Appendix B
The Five Orders of Ignorance

He who knows not and knows not that he knows not; he is a fool — shun him!
He who knows not and knows that he knows not; he is simple — teach him!
He who knows and knows not that he knows; he is asleep — wake him!
He who knows and knows that he knows; he is wise — follow him!

— Isabel Lady Burton 1831–1896
Arab Proverb, *"The Life of Captain Sir Richard F. Burton"*

Software is not a product. It is the fifth knowledge storage medium that has existed since the world began. This premise leads us to an interesting question: If software is not a product, then what is the "product" of our efforts to "produce" it? The answer, of course, is that the real product is *the knowledge contained in the software.*

It is rather easy to produce software; dangerously so, in fact. It is much harder to produce software that "works," because before we can produce it, we must understand what "works" means. It is easy to produce software that is simple, because it does not contain much knowledge. It is easier to produce software using an application generator, because the knowledge of how to produce a system (although not necessarily of the system that needs to be produced) is actually stored in the application generation software. It is easy for me to produce software if I have already produced this type of software before, because I must have already obtained the necessary knowledge — assuming I have not forgotten how to create it.

Therefore, the hard part of building systems is not building them, it is in knowing what to build — it is in acquiring the knowledge necessary to build the system.

This leads us to another very important observation:

If software is not a product, it is a medium for storing knowledge; then software development is not a product-producing activity, it is a knowledge-acquiring activity.

It is quite easy to show this using a (slightly exaggerated) example, as shown in Exhibit 1.

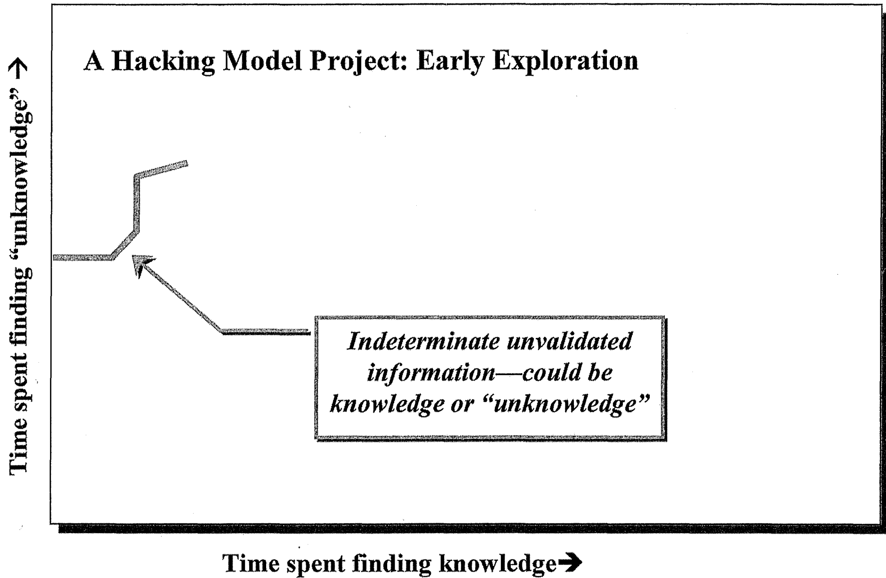

Exhibit 1. Hacking project early exploration.

The activity of hacking (in the software life cycle sense, rather than the other common usage of illegal entry into someone else's computer system) is the writing of code for the purpose of constructing a system whose function is at least somewhat unknown at the outset. While this kind of hacking has a justifiably bad reputation, it is quite common for programmers to use this approach in the small — for simple problems and for systems where the knowledge-to-be-gained is primarily the program steps or control sequence.

In the earliest stages of hacking, we have little or no basis on which to validate the knowledge content of the code — we just write code. The diagram in Exhibit 1 represents a project using this hacking approach to development. The approach could be summarized as *"we have no idea what we're doing, but we'll do it and somehow it'll work."* In a nonrigorous sense, both the X and Y axes in the diagram represent time. X-time is time spent mostly in developing "correct" knowledge, that is, knowledge that will ultimately find its way into the product shipped to the customer. Y-time is time spent mostly in developing "incorrect" knowledge, which is knowledge that is not immediately relevant to the product at hand and will not be, or rather should not be, incorporated into the product. Because hacking (except in the trivial case where we have already built this exact system before, in which case we might reasonably ask why are we doing it again?) is building a system without knowing what it should do, the fact that the

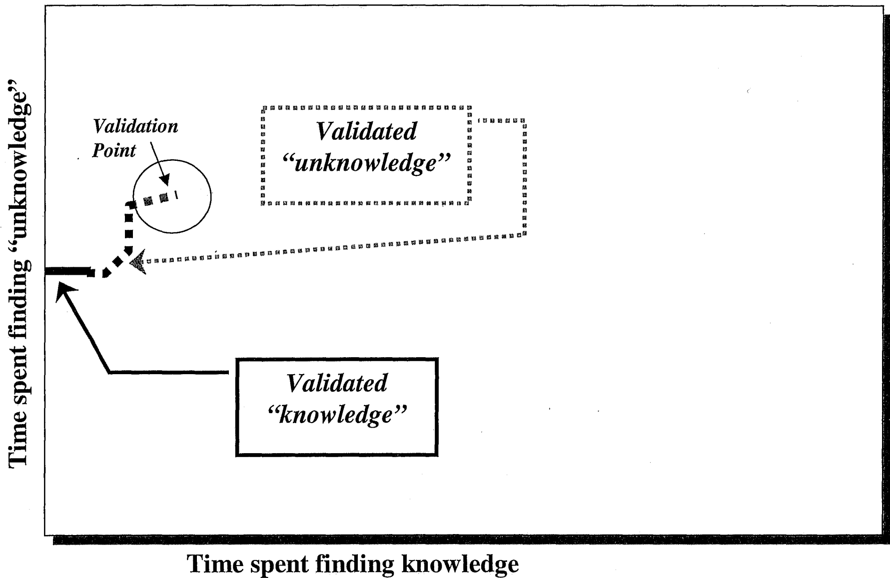

Exhibit 2. Hacking project: validating the knowledge.

path deviates from the straight and narrow is simply caused by the fact that we do not know what to begin with, what that path should be, and where it should go.

Using hacking we usually salvage *some* useful knowledge (shown in the solid line in Exhibit 2), from the coding activity. Much of what we learn, however, is not useful knowledge, at least not for this particular system. This "*unknowledge*" is shown as a dotted line. Generally, it is stripped from the code product, leaving only the solid "useful" knowledge. This strategy continues until the complete set of knowledge is obtained (we hope).

At some point, to determine whether what we have done is knowledge or unknowledge, we have to "validate" the knowledge we have gained and incorporate it into the code artifact. In Exhibit 2 it is called the "Validation Point." With hacking, this usually occurs quite frequently. It also means, incidentally, that we must have access to another source of knowledge about what the system should do; otherwise we must end up comparing the knowledge store against itself.

There are two results from this validation process:

1. We determine what works.
2. We determine what does not work (for this particular system).

The "what works" we leave in the code, the "what does not work" we usually remove.

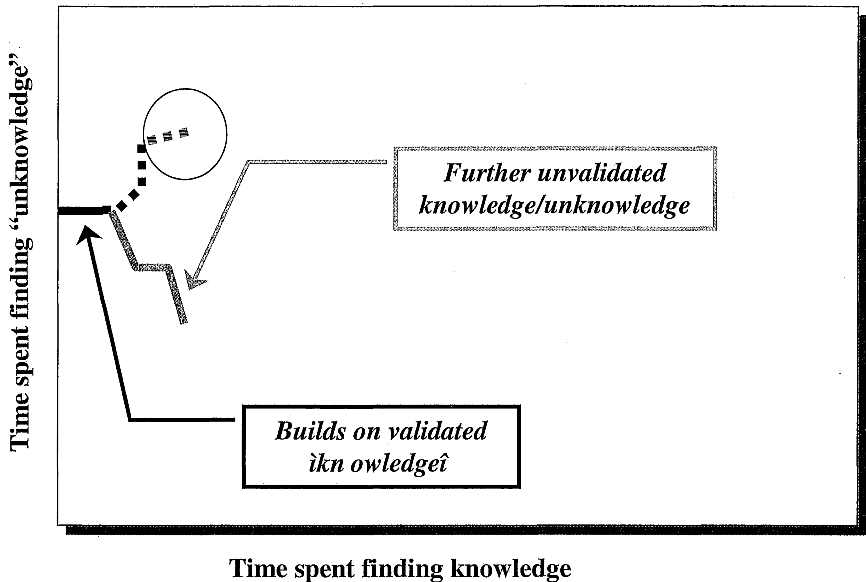

Exhibit 3. Hacking project: building on the knowledge.

Using the residual “what works” knowledge, we commence further exploratory coding, as shown in Exhibit 3.

Carrying on in this fashion, we continually add to the system knowledge, backtrack, and purge the “incorrect” knowledge. The final “product” is shown in Exhibit 4.

There are a few observations we can make about this activity:

- *The problem of late discovery.* The approach does not work too well when there is a likelihood of later knowledge invalidating earlier knowledge. On the graph this would mean a very big backtrack and a lot of dotted “unknowledge.” In the real world this would mean a large amount of redesign late in the development cycle. This happens often in larger, complex systems where a great deal of information is obtained from the later design and testing phases. It also seems to be a feature of embedded real-time systems and other applications where there is a high degree of design dependence.
- *Two kinds of knowledge.* We are actually acquiring two different kinds of knowledge: solid-line and dotted-line, knowledge and “unknowledge,” or in English what works and what does not (for this system). Note that the “solid-line” knowledge is incorporated into the software artifact, while the “dotted-line” unknowledge is simply thrown away. We could argue that knowing what does not work is also

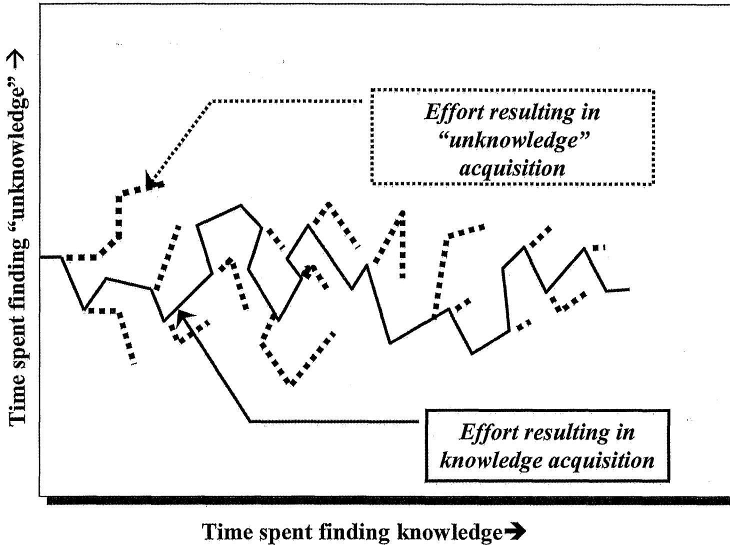

Exhibit 4. Hacking project: Full build.

Exhibit 5. Edison quote.

Just because something doesn't do what you planned it to do in the first place doesn't mean it's useless.... If I find 10,000 ways something won't work, I haven't failed. I am not discouraged, because every wrong attempt discarded is just one more step forward.

— Thomas Alva Edison

potentially very valuable information but such unknowledge is usually thrown away. Thomas Edison's apt quote in Exhibit 5 does not tell the full story. Not only are missteps eliminating "wrong" paths, the process of "misstepping" may be the only mechanism available to us to illuminate the real path. Sometimes we have to try it to find out what will work by exploring what does not.

- *Corrupted knowledge.* The final delivered product (the wandering solid line) is not usually a good and clean representation of the knowledge necessary for the system. The "kinks" in the line are caused by the activity of acquiring the knowledge or, more correctly, the activities of separating the knowledge from the unknowledge and validating the knowledge in some way. That is, the final code representation of the knowledge does not just contain the knowledge

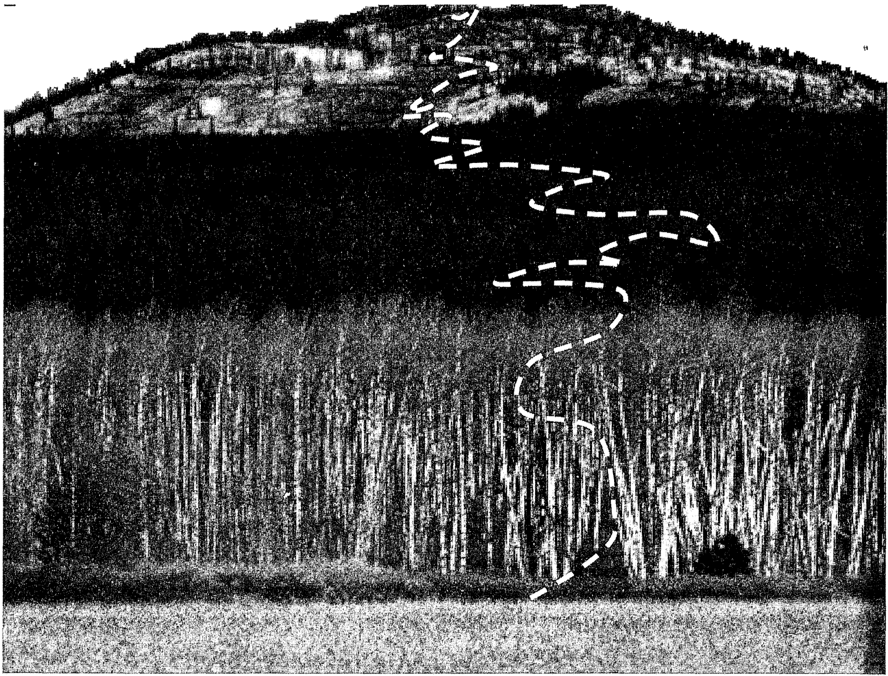

Exhibit 6. A path through the woods.

> necessary for the system to function correctly. Unless great effort is made to separate what works from what does not, it also contains the remains of the journey to find that knowledge.

Invariably, the final code product is somewhat contaminated with the legacy of the process used to build it. Usually the developer knows this and understands that the code, while "correct," is not "good code." However, no one else does. And in a year's time, even the original developer will be at a loss to explain just why the code is written the way it is. The reason for this is that the *contextual knowledge* (knowledge of *why* the code was written the way it was) was stored in the most volatile of knowledge stores — the human brain of the developer.

A Walk in the Woods

An analogy may help to explain. Imagine walking through a dense wood through which you have never set foot (see Exhibit 6). Given that you have never walked through this wood before, it would be close to impossible to traverse the wood without taking a "wrong" path. This is intrinsic to the process of discovery. Even if the destination is clearly visible (the requirements are well defined), the path to get there is at least partially obscured.

In fact, we could argue that the only way in which we could flawlessly navigate to the destination is if we had already been there (we have already built this system). In which case, we could argue "*why are we going there again?*" If we had a map available to us to assist us in minimizing our wrong turns, it means that *someone* (the map maker) must have followed this path. This means this product has already been built — so, again, why are we building it? The concept of a process "map" is often held out as the purpose of establishing process in software development. We shall see later how flawed this idea is in practice.

The only way in which we can very quickly and effortlessly navigate to the destination is if someone has built a six-lane highway through the wood. In this case we are, of course, going in the same direction as everyone else. In the systems sense, we are building the same kind of system everyone else is building, in which case we have no competitive advantage.

A Path Less Traveled

We could argue that the only paths we should travel are those that *no one* has taken before. These are the journeys that lead into the unknown, to the novel destinations, that uncover new knowledge, rather than revisit old knowledge. While we rarely if ever develop 100 percent new systems, the entire knowledge content of which is novel, most systems of any worth have at least some of this new discovery. But it is in this *new* knowledge that the real value of the systems lies. What we shall explore is the nature of this knowledge and the processes we use to discover and encapsulate it.

Tracks

As we make our way through the woods, we leave footprints. When we find ourselves backtracking because the path we took turned out to be "wrong" — it led to a different destination than the one we wanted — we leave more footprints. Unless we are very careful to wipe out the footprints heading in the wrong direction, they will still be there when someone else follows us. In the absence of other information, these tracks are likely to lead the other person astray also. In code, these tracks are the legacy of the earlier attempts to write effective code. Unless the author works hard to remove them, there will be extra variables, states, conditional statements, loops, and other code devices that are not necessary for the final solution of the problem. It is the sheer amount of rewriting of code to remove this legacy that makes the hacking model a poor one for larger systems

It is not usually possible to tell immediately if the code is "real path" code or a legacy from a "false path" — unless one has an alternative source of knowledge. For the person writing the code, it may be in his or her brain. For the maintenance programmer several years later, the comments in the code (a form of knowledge-in-books) may explain why the code looks the

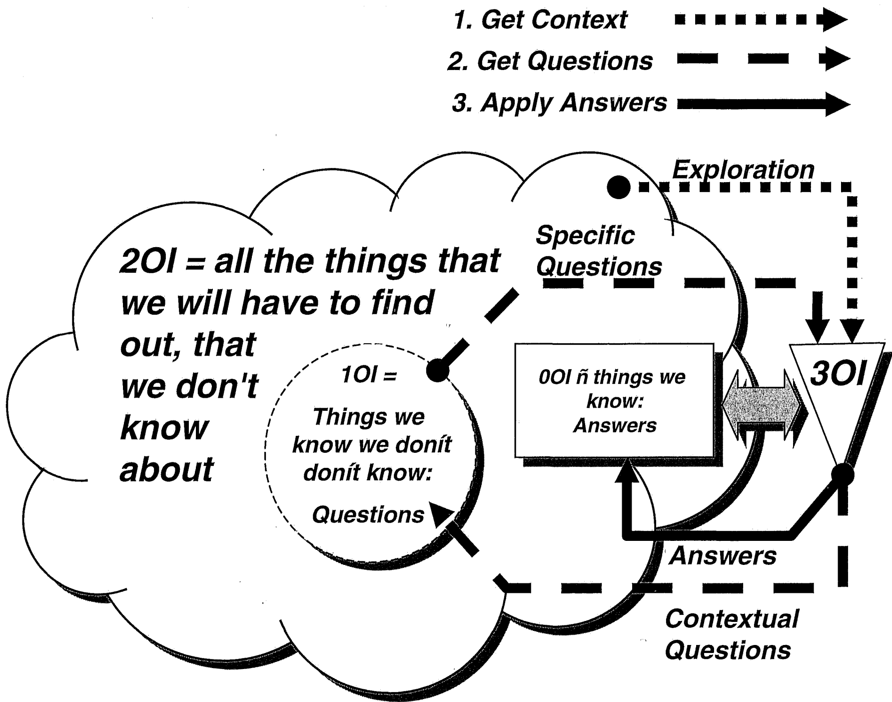

Exhibit 7. Prototyping.

way it does. The difficulty of separating real knowledge from the missteps is one of the reasons that reverse engineering activities have rarely been effective.

Code is, in essence, a *write-only* knowledge store — it is much easier to put knowledge into code than it is to extract it.

Prototyping

From the prototyping example (Exhibit 7) it is evident that the real job is not writing the code or even "building" the system — it is acquiring the necessary knowledge to build the system. In fact, when hacking we use the activity of building the system (or rather attempting to build the system) as the *mechanism* for understanding what the system has to do. Hopefully, the correctly coded system is a by-product of this activity. The problem arises when we think the code *is* the product rather than the knowledge in the code. Then we are tempted to ship the code as it is, however it is, once we get enough of it. If we wished to gain an untainted representation of the code, what we should do with the hacking model, of course, is to rewrite the code so that 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 and arguably a business model prototyping actively acknowledges that our job is not to build a system but to acquire knowledge. We do not expect to get a functioning system first time out when we prototype. What we do expect to get is (at least some of) the knowledge we need to build the system. We use prototyping particularly when we do not know in advance what kind of knowledge we might need. We would not consider prototyping in situations where we knew what we had to do in advance.

While we have used the activity of hacking code as a way of explaining this concept, it is actually true of all development and all development stages. We leave tracks and missteps in feasibility studies. We have ambiguities and mistakes in requirements and design documents. We learn things that invalidate what we have put down to date during the creation of test cases and test plans just as much as in code. All software development is predominantly knowledge acquiring rather than product producing.

The Expectation of Product

However, the acquiring of knowledge is neither the business expectation nor the business goal in most companies. Few companies, even those that create and sell software only, count knowledge acquisition and management as their highest priority. Most operate on a modified manufacturing model that views the creation and delivery of the system to the customer as the highest priority. It is not, and this prevailing view has caused considerable problems to both customers and developers for decades.

Kinds of Knowledge

If our job is to acquire knowledge, what kinds of knowledge should we acquire? In a later chapter we will discuss systems knowledge as well as other kinds of essential knowledge that is not coded into the functional artifact. For now, we will talk in more-general terms about what we might know and what we might not know.

For every item of knowledge we possess, we also have a certain amount of ignorance. In fact there is evidence that our "ignorance" *always* exceeds our knowledge. Ignorance is simply the other side of the coin of knowledge. If we view systems development as the acquisition of knowledge, then we can also view it as the reduction or elimination of ignorance. We can also reasonably assume that, at the start of the project, we are more ignorant than we are at the end of the project, although as we shall see, in terms of

known ignorance, this may not be true. So what kinds of ignorance might we exhibit?

Based on what we know and what we do not know, we can classify our ignorance into strata or layers. I call these levels the *"Five Orders of Ignorance."* While the concept is quite general and even somewhat philosophical, quantizing our knowledge and ignorance can be helpful as we try to understand what we need to do to learn and build a system that works. The Five Orders of Ignorance (5OoI) also helps to explain some of the puzzling things that routinely happen in the software development environment, and also some of the behaviors we exhibit trying to create software.

The Five Orders of Ignorance

For very logical, but to noncomputer folk entirely baffling, reasons we in the software business always start counting from zero rather than one. Therefore, the 5OoI start with zero.

Zeroth Order Ignorance (0OI): Lack of Ignorance

I have Zeroth Order Ignorance (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 provable and proven knowledge that is deemed "correct" by some qualified agency. In software this means that the knowledge is invariably factored into usable form. In all forms of knowledge there must be some external "proof" element that qualifies the knowledge as being correct.

In a nonsoftware arena and as a personal example, because 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.

First Order Ignorance (1OI): Lack of Knowledge

I have First Order Ignorance (1OI) when I do not know something and I can readily identify that fact.

1OI is basic ignorance or lack of knowledge. Example: I do not know how to speak the Russian language. I could remedy this deficiency by taking lessons, reading books, listening to the appropriate audiotapes, or moving to Russia for an extended period of time.

Second Order Ignorance (2OI): Lack of Awareness

I have Second Order Ignorance (2OI) when I do not know that I do not know something.

That is to say, not only am I ignorant of something (I have 1OI), I am unaware of what it is I am ignorant about. I do not know enough to know

what it is that I do not know. Example: I cannot give a good example of 2OI, of course.

Third Order Ignorance (3OI): Lack of Process

I have Third Order Ignorance (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, which is lack of a suitable knowledge-gathering process.

This presents me with a major problem: If I have 3OI, I do not know of a way to find out that there are things that I do not know that I do not know. Therefore, I cannot change those things that I do not know that I do not know into either things that I know, or at least things that I know that I do not know, as a step toward converting the things that I know that I do not know into things that I know.

For systems development, the *"suitably efficient"* proviso must be added, because there is always a default 3OI process available. The "default" 3OI process is to go ahead and build the system without knowing what is not known. The code hacking model does this using the coding activity. For very small systems, with certain characteristics that we shall discuss later, this can sometimes be an efficient process. For larger systems, the default 3OI process is usually neither suitable nor efficient

Fourth Order Ignorance (4OI): Meta Ignorance

I have Fourth Order Ignorance (4OI) when I do not know about the Five Orders of Ignorance.

I do not have this kind of ignorance, and now neither do you, dear reader. 4OI is *meta ignorance* — it is rather like being ignorant of the subject of ignorance. However, a version of 4OI is the prevalent attitude that this book attempts to challenge; specifically, that software is a product and that the software development business is the business of building systems rather than acquiring knowledge.

Knowledge is highly and intrinsically recursive — to know about anything, you must first know about other things which define what you know. The Fourth Order of Ignorance for software development purposes could be restated as: *"I have Fourth Order Ignorance when I don't know that software development is the activity of acquiring knowledge, and I don't know what my levels of knowledge are."* It reflects the natural recursion we always encounter when talking about knowledge.

The Five Orders of Ignorance in Systems Development

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

0OI

0OI is provable, functional, and correct knowledge. In order to qualify for the label "knowledge" it must have been:

- "Known" by someone
- "Validated" against another source of knowledge
- Made into an executable form (if the storage medium of choice is software)

These are the correctly functioning elements of the system that I (obviously) understood, and have successfully incorporated into the system. When I have 0OI, I have the *answer* to the problem.

1OI

These are the things I know I do not know. In a typical system's development project, they are the known variables, where the presence of the variables is known, but not their instance values. When I have 1OI, I have the *question*. In the gamut of systems development effort, we usually find that having a good question makes it fairly easy to find the answer. Of course, we may have a good question but not know how or from whom to obtain an answer. This means our 1OI is incomplete, and incorporates other levels of ignorance. We will tackle more subtle variations of the Orders of Ignorance in a later chapter.

2OI

Second Order Ignorance represents my primary problem in constructing systems. Not only do I not have the answer I need, I do not even have the question. This is, in fact, where we start many projects. Usually, when we start a project, we know from experience that there are many things we will have to learn. The problem is we just do not know what they are. 2OI explains, for instance, most variation in project estimates, and the famous "90-Percent-Complete Program Syndrome."

3OI

Third Order Ignorance operates at the process level. Rather than lacking product knowledge (i.e., of the target system), I am lacking knowledge of how to *acquire* the target knowledge. For the fully qualified 3OI, I am lacking the knowledge of how to acquire the knowledge in a suitably efficient way. This means I do not have a sufficiently effective process that will allow me to build the system (acquire the knowledge) within my budget and time constraints. If this is coupled with 2OI, I have a real danger — I simply do not have a way to resolve my lack of knowledge in the time I have available. In later chapters, I maintain that *all* software development methodologies are

actually 3OI processes; their job is to show the areas of the product or process where there is lack of knowledge.

Coupled with 2OI, 3OI represents the true challenge of software development. The reasoning is simple: I have 0OI (the answer), then it is simply a matter of putting the extant knowledge into the product (assuming that I know how to do that, of course). If I have 1OI (the question), it is simply a matter of finding out where the answer to the question exists and obtaining that answer. While resolving 1OI is somewhat more effortful than applying 0OI, both these operations are typically low in effort. It is in the reduction of 2OI and 3OI that the real effort lays.

In the pursuit of process and methodologies, people and organizations sign up for some very "heavyweight" procedures: enormous manuals on how to factor systems, huge checklists, multiple process steps, and repetitive reviews and inspections. Others look for the answer in the methodology — they adopt a set of complex and difficult systems definition and design conventions and languages in the hope that in transcribing their knowledge into these modeling forms, they will acquire the knowledge they need. Both process and methods (and languages) have their places and they are important. But it is important to note that the answer we are looking for *cannot* be in the methodology or the process. A methodology simply gives the syntax in which to frame the question and a discipline for identifying those areas where I might have 2OI. But it cannot know what I am trying to do. A process simply gives a framework in which the discovery activities can take place. The process cannot *perform* the discovery activities.

A movement is afoot in the software business that is leaning toward what are called "Agile" (or "lightweight") methods. These methods attempt to allow for the freedom of discovery while still maintaining the consistency of process necessary to obtain predictable and repeatable results. We will discuss these methods at length in a later chapter.

4OI

Fourth Order Ignorance is probably not too much of an issue at a practical level on projects, although I have found thinking of the process of developing software even in the small does help. At an organizational level, I believe this is the problem that is holding us back from truly capitalizing on the productivity gains we are capable of. The nature of knowledge is recursive, and it is appropriate that the "highest" level of ignorance reflects this recursion.

The 3OI Cycle

The function of process is threefold:

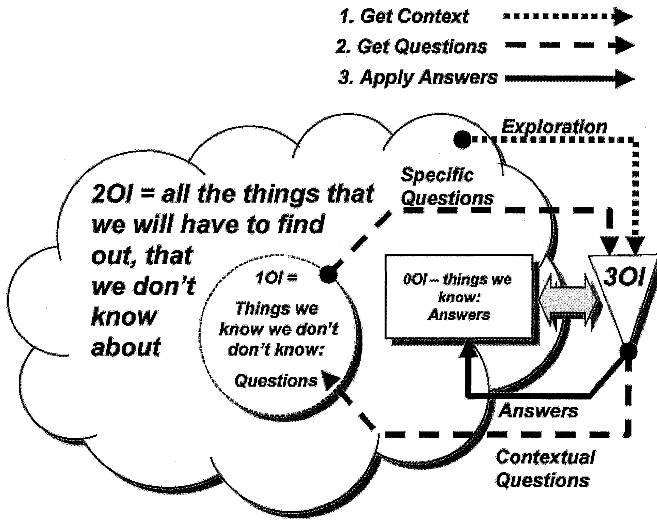

Exhibit 8. Order of Ignorance cycles.

1. To identify whether there are areas where we have ignorance (need to acquire knowledge)
2. To identify what questions we would need to ask to resolve ignorance in these areas
3. To obtain the answers to these questions in a form that we can usefully integrate into the system

The operation of these processes is shown in Exhibit 8.

- The "highest level" 3OI process operates on the "body of *lack of knowledge*" implied by the 2OI cloud. The 3OI process is shown by the small dotted line. The 3OI process somehow acts on both the environment that (presumably) contains the needed knowledge plus the currently available knowledge present in the project team. How the 3OI process actually works depends greatly on the system, the situation, how much knowledge is needed, and what is already known. For situations with high degrees of 2OI, these are extremely exploratory operations, often executed in cycles, with each cycle closing in on the real knowledge source. The output from this phase is a set of contextual questions. Comparing the contextual questions and their answers against the environment allows us to identify where our ignorance lies.
- The next level converts the contextual questions into specific questions. The purpose of the first loop is to identify *where* we might

have ignorance, this second loop is to identify *what* that ignorance might be. Fed through the process, these questions should elicit answers, although commonly each answer generates other questions and sometimes painfully illuminates whole areas that require further investigation.

- The final step is to convert the specific questions into specific answers and apply these answers. At this point we can certify the output as 0OI or extant knowledge.

The steps are:

Undifferentiated Lack of Knowledge (2OI) →
Identified Lack of Knowledge (1OI) →
Knowledge (0OI)

The fly in the ointment has already been identified — it is that the question-answering process almost invariably generates more questions. Simply put, acquiring knowledge also illuminates more areas of lack of knowledge. For projects tackling systems targets with large quantities of 2OI, this can seem never-ending. For each fact that is found, it seems that an equal number of questions are raised. If a project's or a manager's vision of the goal is a fully factored set of knowledge as exhibited by a working system, simply exposing more and more areas of ignorance is very frustrating. However, if we took the goal to be the acquisition of useful knowledge, we would find the process significantly less frustrating. Here we see one of the functions of changing our business goals, outlook, and expectations away from product and onto knowledge.

The Inability to Measure Knowledge

The view of software as a knowledge medium and software development as a process of acquiring the knowledge necessary to populate this medium leads us directly to a very uncomfortable conclusion concerning what we do. After thinking about the problem for a couple of thousand years, *the human race has not found a way to empirically measure knowledge*. Not only can we not measure it, we do not have a unit for knowledge. We can weigh a book, we can count the number of pages, the quantity of lines and words in it, but we cannot count the quantity of knowledge in it. There is simply no way to do this.

Assessing quantity of knowledge is *always* done using a comparison against another body of knowledge. There are no knowledge measurement axioms on which we can base a quantification system. This is true for books, it is true for humans, and it is true for software. In fact, we assess the quantity of knowledge in a software system in exactly the same way we assess the quantity of knowledge in a human — by examination. In a human, the test results from the completed examination paper (run under

controlled conditions) are compared against the professor's answer. If the match is sufficiently close, the person gets a gown, a hat, and a roll of paper. This certifies that the person (at that moment in time) "knows" a quantity of knowledge. For a software system, if the actual observed results of the test (run under controlled conditions) sufficiently match the expected results, it is presumed that the system does, in fact, contain the required knowledge (at that moment in time). The system is certified for use and is released to the world to go and find a real job. Program code inspections closely resemble job interviews for much the same reason — they are both knowledge and capability assessment practices. If a person passes the interview it is presumed that he has either the necessary knowledge or capability or both, and he is offered a job. If a piece of code is deemed by inspection to possess the appropriate knowledge, or is sufficiently well structured that the expected knowledge can be easily added (a measure of knowledge capability), the program is released into the next stage of development.

Our inability to actually measure knowledge means that much of our metric process is built on a foundation of sand. Compounding this is the fact that the critical measure of knowledge in software is not the measure of knowledge in software; it is the measure of the knowledge that is *not* in the software. This is the knowledge we have to *get*, not the knowledge we already have. As described earlier, the key determinant of a software project is the 2OI, which is knowledge we do not know we do not know. So we are in a double bind. Not only can we not measure knowledge we have, what we really want to measure is knowledge we do not have. If we could empirically measure knowledge, we would be able to assess 0OI, and probably we would be able to do a good job at measuring 1OI. We still would not be able to "accurately" measure 2OI, because we would still not know what it is by definition.

This is not a purely philosophical challenge. All project estimation approaches fail to some degree at this point. All project status tracking efforts are compromised by this, and it is the biggest source of recurrent failure in our ability to make commitments we can keep and keep commitments we make.

Summary

At a practical level in developing systems, the critical levels of ignorance on most projects seem to be 2OI and 3OI. It is reasonable to assert that almost all of our work on projects involves the reduction of 2OI into 1OI and finally into 0OI. The rationale is straightforward: if we already have the answer (0OI), it usually does not require much effort to apply it. Even if we do not have the answer, but we do have a specific question (and presumably also the knowledge of how to get an answer), then obtaining the answer does require some effort, but not much. The effort-intensive activ-

ity is discovering what it is we do not know. Therefore, it is reasonable to assert that most of our work is the reduction of 2OI. We will also assert that *all* software and systems methodologies are 3OI processes whose job is not to tell us what we know as much as to illuminate our 2OI. The application of a 3OI process to 2OI generates either 1OI or more rarely 0OI. That is, applying an effective development process either gives us the answer (0OI) or, more commonly, it gives us the question (1OI).

Because process and methodologies are often sold on the basis of how much they can structure the knowledge and the activity of acquiring it, it can be quite startling to realize that the primary purpose of process is to show us where we have *lack* of knowledge. Yet if we acknowledge that the true role of the development process is to acquire knowledge, and the most valuable knowledge is knowledge we do not already have, this is the most powerful thing we can do in development.