THE FIVE ORDERS OF IGNORANCE: KNOWLEDGE, IGNORANCE, AND THE NATURE OF SOFTWARE

Abstract

Software is not a product, it is a medium in which we store knowledge. As simple as this idea seems, the consequences of it are quite significant. If software is not a product, then software development is not a product production activity, despite the common practice of managing it as such. Most organizations believe that the job of software developers is to build a system that we then ship to a customer. It is not. The system we build and ship to the customer is actually the ***by-product*** of the real activity which is learning. Software development is the activity of acquiring certain kinds of knowledge. The software medium is simply the place we put the knowledge once we have acquired it.

Acquiring knowledge can also be considered as the reduction of ignorance. This article makes some observations on the nature of software, the acquisition of knowledge, the reduction of ignorance, and how these activities play out in the software field. In doing so, it helps to explain the some of the classic conundrums of software development.

THE FIVE ORDERS OF IGNORANCE

Software is not a product, it is a medium for the storage of knowledge. 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.

DNA is where species store knowledge. When certain kinds of knowledge are stored in DNA, the individual members of that species do not need to acquire that knowledge themselves. For instance, as humans we do not need to learn how to make our pancreas function—this information is stored in our DNA and is available when we are born.

The brain is the second knowledge storage medium. Humans, for instance, store most of their behavioral knowledge in this medium. Our social interactions are largely learned, not inherited. To differentiate DNA and brain-stored knowledge consider our use of language. All humans are born with a capacity to learn a language—this capability is "hard-wired" in DNA. The specific language, however, is learned and can be relearned, replaced, or forgotten. This knowledge is stored in the brain.

The third knowledge storage medium was hardware. We do not usually think of hardware as "knowledge storage," but that is a large part of its function. For instance, humans are poor at

l and store on it the knowledge of standard length. We can then use this ruler to reapply the knowledge of standard length.

Books are an entirely passive storage medium. In order to activate knowledge in a book, it must first be removed to another medium, typically the brain.

The most recent knowledge storage medium, invented a mere fifty years ago is software.

At the present time, we are spending enormous amounts of effort in transferring our knowledge from the other media into software. The reason is that, in software, knowledge is made both usable and transportable. Knowledge in software is ***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 that is missing from knowledge in DNA or hardware.

But software is a medium not a product. The true "product" of our efforts to "produce" software is the knowledge contained in the software. In fact, it is rather easy to produce software in any quantity. It is much harder to produce software that "works", because we have to understand what "works" means. It is easy to produce software that is simple, because it doesn't contain much knowledge. It is easier to produce software using an application generator, because much of the knowledge is already stored in the software. It is easy to produce software if I've already produced this type of software before, because I have already obtained the necessary knowledge.

The hard part of building systems is not building systems, it is in discovering what we are trying to or need to build—it is in acquiring the necessary knowledge. If software is not a product then software development is not a product production activity—it is it is a knowledge acquiring activity. However most companies that build software use business models derived from product production. The nature of a product production business is quite different than the nature of a knowledge acquisition or learning business.

If the job of software development is to acquire knowledge, what can we assert about the knowledge we have to gain? For every thing we know, we must also have a certain amount of "ignorance". Ignorance is 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. So what kinds of ignorance might we exhibit in software development and how would we deal with it?

The Five Orders of Ignorance

Based upon what we know and what we don't know, we can classify ignorance into strata or layers. These are called the **"Five Orders of Ignorance"**. Understanding their nature and how we navigate through them as we create software can be helpful in explaining some of the characteristics of software development. They also help to explain some of the artifacts of the software development environment, and some of our behaviors working in this environment.

In computer systems, the first number is usually zero. And so it is with Orders of Ignorance.

0th Order Ignorance (0OI)—Lack of Ignorance
I have Zeroth Order Ignorance (0OI) when I (provably) know something.

I must be able to demonstrate my knowledge, otherwise it may be that I do not actually know what I profess to know. This is clearly demonstrated in the testing of systems—we cannot generally assert that a system works (ie., contains and executes the correct knowledge) unless we can prove it through the agency of a controlled test.

1st Order Ignorance (1OI)—Lack of Knowledge
I have First Order Ignorance (1OI) when I don't know something.

If I don't know something but I am aware of this fact, then I have 1OI. 1OI is basic and identifiable lack of knowledge. This occurs when we build systems of a type we have already built. If we had already built tax systems for US State Governments and were then engaged to build a tax system for (say) the State of Ontario, Canada. We know that the tax systems are quite different, but they are also probably somewhat similar too. Without much effort, we could construct a series of questions the answers to which will give us the information we need.

2nd Order Ignorance (2OI)—Lack of Awareness
I have Second Order Ignorance (2OI) when I don't know that I don't know something.

This means that not only am I ignorant of something (I have 1OI), I am also unaware of that fact. I don't know enough to know that I don't know enough. In essence 1OI is a type of knowledge (I do know what I don't know) whereas 2OI is a more complete form of ignorance. It is 2OI that causes us most problems in building software systems and is also the primary reason why we test software at all.

3rd Order Ignorance (3OI)—Lack of Process
I have Third Order Ignorance (3OI) when I don't know of a suitably efficient way to find out that I don't know that I don't know something.

3OI is lack of process and when coupled with 2OI presents an intractable problem, since I will continue to be unaware of those things I do not know. In software development, we must add the *"suitably efficient"* proviso, since there is always a default 3OI process available—build and ship the system. In this case the customer can be relied upon to inform us of all the things we did not know. However, this process is usually neither suitable nor efficient.

4th Order Ignorance (4OI)—Meta Ignorance
I have Fourth Order Ignorance (4OI) when I don't know about the Five Orders of Ignorance.

4OI is *meta* ignorance. While it provides a nice recursive end to the Five Orders of Ignorance, there are practical applications of the concept.

The Five Orders of Ignorance in Software Development

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

0OI: 0OI is knowledge. These are the correctly functioning elements of the system that I (obviously) understood, and can successfully incorporate into the system. When I have 0OI, I have the ***answer*** to the problem. The direct transfer of what we already know into an executable form does not usually require much effort. If this knowledge is already stored in an executable form (eg. a reuse library), it may require even less effort than if the knowledge is stored in a brain (an experienced developer) or in book form (documentation).

1OI: These are the known variables, where the presence of the variables is known, but their specific values are not. When I have 1OI, I have the ***question***. Usually, having a good question

makes it fairly easy to find the answer. Checklists of questions to ask customers are evidence of 1OI. Many paper driven methodologies are of this form.

2OI: This is the source of most problems in software development. Not only do I not have the answer I need, I don't even have the question. This is where we start many projects. At the beginning of a project we know, from experience, that there are many things we will 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). It 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. But because of 2OI, the actual progress cannot be determined with accuracy. 2OI also accounts for rework cycles, late phase "gotchas" and is the primary reason why we test systems.

3OI: Coupled with 2OI as it usually is, 3OI presents a real danger—I don't have a way to resolve my lack of knowledge in the time I have available. At their core *all* software development methodologies are actually 3OI processes, though they are rarely considered as such. The true purpose of a methodology is to show me the areas of the product or process or whatever where I have lack of knowledge. However, many methods and modeling approaches and development disciplines purport to give us the answer. They cannot. They do not contain the answer. The best that systems modeling can do is show me whether or not I have the answer. Conforming my understanding of the system's knowledge to the syntax of the model forces me to consider whether I can describe it or not. If I can, I must have the knowledge; if I cannot, I do not have the knowledge.

The job of a 3OI process is to convert 2OI into 1OI (and sometimes into 0OI). Testing systems is a 3OI process whose job is to (a) prove that our 0OI is correct and (b) find out if there is anything else we don't know about. Note, we would never explicitly test for 1OI.

4OI: As a meta-level is more abstract. However, the fact that most organizations creating software believe and act like their job is to build product rather than acquire knowledge is proof of 4OI. In doing so, companies simply do not take care of the primary product: the knowledge within the system. This is clearly shown in the area of Configuration Management (CM). Most CM systems will dutifully return (say) version 1.1 of the system specification upon request. But what is ***in*** version 1.1 of the system specification? The CM systems have no visibility into the content. They do not manage the knowledge asset, they manage the buckets into which we put the knowledge asset.

The critical levels in software development are 2OI and 3OI. Most of our product-directed work occurs in the reduction of 2OI. The development and use of software and systems methodologies are 3OI processes.

There is a further differentiation: there are two kinds of work we do in software development: the application of what we know, and the discovery of what we don't know. The same kind of process does not work for both activities. Processes that work for 0OI and 1OI (the application of what we know) do not work for the less deterministic 2OI and 3OI discovery activities.

This is further evidenced in the application of processes and methodologies. These are often sold as providing the answer, when in reality and correctly used they provide the ***question***. Disciplined use of such approaches then gives the appearance of ***increasing*** the amount of work to be done. Coupled with the additional knowledge necessary to actually use the tools and methods, and the source of resistance to such process changes is clear. Practically speaking, we need to separate the application from the discovery activities and apply strong process to the first, and allow freedom for the second. These polarities are not possible in the same definition of process; we cannot both constrain and liberate at the same time.

In essence the working system is merely the ***proof*** that we have obtained (and validated) the necessary knowledge. The functioning system is the ***byproduct*** of the activity of acquiring knowledge that is the core of software development.