and where we apply it.

First, we should note that the kinds of knowledge that must be gained will vary from system to system. The degree to which the knowledge is unknown will pretty much determine how long it takes us to acquire it. These degrees of "unknownness" are called the "Five Orders of Ignorance" (see the "Business of Software," Oct. 2000, p. 17, for a complete discussion), and they greatly affect the kind of process we can use. They are:

0th Order Ignorance (0OI)—Lack of Ignorance. I have 0OI when I (provably) know something. With 0OI we have the answer in a provable form. When projects already have the answers, they can have highly explicit and well-controlled processes to implement those answers. An example of this could be well-designed Configuration Management (CM) procedures for code control.

1st Order Ignorance (1OI)—Lack of Knowledge. I have 1OI when I don't know something. With 1OI we have the question in a well-factored form. These projects can use a fairly explicit process for getting the questions answered. An example might be developing a standard accounting system—we may not know exactly what tax codes to use, but we do know that we need to know them.

2nd Order Ignorance (2OI)—

In some circles, software process is considered to be the issue that needs to be resolved to fix "the software crisis."

Lack of Awareness. I have 2OI when I don't know that I don't know something. With 2OI, we don't even have the question. Such projects *cannot* use a well-defined process, because we don't know what process might work.

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. Projects with 3OI are wrestling with basic process and life cycle issues. Not only does the project not have the answer to its system questions, it does not have an assurance of a process to get the answer. Such projects may not even have a viable metaprocess.

4th Order Ignorance (4OI)—Metaignorance. I have 4OI when I don't know about the Five Orders of Ignorance. Processes for projects with 4OI do not concern themselves with the storage and retrieval of project knowledge, since they don't recognize that this is at all important.

The challenge is all projects have different quantities of 0OI, 1OI, 2OI, and even 3OI, and therefore require different types of processes. Porting an existing business application across platforms is mostly 0OI and 1OI, and the process for this can be well-defined, perhaps even automated. Research projects are usually heavy in 2OI problems, and process cannot be well-defined, or rigorously implemented because we don't know what kind of process might work. In every system development there is some measure of 2OI, and we must deal with this in a different way than 1OI. This means we have to adopt a different process for each kind of unknown.

A few years ago, I formulated a set of "laws" for software process. The occasion called for some humor, but as with a lot of humor, there is an underlying vein of seriousness. There are three laws and a number of associated observations.

The First Law of Software Process

Process only allows us to do things we already know how to do.

The corollary to the First Law of Software Process. You can't have a process for something you don't know how to do or have never done.

Explanation: What the first law is saying is that we can only define processes to the extent that we know what to define. In Orders of Ignorance terms, we can only define detailed processes for 0OI and 1OI. For 2OI, we can only define metaprocesses. Detailed processes are useful only in known situations (0OI and 1OI); the applicability of detailed process for 2OI and 3OI situations must be limited, and may be restrictive.

The lemma of eternal lateness. The only processes we can use on a project were defined on previous projects, which were different.

Explanation: This is a restatement of author Fred Brooks' Second System Effect. We develop processes on earlier systems that are not like the current system to some extent. So, to the same extent, the processes will not apply. This extent is determined by the differences between the previous projects and the current one. The degree of 2OI is determined by the same thing. This is not a coincidence.

The Second Law of Software Process

We can only define software processes at two levels: too vague and too confining.

Explanation: This is a consequence of having to deal with both 0/1OI and 2OI. Processes always tend to be too vague for those things we know how to do (the process should tell us exactly what to do, or even do it for us) and too specific for those things that the process doesn't fit (which are usually 2OI problems we haven't encountered before, which is why they are 2OI problems).

The rule of bifurcation. Software process rules should be stated in terms of two levels: a general statement of the rule, and a specific detailed example (see also the Second Law of Software Process).

Explanation: The idea is that the general rule can deal with the high level (the context), and the example with the low level (the application). Particularly for engineers, it's a good idea to back up a general-rule statement with an example

explaining the use of the rule. For example, engineers might not know how to use this rule unless I included this as an example.

Armour's observation on software process. What all software developers really want is a rigorous, ironclad, concrete, hidebound, absolute, total, definitive, and complete set of process rules they can break.

Explanation: This is a statement of the need for both rigor and flexibility. For those things that are well defined (0/1OI), we do need and can obtain a precise definition. Precise definition for those things that the process doesn't fit (2OI/3OI) is usually very ineffective; the process won't work and must be modified. Developers want and need both.

The Third Law of Software Process

The last type of knowledge to consider as a candidate for implementation into an executable software form will be the knowledge of how to implement knowledge into an executable software form.

Explanation: For a number of reasons, the target medium of most process efforts seems to be paper rather than software. This is, to say the least, ironic.

The twin goals of optimal termination. 1) The only natural goal of a software process group should be to put itself out of business as soon as possible, and 2) The end result of the continuous application of effective process will be that nobody actually has to use it.

Explanation: A quality group should fold its results back into the manufacturing line to develop a high-quality process that doesn't need a quality group. Equally, a software process group should encapsulate the known process (say, into a software medium) and the mechanism for changing the process into the development activity in a way that doesn't actually require a process group.

The second goal alludes to the fact that, since process can only be defined for well-understood activities, the process can and should be made so mechanical that it can be automated and won't need the intervention of developers. This frees the developers to work on the things they don't know how to do, for which there is no process. That is to say we automate what we do know, and discover what we don't know. This is what developers have wanted all along.

Applying the Laws of Software Process

1. Since well-defined process only works for well-defined situations, we need to understand the inherent limitations of process and not expect it to do things it cannot do (like deal with wholly new situations). [1st Law]

2. Separate process targets into definable and less-definable. For example, a CM code check-in procedure should be highly definable and preferably automated, while a usability lab research project might be much less definable. [1st Law]

3. Stop attempting to develop monolithic processes that try to define every activity at every level. [2nd Law]

4. Build into processes dealing with 2OI a "creative space"—a sandbox where developers can play, to allow them to come up with new ideas. [1st Law, 2nd Law, Armour's Observation]

5. Build "process labs" for projects and organizations that deal with 2OI and 3OI. Don't forget, these will need a really big creative space. [1st Law, 2nd Law]

6. Lock down the well-defined tactical activities we know a lot about, and for which there is little value in reinventing. [1st Law, 2nd Law]

7. Build and use systems that capture knowledge as it is discovered and make that knowledge available to others in a usable form. [3rd Law]

8. Have term limits for process groups. [Twin Goals]

9. Apply effectiveness and value metrics for process—only implement rigorous process where consistency and repetition are valuable (0OI, 1OI), not where novelty and inventiveness are needed (2OI, 3OI). [1st Law]

10. Automate, automate, automate (but only where we know the process works). [3rd Law, Twin Goals]

THE JOB OF PROCESS IS TO ensure that, starting from *here*, we can guarantee we end up over *there*. Which is fine, but only if we know where we're starting from and where we need to end up. C

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