Armour, P. 2004. The Laws of Software Process: A New Model for the Production and Management of Software. Auerbach Publications.

Chapter 1
The Nature of Software and the Laws of Software Process

Pardon him, Theodotus: he is a barbarian and thinks that the customs of his tribe and island are the laws of nature.

— George Bernard Shaw
Caesar and Cleopatra

Software is not a product.

Behind this simple sentence lies a whole world of behavior, radically changed business models, a fundamentally different economic reality, and a view of development methodology and software process that is quite different from the way we practice the business of software today. Today, most companies view software as a product. But it is not. Software is a medium.

A Brief History of Knowledge

About 2 billion years ago, approximately half the time our planet has been in existence, nature developed a knowledge storage mechanism that allowed species to learn and to store that knowledge in a way that could be passed on to their descendants. Instinctual knowledge is stored and transmitted in strands of deoxyribonucleic acid (DNA). DNA (see Exhibit 1) was the first knowledge-storage medium.

Between 8 and 5 million years ago, it is widely believed, the first recognizable ancestors of the human race evolved. These proto-humans possessed a concentration of nerve cells that was quantifiably different from animals around them. It allowed them to change their behavior and adapt to different situations and environments, to learn and relearn. The brain (Exhibit 2) was the second knowledge-storage medium.

Louis Leakey called early mankind “Homo Habilis” or “tool maker.” The primitive tools he discovered in the Olduvai Gorge in Tanzania were made

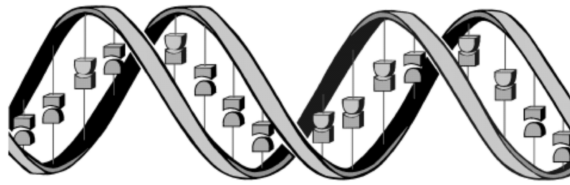

Exhibit 1. DNA: The first knowledge medium.

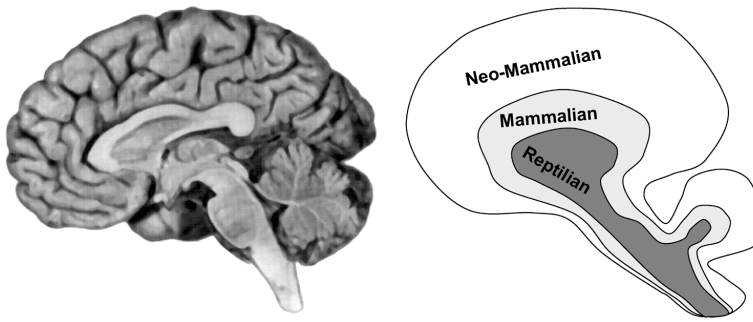

Exhibit 2. The brain: The second knowledge medium. © Wellcome Department of Imaging Neuroscience, University College, London, England. With permission.

of rocks but were more than rocks. Their makers had crafted the rocks by first selecting the types of stone with the properties needed to make cutting instruments (see Exhibit 3). They then flaked them and modified them for the particular task for which they were used. The value of the rock as a tool came mostly from the skill or knowledge of the craftsman creating it. We do not usually think of hardware as being "knowledge storage devices" but it is. The value of any tool is a direct function of the quality and quantity of knowledge that went into its making and its fitness for the use we make of it. Hardware was the third knowledge-storage medium.

The first recognizable books appeared around 3500 B.C.E. in the Middle East and were created using clay tablets. Around 2500 B.C.E., people in Western Asia wrote on animal skins and Egyptians started using papyrus to record their thoughts, instructions, transactions, and laws. But it was not until about 600 years ago with the invention of moveable type that books really started to be used in society to record, store, transmit, and reuse knowledge. Books were the fourth knowledge-storage medium.

After World War II, a number of scientists collaborated to create the earliest computers. John von Neumann is generally credited with the first statement of the concept of the stored program in 1945. The knowledge

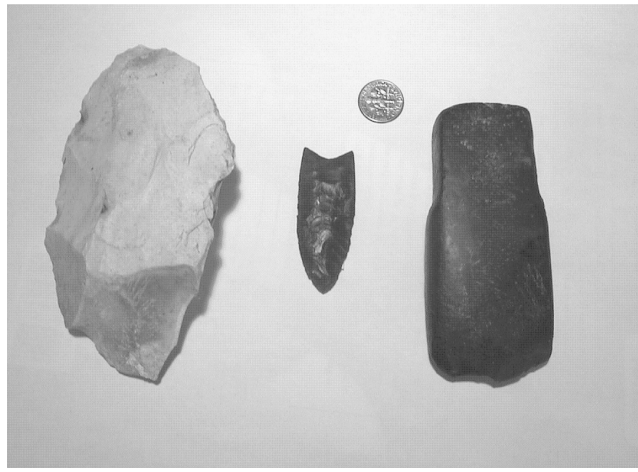

Exhibit 3. Hand axes: Hardware, the third knowledge medium.

Exhibit 4. Comparison of properties of knowledge media.

Storage medium	Persistency in medium	Update frequency	Intentionality	Ability to self-modify	Ability to modify the outside world
DNA	Very persistent	Very slow	Low	Moderate	Quite limited
Brain	Very volatile	Very fast	High	High	Quite limited
Hardware design	Very persistent	Slow	High	Low	Limited to specific design
Books	Quite persistent	Quite slow	High	None	None
Software	Quite persistent	Fast	Quite high	High	Relatively unlimited

that goes into a computer program has the distinct characteristic that it has been made executable. Software is the fifth knowledge-storage medium.

The Characteristics of Knowledge-Storage Media

If we look at the characteristics of knowledge stored in each of the media shown in Exhibit 4 and Exhibit 5, we can see why software is becoming so universal.

From these charts we can see that knowledge stored in software has a wider range of useful characteristics than knowledge stored in other media. It is more intentional in its knowledge storage than DNA, has a

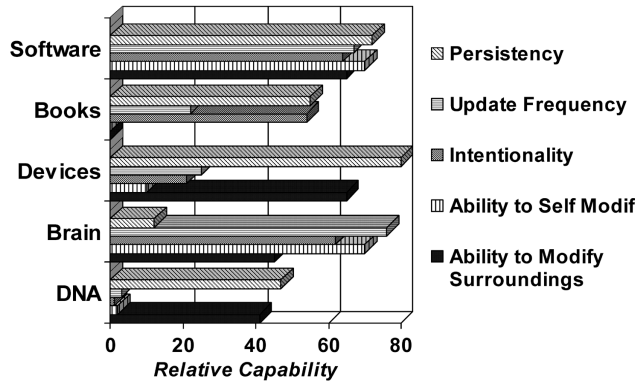

Exhibit 5. Relative capabilities of knowledge media.

greater capacity to modify itself or be modified than knowledge stored in hardware, is more persistent than knowledge stored in brains, and is infinitely more able to interact with its surroundings than knowledge in books. It is for these reasons that all of human knowledge is currently being transcribed into software.

If software is not a product, then what product do we create when we build software systems? The answer is simple — the product is not the software, it is the knowledge contained in the software.

The Nature of Software Development

If software is not a product, then software development is not a product producing activity — it cannot be. If the true product is not the software but the knowledge contained in the software, then software development can only be a knowledge acquisition activity. True, we may consider some of the software development function to be a transcription of knowledge into the executable form. This is what coding is. However, coding is only one small part of the software development activity, and it is getting smaller. We can also reasonably assert that if we already have the knowledge, then transcribing it does not require much effort. The real work occurs when we try to transcribe our knowledge and find it is incomplete. The activity then quickly changes into a search for the correct knowledge.

But software development is not usually viewed or managed as a knowledge acquisition activity. Most of our processes and our expectations in the software business are centered on the creation of a product (the delivered system). This is a quite logical view but it is also quite wrong. The creation of a functional system is not the product of the true activity of software development; it is the by-product of the activity of learning.

This can be seen using a negative example: it is quite easy to deliver a system to the customer. The challenge is not to create a system but to create the correct system. If we know exactly what it is we have to do, we can create systems as fast as our fingers can type. The real challenge is finding out what the system should do, and how we can construct a system that does it. Once we have acquired this knowledge, the creation of the system is usually quite straightforward. Of course, once we (think we) have acquired the appropriate knowledge, and go ahead and build and test the system, we invariably find out there are things we did not know. This, of course, requires further knowledge acquisition.

We can see this very clearly in the activity of testing systems. Testing of systems usually takes very little time — if we do not find out anything new. It is only in the uncovering of new knowledge that we take time. The purpose of testing is only partially to prove what we know. In reality, most of our time is spent finding out things we did not know.

The Laws of Software Process

In the 1980s, I participated in two separate "process definition" activities at the major airline where I was then employed. For the first endeavor about a dozen of us were locked up in the company's training center, chartered with the task of attempting to define just how the computer department would create software. After several months of diligent labor, we proudly unveiled our process definition.

It was very big.

Occupying three large ring-binders, it described, sometimes in exquisite detail, exactly all the steps we thought necessary to create functional software.

The binders were duly shipped to the designated recipients — every manager and team leader in the installation. They promptly and strategically positioned the unopened binders on their bookshelves where they sat gathering dust, until a couple of years later when I participated in another process definition exercise. The first process was generally not used because it was felt it was too heavy, too cumbersome, too restrictive, and too "big." Our mandate for the second attempt was to "cut it down" — to provide more a set of guidelines than a restrictive recipe, to give a framework rather than a prescription, to direct rather than constrain but mainly to make it smaller. This much-faster exercise resulted in a slim volume being produced just a few weeks later. It was mailed out to the designated recipients, only to suffer the same fate as the first edition. This time the criticism was that the guidelines were "too vague," and that the process did not really add anything that was not already known, that it did not have enough "meat."

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 organizationally imposed process has assumed the status of guerilla warfare in others. Process models and methodologies duel with each other across message boards, magazines, and bookshelves. People are assessed and undergo extensive certification to allow them in turn to assess and certify others in process frameworks, while others espouse rapid "get-it-done-now" approaches as well as process so lightweight it barely exists.

Why is this? Why have some companies, institutions, and individuals allocated such significant 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, severely modify parts of it, or simply shun it altogether? Are the processes badly defined? Is the process we use to define the process flawed? Should we not attempt to define process at all, and let the developers make it up as they go along? Or should we try to define process to greater levels of precision?

Maybe the answer is both. But first, perhaps, we should take a look at the purpose of process — what do we expect process to do for us? Where is the value added by applying process? It might be that our problem is not process, it is what we are asking process to do, and when, where, and how we apply it.

Software is knowledge made active. The difficult and time-consuming activity in creating software is not one of transcribing our already-available knowledge into the active form; it is in effectively acquiring this knowledge in the first place. Even more specifically, it is in finding the essential knowledge necessary to make the system work that we do not know we are missing.

By "effectively acquiring" I mean obtaining all the necessary knowledge and then structuring it so that it is consistent, complete, and usable. "Consistent" means that the knowledge does not "contradict" itself (including being consistent with the transcription activity and language, of course). "Complete" means that all the necessary knowledge required for correct executability is obtained, and "usable" means that the knowledge is rendered into a form that we can easily process to create the system.

Software is a knowledge-storage medium and the development of software is a knowledge-acquiring activity. But most companies do not manage the creation of software as a knowledge-acquisition activity. The lack of focus on knowledge acquisition and management is the root cause of many of the problems we perennially experience in the creation of software. To take the argument a step further we can, in fact, make a case that our basic business model for the production of software is simply wrong.

Starting from these premises:

- Software is a knowledge medium.
- The "product" is the knowledge contained in the software.
- The activity of developing software is the activity of acquiring specific types of knowledge and translating that knowledge into a specific language form known as "code."

First, we have to look at the source: what kinds and quantities of knowledge need to be gained? These will vary enormously from project to project, environment to environment, and system to system. There is knowledge of what the customer requires, usually known as "requirements." There is knowledge of which system and module designs would work best to deliver the functionality to the customer; and there is knowledge of how to transcribe the knowledge into executable form ("coding" knowledge). Of course, there are many, many finer distinctions of these knowledge forms: the format of the customer interface, the persistent data storage needs of the system, data security, operational states, and error recovery. The list goes on and on.

However, even without pausing to categorize the knowledge needed, we can abstract characteristics of all knowledge into the degrees of "unknownness" that I call the "Five Orders of Ignorance":

- *Zeroth Order Ignorance (0OI):* Lack of ignorance. I have 0OI when I provably know something.
- *First Order Ignorance (1OI):* Lack of knowledge. I have 1OI when I do not know something.
- *Second Order Ignorance (2OI):* Lack of awareness. I have 2OI when I do not know that I do not know something (see Exhibit 6).
- *Third Order Ignorance (3OI):* Lack of Process. I have 3OI when I do not know of a suitably efficient way to find out that I do not know that I do not know something.
- *Fourth Order Ignorance (4OI):* Meta ignorance. I have 4OI when I do not know about the Five Orders of Ignorance.

Each of the Orders of Ignorance requires a different type of process. Zeroth Order Ignorance (0OI), the application of already-known information, requires a stable, detailed, and highly repeatable process — basically a streamlined version of the same process that was used when the knowledge was first successfully implemented after being obtained. On the other hand, Third Order Ignorance (3OI) requires a very different kind of process. With 3OI, not only do we not have the necessary knowledge, we do not know exactly what it is that we need to find out, and we are wrestling with defining and implementing a process of any sort that will help us find out what we need to know.

Exhibit 6. The effects of 2OI.

Inaccurate Estimates

When projects are started, an estimation of effort, schedule, and cost that is used to define the project expectations and commitment is usually prepared. Experienced developers know that their best efforts at estimating the effort are always less than the task turns out to require. Why? Because of Second Order of Ignorance, we can only estimate based on what we know plus what we know we need to know (0OI and 1OI). It is not possible to estimate based on what we do not know we do not know any more than a scientist can design an experiment for something for which she is not looking. The usual coping mechanism for this is to add "contingency." This is often a simple percentage — for example, 35 percent — that is added to the calculated effort or schedule. The contingency shows for the things that we do not know we do not know. Note that we typically assign a much-higher contingency for really new projects than we would for projects that are routine and we have successfully completed before. This is tacitly acknowledging that such projects are higher in 2OI. In fact, it is reasonable to assert that almost all of the effort on a project is the result of 2OI (with some 3OI).

Our usual compensating mechanism for 2OI is the addition of contingency resources and time for those things that we know will go wrong but which we cannot actually identify.

The 90-Percent-Complete Syndrome

This occurs when a programmer (or tester or whomever) maintains, sometimes for months, that despite diligent effort he remains at "90-percent complete." Basically, the programmer does not actually know how complete he is. Why? Second order ignorance is always high when we are engaged in a discovery activity. The 90 percent complete syndrome occurs when we believe we are performing a product-production activity. The "completeness" question is actually misplaced because it cannot be assessed realistically beyond a finite degree of accuracy. The question should be, of course, "how much have we learned that is valuable?"

The Five Orders of Ignorance

Because the nature of these "levels of ignorance" is quite different, we need to establish different process criteria for each Order of Ignorance:

- *0OI:* I have the answer. Developing a system is simply a matter of transcribing what I already know, into the appropriate programming instructions. Note that this is not possible unless I know the appropriate coding medium — that is, I know how to program. Of course, not knowing the language, not knowing how to program, is actually a form of ignorance, which I would presumably know about. If that were the case, I would classify it as a form of 1OI to be dealt with below.
- *1OI:* I have the question, and I either know how to get the answer to it (I have 0OI about the activity of answer acquisition) or I do not know how to get the answer (I have 1OI about answer acquisition).

In the first case, I simply go and get the answer. In the second case, I have two questions: how do I get the answer? And then what is the answer? In the event that there is no way to get the answer, I may have to ask a different question and look for a different answer, such as "what do I do when I cannot get an answer?" Both the question and the answers can have levels of "granularity." We can classify very high-level, open questions (such as, "What are the requirements for this system?") as being type-questions. They are the typical questions posed by meta processes (such as a process defined as Gather Requirements). They are generic, vague, context-free, and domain nonspecific — they could apply to any domain or a wide range of domains. They may also not be very useful in some situations because they will often generate vague (context-free) answers. Type-questions occur when the question knowledge content is nonspecific and is simply of a type of knowledge. For instance, I could ask the question, "What are the requirements for this system?" I know that all systems have some requirements; it is simply a type of knowledge I need to get. The format of the question is very nonspecific and could generate a wide range of responses. I might get generalized function information ("this system must process credit card transactions") or I might get detailed implementation information ("This system must be written in Z8000 assembly language"). While such questions are helpful when I do not have a context to ask specific questions, they do not usually produce usable engineering answers. The most useful output from a type-question is often another more-specific question. If I find I am using many type-questions, it usually indicates that I have significant amounts of 2OI.

- *2OI:* I do not have the question. I do not know enough to frame a question that is contextual enough to elicit a definitive answer. I may fall back on type-questions or some other mechanism. This fall-back mechanism is a "Third Order Ignorance Process." I will argue that most, if not all, software methods are 3OI processes. Some types of systems are predominantly 0OI and 1OI heavy and others have large amounts of 2OI. A research project, for example, is often heavy in 2OI problems. Tactical systems development, such as porting, which has been successfully completed before, tends to emphasize 0OI and 1OI — the issues and questions are quite well known and answers are readily forthcoming. But in any systems development there is always 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 and different processes for 0OI/1OI and 2OI/3OI.
- *3OI:* 3OI is the process level. As there are levels of granularity of questions at the 1OI level, there are levels of granularity of processes at 3OI. Detailed, granular processes are the most useful, although

they are usually only useful across restricted problem spaces. The reason they are useful is that the processes contain the context of the problem. I call high-level, general processes that do not contain the context meta processes. They are usually not very useful in obtaining definitive answers and instead tend to generate more questions. The highest level of a waterfall life cycle model is an example with its phases of: Requirements, Design, Code, and Test. While undoubtedly "correct," this tends not to be a very useful process. We have to get requirements for every system; ditto design and code and test. The process at the life cycle level does not tell us exactly what to do, and when and where to do it. It does not tell us what kind of requirements to get and from whom. This is because generically we do not know what to do that would be effective; because we do not know what questions we are trying to answer, until we start operating on this system. This limitation is inherent to all processes.

- *4OI:* 4OI is the understanding that software development is a knowledge-acquisition activity. The global application of the same methodology regardless of what is known or not known of the problem, or what kind of problem it is, could be viewed as 4OI in operation. The squandering of hard-won knowledge assets is another example of this level of ignorance in operation.

There seem to be some basic "laws" or conventions operating in the software area and in the software business that can help us to understand the nature of this activity.

The Laws of Software Process

A few years ago, I was asked to be a keynote address speaker at the end of a five-day conference on software process. The conference organizers had scheduled this address as the final activity of the conference to fill a 4-hour time slot(!). Because I and my presentation would be the only things keeping the attendees from returning home to their loved ones, the occasion called for some humor and audience participation if I did not want to see people sneaking out of the conference as I was speaking. For the occasion, I formulated a set of "laws" for software process and engaged people in working with them interactively.

The presentation was intentionally light and humorous, although with an underlying vein of seriousness. At one level, these "Laws of Software Process" are intended to be "catchy" — humor can be a useful mnemonic device. However, their point and their humor lies in the fact that they are actually very true, and the humor just allows us to appreciate the sometimes-incongruous nature of what we do. The laws are not intended to be rigorous as, for example, a physical law but are more on the lines of "Murphy's Law."

They are accompanied by a collection of related observations on software process.

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 cannot have a process for something you have never done nor do not know how to do.

Explanation and Observations

The First Law is saying is that we can only define processes to the extent that we know what to define. Topics that are extremely well defined — that is, about which we know a great deal — can have extremely well-defined processes. Topics that are vague or even unknown cannot have well-defined processes. In Orders of Ignorance terms, we can only define really detailed processes for 0OI and 1OI. Therefore, detailed processes are useful only in known situations, and the applicability of processes for 2OI must be limited because, by definition, we do not know what the 2OI knowledge is.

A further inference of the First Law is that in general we cannot have well-defined processes for things we have never done. Because every software development must have some components (or combinations of components) we have never created before, we have an immediate restriction on the applicability of any process to software development. Of course, this restriction can be modified in several ways:

- Even the most radically new elements of most development activities tend to be similar to development activities in other projects. We may be able to apply analogistic and metaphoric processes (this is like that) to allow us to move forward.
- The activity of learning may have attributes that are more a function of the action of learning than what is being learned.
- We can "bootstrap" processes from the known to the unknown elements, in a way analogous to a student answering the well-known questions on an exam paper before tackling the unknowns, although this approach has its perils.

The Reflexive Creation of Systems and Processes

1. The only way that effective systems can be created is through the application of effective processes.
2. The only way that effective processes can be created is through the construction of effective systems.

Explanation and Observations

These two statements are so self-referential that they can be neither separated nor disputed. As well as being mutually self-referential (between statements 1 and 2), they are also internally self-referential (within statements 1 and 2). The only logical definition of an "effective" process is that it is able to effectively create systems (although there may be other "effective" attributes). If an effective system is created, then the process used must have been effective, at least to some extent. Underlying the surface paradoxes, which always accompany self-referential statements, is a core truth. Systems development processes cannot be created independently of their purpose, which is to create systems, any more than the activity of reading can be separated from something being read. Having said that, there have been many cases in the history of software development where software development processes have been developed without acknowledgment of the actual necessities of creating systems in the real world. And I have personally participated in the development of systems where it would be hard to point to anything that we could really qualify as a "process." One of the tenets of the "Agile Methods" is most of them have been created by practitioners, rather than theoreticians, practicing their craft.

The general view of software process is that the process is created to allow the development of systems. This is true, and we should not create processes that do not do that. However, a legitimate use of systems development is to create processes that facilitate later development. This is an underlying principle of most process control methodologies: learn from your process to build your product; learn from building your product to improve your process.

The Lemma of Eternal Lateness

The only processes we can use on the current project were defined on previous projects, which were different from this one.

Explanation and Observations

We develop our processes at least one project behind, sometimes on systems that are not quite like the current system. Therefore, always to some extent, the processes will not quite apply. The extent to which the previous processes do not fit the existing situation is determined by the degree of difference between the previous projects and the current one. The degree of 2OI is also determined by the differences between the projects — in fact it usually is the difference. The fact that both govern the applicability of the process is not a coincidence.

The Second Law of Software Process
(see also The Rule of Process Bifurcation)

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

Explanation and Observations

This is a direct consequence of having to deal with both 0OI/1OI and 2OI/3OI at the same time. The anecdote I related concerning creating process for my airline employer illustrates this exactly: the first process definition activity for the airline fell into the "too confining" category. Learning from that, our second effort was promptly deemed "too vague."

Processes always tend to be too vague for those things we know exactly how to do. In fact, not only should the process tell us exactly what to do, ideally it should do it for us. Highly detailed, specific processes are useless or dangerous for those things that the process does not fit. The things that do not fit are usually the 2OI problems we have not encountered before — which is, of course, why they are 2OI problems.

The Rule of Process Bifurcation

Software process rules are often stated in terms of two levels: a general statement of the rule, and a specific detailed example (e.g., The Second Law of Software Process).

Explanation and Observations

This is actually a good approach to use to explain many problems: describe the theory and then give a good example. The theory deals with the high-level conceptualization or general rule, and the example with the low-level implementation or application of the rule. Engineers, in particular, tend to be more comfortable with concrete examples than with general rules. However, in systems development the structure of the systems is usually defined in the general rules from which the exceptions and examples flow. Also, engineers tend to be solution oriented rather than problem oriented. So for engineers, it is a good idea to back up the general rule statement with an example that explains the use of the rule. As an example for the Rule of Bifurcation, engineers might not know how to use the Rule of Bifurcation unless I included this sentence as an example.

The Dual Hypotheses of Knowledge Discovery

- Hypothesis One: We can only "discover" knowledge in an environment that contains that knowledge.
- Hypothesis Two: The only way to assert the validity of any knowledge is to compare it to another source of knowledge.

Explanation and Observations

In some ways, these are a pivot point around which opinions of knowledge rotate. Debating these points has occupied the attention of some of the best thinkers the human race has ever known. From a purely practical point of view, however, they appear to be reasonable working hypotheses. We cannot obtain requirements from a customer who does not know what he or she wants (Hypothesis One). We cannot test a system unless we have a set of test data and expected results against which to measure the system's performance (Hypothesis Two). These hypotheses have some significant practical use in designing projects. For instance, there is no point in engaging in a long and detailed analytical process to derive requirements if no one can be specific about them. What we must do is move the project into an environment where the necessary knowledge will be exposed. We see this, for instance, in the testing of large and popular operating systems. There is no analytical process that Microsoft could use to perform complete environmental testing of its Windows™ operating system. The knowledge of all the different ways people might choose to run Windows™, and all the applications and hardware configurations that people might decide to run it with, simply do not exist in Redmond, Washington. To expose what might happen when Windows™ is run with these various configurations, we have to place Windows in the only environment that contains that knowledge. Specifically, we have to run it in the "real world," because that is the only place the knowledge actually exists.

Armour's Observation on Software Process

What all software developers really want is a rigorous, iron-clad, hidebound, concrete, universal, absolute, total, definitive, and complete set of process rules that they can break.

Explanation and Observations

This is a wry description of the need for both rigor and flexibility. For those things that are or can be well defined (0OI and 1OI) we need and can obtain a precise (rigorous, iron-clad, etc.) process definition. Following such a process will duplicate our earlier successful effort and give us the result we are looking for. However, precise definition for those Second and Third Order Ignorance things that the process does not fit is usually very ineffective, the process does not work, and has to be modified (they can break). The challenge for process definition is that it is intrinsically very difficult and perhaps impossible to clean both sides of the street at the same time — to define a single process at a single level of abstraction that will be both sufficiently rigorous for the known quantities and sufficiently flexible for the unknown quantities.

The Third Law of Software Process (also known as the Footwear Manufacturer's Minor Dependent's Law)

The very last type of knowledge to be considered as a candidate for implementation into an executable software system is the knowledge of how to implement knowledge into an executable software system.

Explanation and Observations

The Third Law targets the software development process. We could plausibly argue that the job of the software developer is to take knowledge from the brain and book media (talk with the customer, read the specification) and convert it into an executable form. For several reasons, which I will discuss later, most process initiatives seem to target the book form of knowledge as the final location in which to deposit an organization's software development knowledge. Additionally, most project managers attempt to hire experienced developers. These developers presumably have some quantity of effective process knowledge stored in their brain medium. This knowledge was deposited by the activity of completing software development projects both successfully and sometimes unsuccessfully. To say the least, it is a little ironic that the target media of software process efforts is usually brains or books, when our job is to make knowledge executable.

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.
2. The end result of the continuous development and application of an effective process will be that nobody actually has to use it.

Explanation and Observations

It is an axiom of the quality movement in particular and process improvement in general that a quality group should fold the results of its efforts back into the manufacturing line. Done effectively, this should result in a high-quality process that does not actually need a separate quality group. In exactly the same way, a "process group" should encapsulate both the known process and the mechanisms for changing the process into the development activity in a way that does not actually require a separate process group. For the unknowns (2OI and 3OI), of course, rigid process cannot be well defined, so most of these processes will reference 0OI and 1OI. The logical result of efforts in this area is to create a highly flexible learning organization that is attuned to the process of learning and discovery.

The second part of the Twin Goals alludes to the fact that, because process can only be defined for well-understood activities, such process can and should be made so automated and mechanical that it does not actually

need the intervention of developers. This frees up the developers to work on the things that we do not know how to do — the Second and Third Order Ignorance activities — for which we cannot have an explicit process. This is exactly the service that most software developers want from process — that the process automatically and effortlessly takes care of the mundane, repetitive, and well-defined activities from which little can be learned, while the developer takes care of the inventive, creative, and new knowledge-discovery activities for which rigorous process cannot be defined or will not be effective.

Summary

The development and application of process as it is generally defined and understood is constrained by some of the realities of the business of creating software. The key to this is that the development of executable knowledge is first and foremost a discovery activity. However, it is associated with certain well-defined translation activities and a certain amount of routine and repetitive work.

We can, and should, rigorously define process for those aspects of our work that we can define rigorously. However, we cannot rigorously define process for those aspects of work that are discovery based. While we can have processes of a general kind for learning and discovery activities, they differ enormously from the repetitive and mechanical actions necessary for true application of the well-known knowledge.

There are several problems we have experienced as an industry in defining process, including:

- Attempting to define a "one-size-fits-all" process or process framework for different kinds of systems where knowledge content and knowledge availability is quite different
- Failing to differentiate between knowledge discovery and knowledge application in process definition
- Attempting to define a single level of abstraction for all process types
- Depositing our process knowledge in book form or leaving it in brain form rather than developing automated systems that assist the knowledge acquisition and application process
- Not properly allowing for the nature of the learning activity in setting up processes, particularly the nature of cognition and problem understanding

The application of the Laws of Software Process and the associated observations should lead us to adopt the following approaches to process:

- Separate activities in software development into known (0OI, 1OI) and unknown (2OI, 3OI) elements, and define different types and granularity of processes for each.

- Stop trying to define monolithic processes.
- Develop processes only from within the context of real systems development to provably solve real problems.
- Recognize the limitations of historically defined processes, and learn from them.
- For processes dealing with 2OI, develop "creative spaces" within the real projects where solutions can be explored.
- On top of the creative spaces, build "process labs" to explore different process options.
- Have term limits and terminating charters for process groups.
- Develop and apply value-added metrics for processes to ensure that following the process actually results in gains over the cost of defining and implementing the processes.
- Build systems that capture knowledge as it is gained, store it in both executable and understandable forms, and make it available to others in a usable form.
- Automate! Automate! Automate! Take as a goal of development the automation, not only of the target system but all varieties of knowledge that are gained as the development progresses.