

# The Zen of Engineering

A collection of essays written by John Wyatt  
©2002, 2008, 2010, 2012

*Before any words, there is something here, something present, something paying attention. It is very ordinary in the sense that it is happening all the time. It is the spacious Awareness in which everything happens. It's just nice to notice that during, after, and in between experiences, it is always here.*

- Nirmala -

*To merely acquire information or knowledge is not to learn.  
Learning implies the love of understanding and the love of doing a thing for itself.  
Learning is possible only when there is no coercion of any kind.*

- Krishnamurti -

*No matter how many instances of white swans we have observed,  
this does not justify the conclusion that all swans are white.*

- Karl Popper -

Preface.....	3
Zen of Defects.....	4
Zen of System Control .....	16
Zen of Validation.....	25
Zen of State Machines .....	33
Zen of Alarms.....	41
VVISSIIOONN .....	50
Nature Walk in the Big City .....	57
Charge of the Developer's Brigade .....	59
The Asylum .....	61
The Project Manager .....	65
Stone Age Programming.....	73

# Preface

*What I have to say has all been said before,  
I am destitute of learning and of skill with words.  
I therefore have no thought that this might be of benefit to others.  
I wrote it only to sustain my understanding.*  
- Shantideva -

When teaching children I enjoy using the Wittgenstein's Beetle problem as a practical illustration of the uselessness of words. I give one child a box containing something, such as a small toy or a piece of candy. The child is then asked to describe the object without ever using any name for it and without saying how it is used. The rest of the class tries to guess the object. In all the years I've done this, no one has ever guessed the object. After showing the object, everyone instantly knows what it is - no words are needed.

When all we have are words, we are never able to grasp the reality. Once we grasp the reality, words are not needed. However, sometimes words are all we have. The words in this collection of essays are to be treated as merely pointers to something - in most cases my attempts to avoid the "Black Swan" effect<sup>1</sup> in our engineering projects. But just as we do not confuse the finger pointing to the moon as being the moon itself, these words are not to be confused with that to which they are pointing. Once the reality is seen, the words will be useless.

So this is my goal - to write because it helps me to organize my own thoughts and to provide pointers that others may find useful as a guide. Once grasped, these words will be entirely unnecessary. I'm not interested in the agreement of others; dissent is simply a sign that I've achieved my real goal - to spark thought. My real goal is probably best stated by a passage from the Bhagavad-Gita (2.46):

As unnecessary as a well is  
To a village on the banks of a river,  
So unnecessary are all scriptures  
To someone who has seen the truth.

I wrote these essays over a period from 2002 to 2012 and have occasionally updated them as my thoughts move. Most are technical in nature, some highly technical and riddled with formal notation. A few are decidedly not technical but are included as examples of taking time to look at ordinary things from a completely different point of view and thus recognizing and challenging all of our assumptions. After all, it's our assumptions about reality that blind us to the Black Swans in the first place.

*"All knowing is past tense. By the time our senses record it, it is already over.  
We're always the last to know."* – Nirmala -

---

<sup>1</sup> "Black Swan" effect: a metaphor that encapsulates the concept that the event is a surprise (to the observer) and has a major impact. After the fact, the event is rationalized by hindsight. The purpose of the metaphor is to make us aware of the psychological biases that blind us to uncertainty and keep us unaware of the severe impact of these rare events.

# Zen of Defects

## ***Abstract***

I once saw a cartoon in which a student announced, “Can I leave now – my brain is full.” With the plethora of software tools and practices on the market (all highly recommended by the manufacturer), I often feel as if my brain is full. Why do I have to change? Can I just write bugs the way I always have?

Over the years, I have seen many programmers resist changes to the way that they design and write software, with a resulting increase in development time and defects. Their attitude was “what worked before will work again,” even if the reasons why it worked before were no longer valid. Naturally, whenever I was guilty of this same dysfunction, I could always cite excellent reasons for continuing to make the same mistakes! But in the end, the need to maintain my own software changed my attitude towards Software Engineering. In this paper, I explore a way in which we can improve our practices with a simple shift in mindset – all with the same brain. I believe that a deeper understanding of the core purpose of Software Engineering can bring about this mind shift and help us to adopt new practices; and it is a lot easier than a trepanning<sup>2</sup> (and considerably less painful).

## ***Thesis Statement***

I contend that, without exception, all software engineering practices relate to one, and only one, attribute: the ability of a human being to understand the software being designed and written.

Corollary 1: The Computer is irrelevant to Software Engineering.

Corollary 2: Defects Do Not Exist.

Corollary 3: If you don’t understand the problem, the software won’t work.

## ***Why Write This Paper?***

For that matter, why make inflammatory, absolute pronouncements as thesis statements? Simply put, to create controversy. If the rationale behind various Software Engineering practices were readily apparent, it would be a simple matter for programmers to immediately see the efficacy of these practices and adopt them; however, this is often not the case. Recall that Civil Engineering came about because of the collapse of bridges and buildings. The field of Medicine came about because trepanning does not work. Likewise, the field of Software Engineering came about because of continual schedule overruns and high defect rates.

Why do some people fail to see how right I am and change their ways? For the same reason that I failed to see how right others were and change my own ways. Cultural anthropologist Edward Hall (1976) notes that the part of the human nervous system that deals with social situations is designed according to the principle of negative feedback. This means that a

---

<sup>2</sup> “Trepanning” is an ancient medical practice in which the patient had a hole bored into their skull to release the evil spirits that were causing the malady. Today we use meetings – the effect is the same.

person is completely unaware of the fact that they have an inner system of cultural and social controls as long as the social program is being followed. Hall further notes that once a person has acclimatized to an environment, they behave as if that environment were an innate facet of their being. The only time that a person becomes aware of their inner controls is when an outside event fails to follow the hidden program. This change in outside events produces a fear reaction; however, the person's initial feeling is simply that "something is wrong."

In just this fashion, I have seen many programmers (not myself, of course) develop a preferred style of programming and simply accept that as the "norm." New practices may elicit interest if the programmer encounters them independently; however, when they are presented by an outside agent, a fear response may result, triggering resistance to change. In this paper, I examine the root cause of all software engineering practices with the hope that it will create some controversy and help to overcome resistance to change. If this fails to work, I may have to resort to trepanning.

### ***What Is Software Engineering?***

The IEEE Computer Society defines Software Engineering as: "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software." (SWEBOK, 2004). Software Engineering is not the same as Computer Science, which is defined as the study of the theoretical foundations of information and computation. Software Engineering includes Computer Science as a sub-discipline, but is mainly concerned with designing software solutions to real world problems, using a controlled and verifiable process.

A computer could not care less about real world problems – or about anything else. A computer has no capacity for assigning an *intrinsic merit* to any particular computer instruction or series of instructions. It has no reason to prefer one sequence of instructions to another. It has no capacity to understand the concept of "software crash." A computer cannot care what it runs, let alone how it was designed or its software written. Consider also the fact that, without exception, all programs run in machine language – computer hardware cannot run anything else. Whatever instructions the computer hardware uses is, by definition, the "machine's language."

The entire operating system, word processor, network drivers, and ancillary software for the computer used to write this paper could have been written in pure machine language – because that is what it is running anyway. One could argue that a Java program is not written in machine language; however, a Java program is simply a collection of data that is read and processed by a machine language program – the Java Virtual Machine. If a human were to write the entire contents of my computer using machine language, the computer would run just as poorly. There is only one reason why no one writes these programs in machine language: it is *hard!* Of course, the computer does not care how hard it is for us to write in machine language – it is unable to care. Only the programmers can care (and the hapless customers who complain about the software defects). This supports Corollary 1: The Computer is irrelevant to Software Engineering.

## ***What is “Quality”?***

If a defect falls in the forest and there is no Quality Engineer to hear it, does the program have Quality? In the medical industry, safety is a common quality attribute. However, if I were making bombs then safety might not be a top priority – after all, they are supposed to explode and kill people! Ultimately, the only true test of Quality is the User experience. In this context, I often define “Quality” as: a) the presence of useful features, and b) the absence of annoying defects. By “annoying defects” I mean those that reduce the usefulness of the features.

Looking deeper, we see that Reality simply does what Reality does. It is pointless to complain: “Reality is broken” (although I do it all the time). This means that what we call a “defect” is no more than the difference between what I *wanted* it to do (my mental model) and what it *really* did. In other words, defects only exist as *mental models* – the perceived differences between intention and actuality. Of course, we never have access to “what it really did.” We are restricted to sensory input and mental processing to tell us what we *think* it really did.

This shift in mindset moves the responsibility for defects. If the *software* contained the defect, then we could easily blame the software, the hardware, or another programmer – anything other than the real source of the defect. If “It” contains the defect, then “I” need not change anything about “Me.” Some might complain that this is simply a language game, changing the definition of “defect.” However, let us philosophically examine what is actually meant by the statement “the software has a defect.” In Reality, the software does not contain anything that could be called a “defect.” The software only contains computer instructions. The computer has no ability to prefer one sequence of instructions to another. The statement “the software has a defect” has no relevance to Reality. Instead, it is a reflection of the fact that the software’s *actual* behavior, as we perceive it, is inconsistent with its *expected* behavior. The “defect” exists only as our perception of the relationship between the actual and expected results – a “defect” has no concrete existence in Reality.

Furthermore, since the “expected result” is only perceived through our mental processing of sensory input, we are left with the realization that even the actual behavior is no more than our *thought* about the actual behavior – the software may have actually done something entirely different. Most software testers have encountered this phenomenon, which is why they often refer to incidents as “anomalies” rather than “defects.” The term “anomaly” better reflects the fact that the tester has *observed* a discrepancy – not that a discrepancy actually exists.

Since software is the creation of a mental process, it makes sense to bring the definition of a “defect” back to this mental process. By denying that defects “exist,” we shift the focus back to ourselves and our mental processes. This leaves us with no option but to accept that the “defect” resides in the only place where it could have originated, and the only place where it can ever be corrected – in our own mental processes. This supports Corollary 2: Defects Do Not Exist.

## ***Failure to Understand the Problem***

If I do not understand the problem being solved, the software will obviously fail to fulfill its intended function. When a system becomes so complex that no one person can comprehend the entirety, traditional development processes break down. Scientific American’s original

attempt to automate their order process is a prime example of correctly solving the wrong problem because no one on the team understood the entirety of the problem.

Scientific American's objectives were to reduce the costs, errors, and delays in its subscription processing system. The software house focused on the part of the problem having a software solution. The result was a batch-processing system that put extra strain on the clerical portion of the system that had been the major source of costs, errors, and delays in the first place. The software people had looked for the part of the problem with a software solution (their "nail"), pounded it in with their software hammer, and left Scientific American worse off than when they started. Had a business case analysis been done, it would have identified the need to re-engineer the clerical business processes as well. (Boehm 1981). This supports Corollary 3: If you don't understand the problem, the software won't work.

However, in Reality, I only have a *belief* about the problem – I can never say that I truly understand it. No matter how well I may believe that I understand the problem, if my only tool is a hammer, the problem will still look like a nail. So I clearly have to have many tools before I can ever begin to trust my beliefs about my ability to understand a problem. However, understanding the problem is not the real issue – expressing it in symbols that a computer can understand is the real issue. Until the advent of computers, people rarely had problems expressing requirements such as "Build me an army worthy of Mordor."

### ***A Pause for Reflection***

In his book Philosophical Investigations, Ludwig Wittgenstein contended that all use of language is predicated upon an assumption of the end use to which the language statements are being put (1950). Without this assumption, all language statements are meaningless. While the absolute pronouncements of my thesis may be criticized on purely technical grounds, I contend that the *use* to which they are put makes them valid statements. Also, statements themselves have no meaning outside the *language game* that we are playing. Wittgenstein defined a *language game* as the context in which the language statements are being used. For example, if a builder points at a brick and says "brick," is that intended as a *description* of the brick? Is it an *order* that means "hand me that brick?" Language games include descriptions, orders, metaphors, and poetry. So what language game am I playing here?

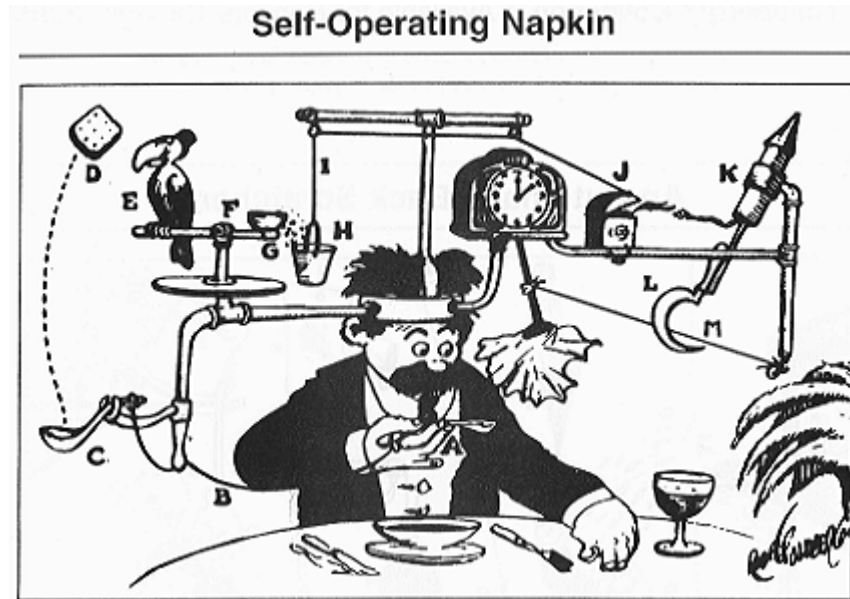
This is my attempt to provoke self-examination; a reminder that all software is a mental construct. Since all software is created in the mind of the programmer, I think it best to make statements that force us to realize the entirely mental nature of this discipline. (Which makes me a "mental case!") Without such self-examination, it becomes too easy to stop growing and simply repeat old behaviors.

Advancements in any field of engineering come from the analysis of past failures and the derivation of methods to prevent such failures in the future. In this respect, Software Engineering is no different from other engineering fields – it is just newer and easier to spell. And, since defects are mental artifacts, we can only prevent them by improving our mental processes.

## ***Software Is a Rube Goldberg Machine***

Reuben Garret Lucius Goldberg (July 4, 1883 - December 7, 1970) is best known for his series of popular cartoons depicting Rube Goldberg machines: complex devices that perform simple tasks in indirect and highly convoluted ways.

(Picture from Rube Goldberg)



When the diner raises the spoon (A) to his mouth, string (B) pulls spoon (C), flipping cracker (D) into the air. The parrot (E) sees the cracker and flies off the bar (F) to grab it, tipping the sand (G) into bucket (H). This pulls down string (I) that, via a pulley, pulls matchbox (J), lighting the match and thus lighting the fuse on the rocket (K). When the rocket launches, sickle (L) cuts string (M), allowing the clock pendulum to swing back and forth across the diner's mouth, thus operating the napkin. A highly complex machine to perform a very simple task!

Now consider this – how many thoughts are contained in this paper? (Probably not very many.) Next, how many characters of text are required to present these thoughts and how long did it take to write them? (28,378 characters, according to MS Word.) Now ask yourself how many machine language instructions are in the word processor that was used to write this paper, and how long did it take to write them? (Over 8 *million* bytes in WINWORD.EXE alone.) Next, consider the number of machine language instructions in the Windows XP® operating system that runs the word processor that was used to write this paper, and how long it took to write them. Finally, how many hundreds of millions of transistors are inside the Pentium chip in the computer that runs the operating system and word processor? Of course, none of those programs were actually written in machine language – that would be too difficult for any human. Instead, hundreds of programmers spent years defining higher-level languages (like C++), then writing the compilers to translate those languages into the machine instructions required to perform those commands. Then they spent years writing the actual operating system and word processors.

Presenting these few ideas is a simple task that can be quickly and easily done with paper and pencil. By contrast, the software to do this on a computer occupies hundreds of millions



of computer instructions that took teams of programmers years to write and debug, and that requires hundreds of millions of transistors to execute. It does not take a genius to realize that software is a Rube Goldberg machine – complex machines that perform simple tasks. In fact, the only thing that makes it possible to even write a program at all is the fact that the task being automated is so pathetically simple and repetitive that we can write a sequence of instructions in sufficient detail that even a computer can follow them!

In one of my electronics classes in the U.S. Navy, we were given fifteen minutes to write detailed instructions for changing the ink cartridge in our pen. These instructions were to be so detailed that they could be used for any ink pen in the room, and no real intellect would be needed other than the ability to read and perform the instructions themselves. Next, we were told to exchange papers with the person next to us, come forward in turn, and change the ink in our pen given the instructions at hand. The results were hilarious! Our antics were amplified by the fact that some of us used fountain pens while others used ballpoints, and some barrels twisted while others pulled! Everyone's instructions contained insufficient detail – such as not telling us to put down the old cartridge before picking up the new one, or not telling us how to orient the new cartridge before inserting it. We came away with an improved understanding of just how many assumptions we make about even the simplest of tasks.

### ***Complex Specifications for Simple Problems***

“Ami, clean your room” is a sufficient specification to have my daughter clean her room (motivation is provided separately). However, what if “AMI” was a robot? How could I translate this simple specification into a syntax that was sufficiently detailed that my AMI Robot could carry it out? I would have to specify in detail how to pick up things, how to open drawers and place things into them, how to handle an exception if a drawer is full, and a myriad of other details that we humans have learned to take for granted (even in our teenage daughters). Also, what does the robot do if someone else is in the room? Will it try and pick them up and stuff them into a drawer? And what if I later want to tell it “clean the bathroom?” Or “clean the living room?” In the end, the task of writing the specification is so complex that I end up writing a specification on how to write specifications (such as UML drawings), and defining syntaxes for describing time sequences and data flows.

Engineering literature often states that requirements specification is the most difficult task in software engineering. I contend that it is only difficult because computers are unable to do anything at all without incredibly detailed instructions. This leads to specialized and highly detailed syntaxes just for defining the problem and writing specifications. In other words, our specifications themselves become Rube Goldberg machines.

I can lay out my daughter's weekly chores on a single whiteboard at the end of the hall. I never have to worry if she is capable of performing these tasks. Motivation is simple – I just promise to plug the X-Box back in when she is done. I never have to write multi-page descriptions of requirements for what constitutes a “room cleaning.” I only have to say “Mom will make sure it's done right” and she knows exactly what to do. The difference is that a human being is capable of learning how to communicate and perform such tasks, but a computer has to be led by the hand each step of the way. The single statement “clean your room” turns into pages of specifications and thousands of lines of code. In fact, the task is so complex that without formal methods of defining the problem and designing the software, it would be virtually impossible. If you think getting a teenager to clean their room is difficult, try writing the specifications for it!

Our ability to define a task (such as “clean your room” or “create a word processor”) implies the ability to create instructions for performing the task in sufficient detail that a computer can perform it. Add to this the fact that much of a program’s code is written just to handle error conditions and it quickly becomes apparent that software is a Rube Goldberg machine. This makes the task of understanding software truly monumental. How can the human brain manage to understand a task in sufficient detail as to create a program that a computer can use to reliably perform the task? Can a human brain even handle such complexity?

## ***Handling Complexity***

In 1956, psychologist George Miller published a paper showing that a human being can mentally manipulate no more than  $7 \pm 2$  mental objects at one time. He says that this forms a concrete limit on our ability to transmit information. Miller concludes by saying that, to manage complex ideas, we must organize complex objects into “chunks,” which then become a single mental object. For example, a brilliant mathematician/programmer once told me that a particular algorithm required taking a Hessian. My response was: “A ‘Hessian’ is an 18<sup>th</sup> century German mercenary. Where do I get one?” Apparently, this was not what was being used in the algorithm. It turns out that, in mathematics, a ‘Hessian’ is a matrix of partial derivative coefficients. To someone who was accustomed to using these, it was a single “chunk;” but for me, it was back to my textbooks on calculus and linear algebra. (And I’ll bet that Frederick the Great doesn’t even get credit for it!)

The concept of  $7 \pm 2$  has been one of many driving forces behind advancements in Software Engineering practices, including Thomas McCabe’s Cyclomatic Complexity metric and the Object-Oriented Design (OOD) methodology. McCabe’s metric recommends limiting the number of possible pathways through a module to no more than 10, a number derived from his empirical studies correlating software defects to the number of paths through a module (Carnegie Mellon, 2008). OOD relies on concepts such as “encapsulation” and “abstraction” to organize complex ideas into simpler “Objects.” A programmer can use an “object” without having to know anything about the internal construction of the “object.” These are just two examples of Software Engineering practices whose purpose has nothing whatsoever to do with the computer, but only with the programmer’s ability to understand the program. In addition, I contend that the difficulty in defining requirements and specifications stems solely from the fact that the computer cannot be told to simply “clean your room.” Simply specifying requirements for something as absurdly stupid as a computer is in itself a part of the program complexity.

I contend that the *entirety* of the Software Engineering field is about only one thing: reducing the complexity of the task by organizing it into “chunks” such that the programmer is now able to understand his or her own design – *because the lack of understanding is the **only** source of defects!* Included here is the lack of understanding of the actual customer needs or how to fill them – a failure to understand the problem being solved. A written specification becomes one “chunk” in the program. There can be wide discrepancies between what the customer thinks they want and what the programmer thinks they really need. There can be a lack of understanding resulting from ambiguity in the requirements, or because no one thought to perform certain kinds of tests. A lack of understanding can invade every step of the software development process, not just the program design. This is why the SWEBOK covers not just design practices, but the entire project life cycle from requirements gathering through testing and maintenance. With

increased understanding comes increased correlation between *intent* and *actuality*, and thus fewer *defects* – and this is what Software Engineering is really all about.

So, what should constitute a “chunk?” I defer to the body of practices themselves to decide that. My point is that we, as programmers, must not assume that we actually “understand” our programs, or that what worked before will ever work again. Such assumptions are an example of a “confirmation bias,” the tendency for people to focus on confirmation of hypotheses and to pay attention to confirmatory rather than disconfirmatory evidence (Nickerson, 1998). This means that we automatically (and subconsciously) give preferential treatment to evidence that supports our existing beliefs and discount evidence that is contrary to our beliefs. This bias encourages us programmers to interpret the faintest evidence, or even the mere “idea” that our program works, as “proof” that it works. We even create tests that are designed to demonstrate our program working and fail to think of tests that might show the contrary – a classic form of confirmation bias. Even the mere “thought” that we understand our programs may be interpreted as “proof” that we understand them. I hope that all programmers learn to seek objective, external confirmation that their software actually works as intended. I also see all Software Engineering practices as tools for reducing complexity, accepting responsibility for our beliefs, and overcoming our confirmation bias. The end result should be better software and happier customers (and hopefully happier programmers).

## ***Rebuttals***

I’ve posed this line of thought to many programmers and Quality Assurance people over the years, and responses vary from “you’re absolutely correct” to “you couldn’t possibly be more wrong.” Such comments say more about the commenter than they do about the ideas themselves. Technical arguments on this subject are of no interest to me – they are outside the scope of the language game for this paper. Technical arguments may simply be bruised egos trying to defend themselves.

One rebuttal proposed that defining the problem was the hardest part of the task, so I added the discussion about the ease of defining a task for my daughter. Defining the problem is easy. Defining it in such detailed terms that a computer program could be written is the difficult part, and it is only difficult because computers are so pathetically stupid that we have to think of every possible event that could happen. I cannot rely on a computer’s judgment if it encounters an unexpected situation. Imagine having to specify every single nerve stimulus to produce the motions for the requirement “clean your room,” including every possible error scenario and you will get an idea of why defining the problem is part of the problem. In fact, this leads to my next rebuttal point, which is a short explanation why I think most requirements documents are badly written.

## ***Writing Bad Requirements***

*"We see the world, not as it is, but as we are - or, as we are conditioned to see it."*  
- Stephen R. Covey -

By way of an explanation of the root cause of bad requirements, I have taken the liberty of quoting a section from the book Zen and the Art of Motorcycle Maintenance, by Robert Pirsig:

An "analytic" description is a classical platform from which one looks at a "thing" in terms of its underlying form. The motorcycle is a perfect subject for an analytic description, since it was invented by classical minds.

A motorcycle may be divided for purposes of classical rational analysis by means of its component assemblies and by means of its functions. If divided by means of its component assemblies, its most basic division is into a power assembly and a running assembly.

The power assembly may be divided into the engine and the power-delivery system. The engine will be taken up first.

The engine consists of a housing containing a power train, a fuel-air system, an ignition system, a feedback system and a lubrication system.

The power train consists of cylinders, pistons, connecting rods, a crankshaft, and a flywheel.

The fuel-air system components, which are part of the engine, consist of a gas tank and filter, an air cleaner, a carburetor, valves and exhaust pipe.

*[And so on and on and on...]*

A motorcycle may be divided into normal running functions and special, operator-controlled functions.

Normal running functions may be divided into functions during the intake cycle, functions during the compression cycle and functions during the exhaust cycle.

*[And so on and on and on ....]*

The first thing to be observed about this description is so obvious that you have to hold it down or it will drown out every other observation. That is: it is just duller than ditchwater. Yah-da, yah-da, yah-da, carburetor, gear-ratio, compression, yah-da, yah-da, on and on and on. That is the romantic face of the classic mode. Dull, awkward, and ugly.

But if you can hold down that most obvious observation, some other things can be noticed that do not at first appear.

The first is that the motorcycle, so described, is almost impossible to understand unless you already know how one works. The immediate surface impressions that are essential for primary understanding are gone. Only the underlying form is left.

The second is that the observer is missing. The description doesn't say that to see the piston you must remove the cylinder head. "You" aren't anywhere in the picture. Even the "operator" is a kind of personalityless robot whose performance of a function on the machine is completely mechanical. There are no real subjects in this description. Objects only exist that are independent of any observer.

The third is that the word "good" and "bad" and all their synonyms are completely absent. No value judgments have been expressed anywhere, only facts.

The fourth is that there is a knife moving here. A very deadly one; an intellectual scalpel so swift and so sharp you sometimes don't see it moving. You get the illusion that all those parts are just there and are being named as they exist. But they can be named quite differently and organized quite differently depending on how the knife moves.

For example, the feedback mechanism, which includes the camshaft and cam chain and tappets and distributor, exists only because of an unusual cut of the analytic knife. If you were to go to a motorcycle parts department and ask them for a feedback assembly they wouldn't know what the hell you were talking about. They don't split it up that way. No two manufacturers ever split it up quite the same way and every mechanic is familiar with the problem of the part you can't buy because you can't find it because the manufacturer considers it a part of something else.

It is important to see this knife for what it is and not to be fooled into thinking that motorcycles or anything else are the way they are just because the knife happened to cut it up that way. It is important to concentrate on the knife itself.

The "knife" in our case is our view of the product that we envision. However, we see this "product" not as it is, but as we are - specifically, instead of writing requirements from a user's perspective, we tend to write a *description* of the functions and components that *we* envision. The resulting requirements document, while technically correct, makes as much sense as the description of the motorcycle. In fact, we sometimes wait until *after* we have assembled a prototype before we write the requirements, which only exacerbates the problem. The field of Software Engineering has long ago created the concept of *Use Cases* as a way of capturing the real *idea* of the product; but as of yet, few companies seem to write them, because they are not analytical documents.

## ***Conclusion***

*There are never any problems - only viewpoints.*

I hope that my absolute pronouncements spur some controversy. Controversy is a sign that my comments have hit a nerve and may inspire reflection on our mental processes. It is natural to talk about "defects" as if they existed, but this is only an artifact of language. In Reality, defects do not exist – we only have misunderstandings. As programmers, we must realize that faith in our own understanding is the only source of defects.

Building a 4-foot tower requires little more than a level surface, a steady hand, and ten beer cans. However, building a 100-foot tower requires much more than simply 40 times as many beer cans. It requires a different kind of planning and construction entirely. It requires a completely different mindset. It requires *Engineering*. By continually examining and challenging our own practices, we should be able to ensure future successes and avoid past failures (or at least have more entertaining failures). Hopefully, our brains will not really get full; and if they do, there's always trepanning.

### ***Silicon Gods***

The Shaman sits in his  
small cave, drawing Arcane symbols;  
Cryptic numerals,  
Read by the Gods that they may do  
The Shaman's Will.

The Gods are implacable  
And insensate.  
They do not their own will, but ours.

Outside, the Faithful wait;  
Eager to receive their gifts from the  
Manufactured God.  
Hungrily, they wait to hear their God's Voice  
Speaking the Shaman's  
Broken words.

To our eternal dismay,  
The language of the Gods is not ours;  
So ever the Shaman sits,  
Patiently encoding human thought into the  
Cryptic tongue of the  
Blind Gods.

### Works Cited

- Boehm, B. Software Engineering Economics. New York, NY: Prentice Hall, 1981. Excerpt available online: <http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00sr008.pdf>
- Carnegie Mellon Software Engineering Institute. Cyclomatic Complexity. Online. Available: [http://www.sei.cmu.edu/str/descriptions/cyclomatic\\_body.html](http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html) (2008)
- Hall, E. T. (1976). Beyond culture. New York: Anchor Books.
- Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. The Psychological Review, 63, 81-97. Reprint Available: <http://www.musanim.com/miller1956/>
- Nickerson, R.S. (1998). Confirmation Bias: A Ubiquitous Phenomenon in Many Guises. Review of General Psychology, Vol 2 No. 2., 175-220. Reprint Available: <http://psy.ucsd.edu/~mckenzie/nickersonConfirmationBias.pdf>
- Pirsig, R.M. Zen and the Art of Motorcycle Maintenance. (1974, 1999) Harper Perennial Modern Classics.
- Rube Goldberg. Online. Available: [http://en.wikipedia.org/wiki/Rube\\_Goldberg](http://en.wikipedia.org/wiki/Rube_Goldberg). (2008).
- SWEBOK – Guide to the Software Engineering Body of Knowledge. (2004). IEEE Computer Society Professional Practices Committee. Available <http://www.swebok.org/>.
- Wittgenstein, Ludwig. Philosophical Investigations. (1950). Trans. Anscombe (Basil Blackwell, Oxford 1963). Republished Prentice-Hall (1979).

# Zen of System Control

Motor spins.  
Software runs.  
In a flash  
Broken.

The purpose of this document is not to rehash the well known mathematics of control algorithms, but to rehash the part of the problem that turns a three month project into a six year project while engineers frantically attempt to bring the system under control. Here are some example "Black Swans" from my own personal experience:

- Watching a 30 ton radar antenna oscillate atop its tower. Fortunately, someone cut the primary power to the block before the tower was torn to pieces. The problem was traced to a single transistor failure - resulting in an unstable control loop.
- Watching an 8-ounce motor rip itself apart, sending the bearings and springs flying. Not an actual hardware failure - a technician (not me) put the wrong resistor value in the PID control loop, sending the system into uncontrolled oscillations.
- Watching a hard disk armature sit locked on track for several minutes; then, without warning, instantly shoot forward so fast that the heads were ripped off the assembly (this happened 2 - 3 times daily). The problem was traced to a mechanical resonance within the bandwidth of the system - a mechanical design defect.
- If the hard drive heads weren't ripped off, the assembly still overshot the target cylinder so often that the design was unusable. The problem was traced to another critical design flaw - the head assembly had so little mass that the dynamics were dominated by the chordal friction of the ball bearings, which is inherently unpredictable (as opposed to static friction, which is the mean friction). This Black Swan killed the project and the company.
- A thermal loop that broke into oscillations after improvements in the heat block assembly. This reduced heat loss, but also increased the loop gain. The loop instability ruined the chemistry and rendered the improved system unusable.
- Watching an infusion pump motor refuse to move when commanded ... the PID loop increased the drive voltage ... nothing ... finally, after the PID loop had increased the drive voltage enough, the motor overcame its static friction and flew forward so fast that it overshot and caused the system to malfunction.

"Control" is an illusion for two fundamental reasons:

1. We can never understand a system so completely that every facet of every transfer function is completely and exhaustively known (e.g. reference the shift in transistors from "white-box" physical models to "black-box" two-port parameters).
2. Even if we could fully characterize a system, a control loop has feedback, gain, and a transfer function that is non-linear. As long as these three conditions exist, there is always the possibility of the system becoming uncontrollable (ref Chaos by James Gleick, or The Turbulent Mirror by John Briggs, or the work of Henri Poincarre.)

In truth, we can never have more than a mental model of how we *think* reality functions in our control loop. Reality wins - but only always.



## The System is “Deterministic”

Being “deterministic” isn’t enough for most real-time hardware control. A system is “deterministic” as long as the system is bound by causality in such a way that any state of the system is completely determined by prior states. The key point here is that:

### **Deterministic ≠ Predictable**

From the field of concrete mathematics we find definitions of recursive functions, which are by definition deterministic. The basic recursive function is:

$$\begin{aligned} T_0 &= \text{InitialValue} \\ T_n &= F(T_{n-1}) \end{aligned}$$

Knowing the initial value  $T_0$ , one can calculate  $T_1$ ,  $T_2$ , etc., to  $T_n$ . The downside is that every value has to be calculated until one finally reaches  $T_n$ . One goal of concrete mathematics is to find a solution to recursive functions, so that one can calculate any value of  $T_n$  without having to calculate all the intermediate values:

$$T_n = G(n)$$

Until one can do this, a control system equation cannot be designed. For example, the recursive equation for the Tower of Hanoi puzzle is:

$$\begin{aligned} T_0 &= 0 \\ T_n &= 2T_{n-1} + 1 \end{aligned}$$

Which creates the series 0, 1, 3, 7, 15, 31, 63 ... and resolves to the general form:

$$T_n = 2^n - 1$$

This takes the system from being simply deterministic to being predictable. However, not all recursive functions have such solutions; therefore not all deterministic systems are predictable. A system in chaos is a perfect example of a system that is deterministic but not predictable; and a system that is not predictable is not controllable<sup>3</sup>.

When trying to control a hardware system, the addition of factors such as mass, inertia, and friction complicates the problem of turning determinism into predictability. For example, although the OS is deterministic, variance in interrupt latency or processing time has the same effect as varying the loop’s phase delay, loop gain, and the loop’s Nyquist frequency, which can in turn cause instability in a P.I.D. control loop. In this example, being deterministic isn’t enough – the variance in latency must also be predictable.

## The "System" Is Invisible

One of the many interesting things that have come out of Chaos Theory is the realization that a system under control is not actually visible. Instead, the control signal is what we

---

<sup>3</sup> Formal mathematical proofs exist and are beyond the scope of this paper. For proofs, refer to the work of Edward Lorenz, Benoit Mandelbrot, or Mitchell Feigenbaum.

are actually observing. No matter how perfect our mathematical models of the system, they are still only models; they are Maya<sup>4</sup>.

As long as the System is in control, we see only our mathematical model – Maya. We never see the reality of the system itself until it becomes unstable. This is because there is no entropy in a stable system; entropy<sup>5</sup> only appears in an unstable system<sup>6</sup>. In the absence of entropy, we see only the behavior that is caused by our control signals. This is good, since we *want* the system's behavior to match our control signals and NOT the system's own characteristics. Were it to match the system's own characteristics, then even minor variance in things like motor friction or mass would affect the system's behavior.

*The ability to control a system is inversely proportional to the entropy in the system.*

The following set of examples shows how the system's actual nature only becomes visible when it becomes unstable. I have chosen a very simple equation that fits the three basic conditions required for Chaos:

1. Gain, in that  $G > 1$ .
2. Feedback, in that  $X_{n+1} = G \cdot f(X_n) + \text{offset}$ .
3. Non-linear, in that  $f(x)$  is not a linear equation.

Note that, without exception, all closed loop control systems fulfill all three of these conditions. One more thing worth noting it that, without exception, by the time a control loop is aware that the system has done “something,” it’s already over. The assumption is that the system is predictable and therefore past behavior is a perfect predictor of future behavior. However, this is Maya.

In the examples below, offset is set to zero and not changed. The initial setting  $X_1$  is set and not changed. Once the system becomes unstable, the initial setting is changed slightly to show how the system becomes sensitive to the initial conditions (also called “The Butterfly Effect<sup>7</sup>”). Two plots are given: 1) (amplitude, time) and 2) phase space ( $X_n, X_{n+1}$ ). Initially, both plots show a stable system. From the data, no information can be obtained about  $f(x)$ . However, once the system becomes unstable, the nature of  $f(x)$  becomes apparent in the phase space plot.

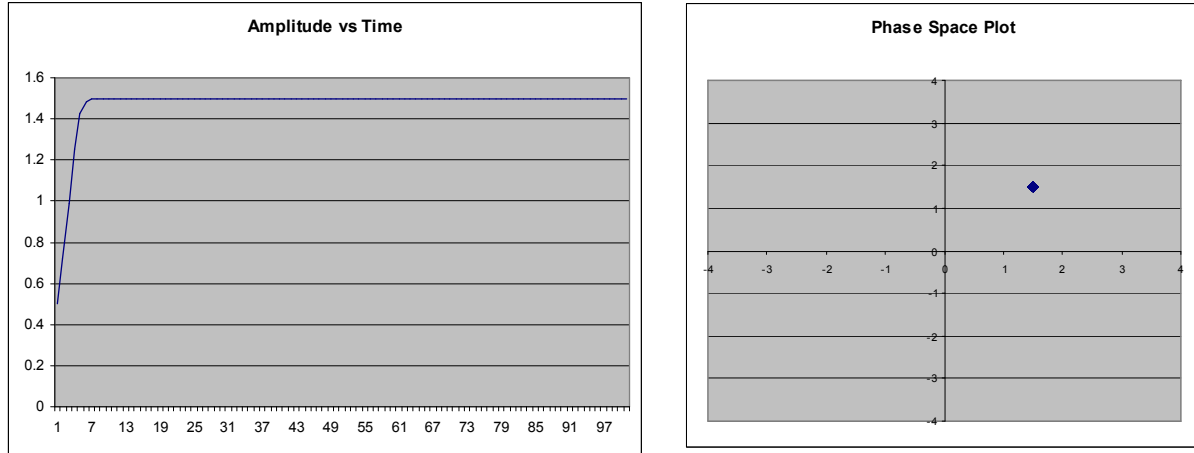
---

<sup>4</sup> Maya: Illusion. Maya is not false; rather, it reminds us that we do not experience the environment itself, but rather a projection of it, created by our senses and our thoughts. Maya is true in that we do indeed have perceptions, but untrue in comparison to the actual Reality that is being perceived. An example is the perception of fire, which at first glance appears to be something that is leaving the log. As a result of this perception, the release of Phlogiston was once thought to be the cause of fire. Until the invention of the Bell Jar, all scientific data was consistent with the Phlogiston theory.

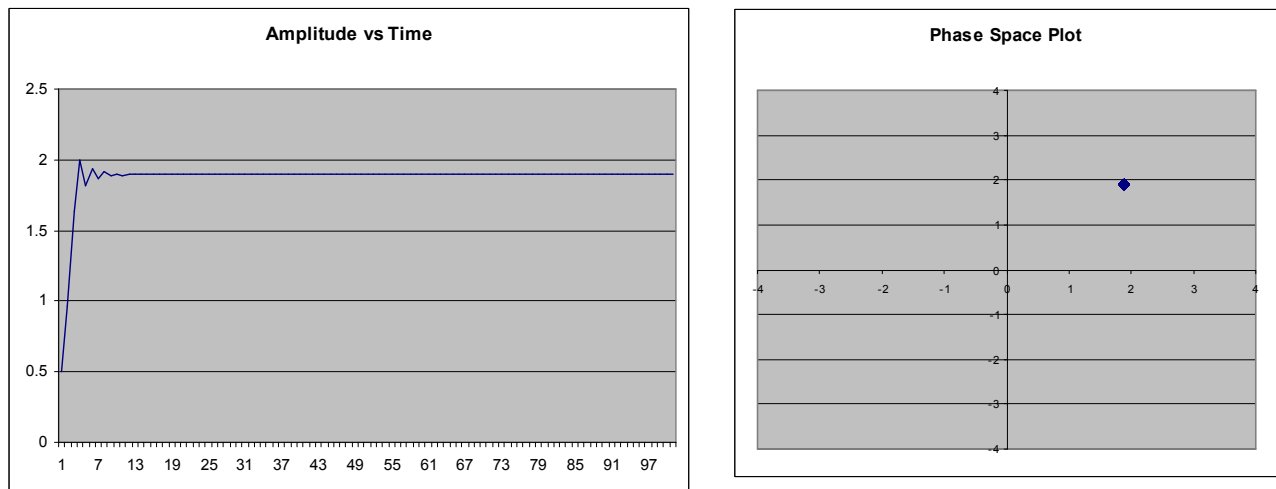
<sup>5</sup> Entropy - In this case, we use the generic definition: “A measure of the disorder or randomness in a closed system.”

<sup>6</sup> For more details and proofs of this, refer to the existing library on Chaos Theory, such as the book The Turbulent Mirror, by John Briggs.

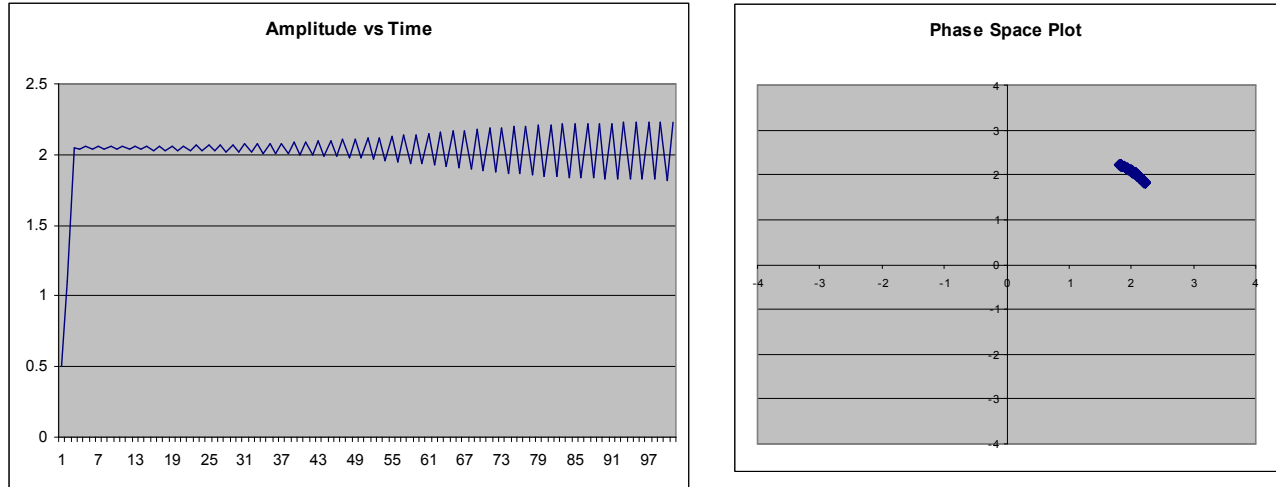
<sup>7</sup> In chaos theory, the Butterfly Effect is the sensitive dependence on initial conditions; where a small change at one place in a nonlinear system can result in large differences to a later state. The name of the effect, coined by Edward Lorenz, is derived from the theoretical example of a hurricane's formation being contingent on whether or not a distant butterfly had flapped its wings several weeks before.

*Figure 1 -  $G = 1.5$ ;  $X_1 = 0.5$* 

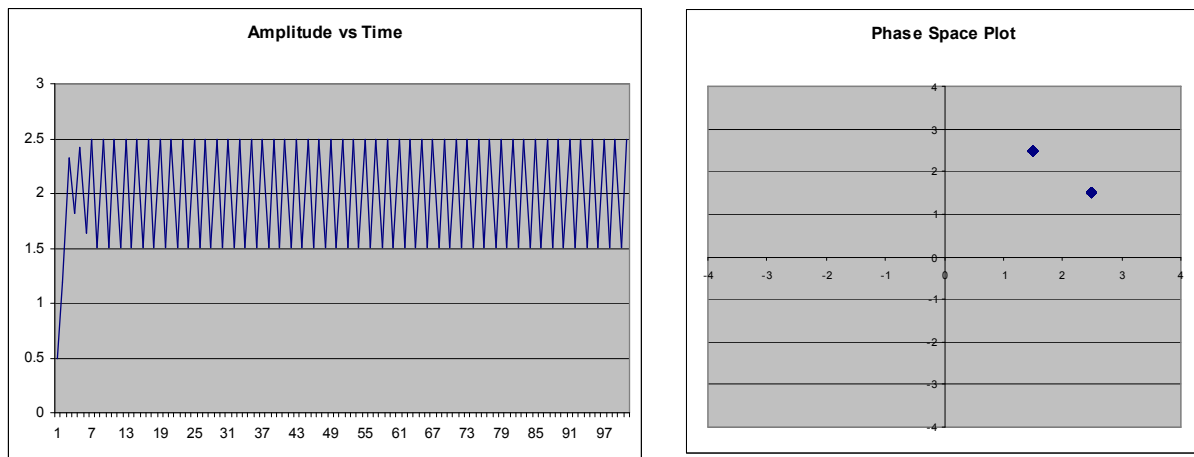
In figure 1, the system quickly becomes stable. Once stable, each succeeding point remains constant, so the phase space plot is a single point. Any  $X_n$  is now perfectly predictable and is a perfect representation of our control signal (in this case, the offset and gain portions of the equation).

*Figure 2 -  $G = 2$ ;  $X_1 = 0.5$* 

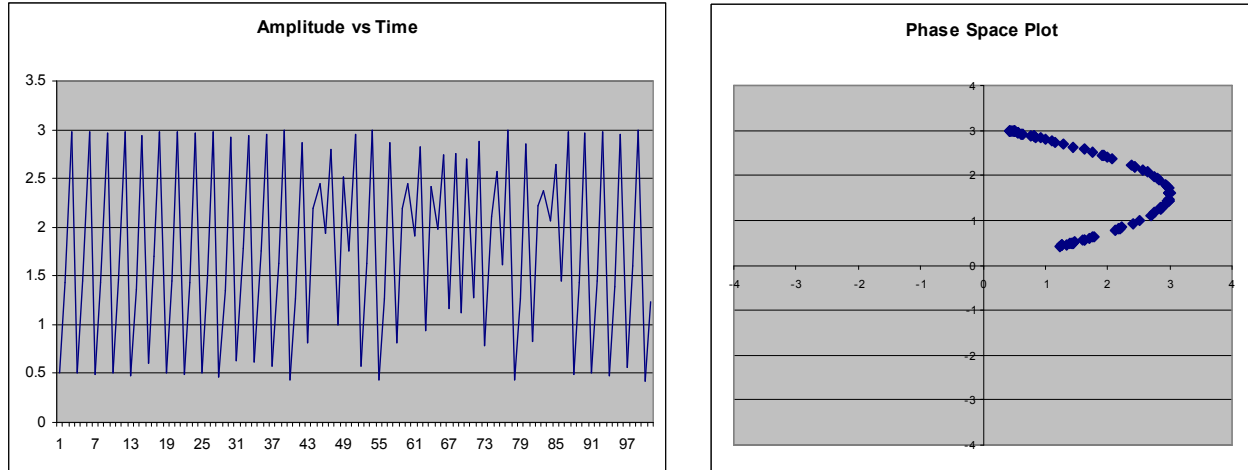
In figure 2 we see the beginning of overshoot, just a touch of under-damping for this much gain. It quickly settles and the system becomes stable. Phase-space plot is still a single point. The details of  $f(x)$  are not visible.

*Figure 3 -  $G = 2.3$ ;  $X_I = 0.5$* 

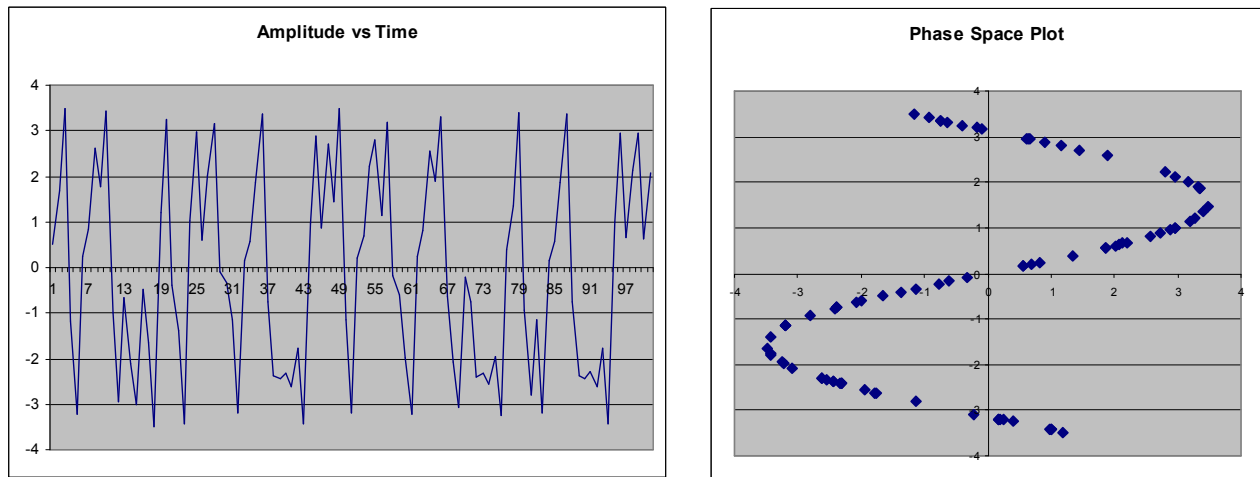
In figure 3, the gain is now high enough that the system breaks into natural oscillations. The phase space plot shows that the system oscillates within boundaries. Anyone who has tuned a PID loops knows that this information can be used to pick optimal values for the Proportional, Integral, and Derivative portions of the loop gain.

*Figure 4 -  $G = 2.5$ ;  $X_I = 0.5$* 

In figure 4 we see that the system is in complete oscillations between the two points noted on the phase space plot. However, the nature of  $f(x)$  is still not visible. (This is what happened in the thermal loop mentioned earlier. Improvements in the thermal assembly reduced the heat loss, which has the same effect as increasing the loop gain. The effect on the chemistry was project destroying.)

*Figure 5 -  $G = 3$ ;  $X_1 = 0.5$* 

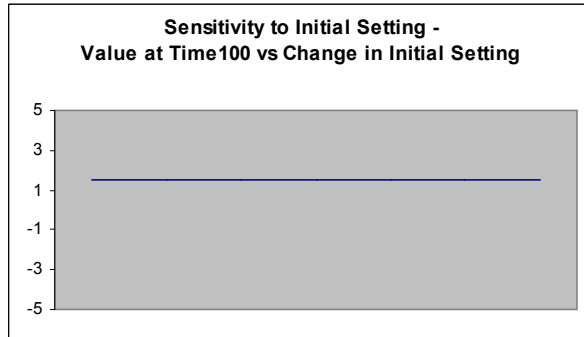
In figure 5 the system is in chaotic oscillations. There is now sufficient entropy in the system to make an estimate about  $f(x)$ .

*Figure 6 -  $G = 3.5$ ;  $X_1 = 0.5$* 

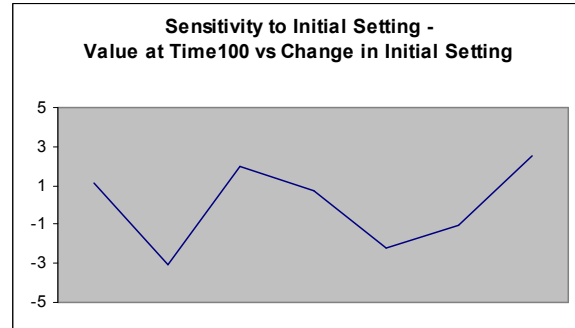
In figure 6 the system is so entirely out of control that the nature of  $f(x)$  is readily apparent - it is a sine function. Formal proofs exist to show that, given any value  $X_n$ , the only value that can be predicted is  $X_{n+1}$ . Contrast with the earlier systems under control, in which it was possible to know any  $X_n$  no matter how far in the future it lay - because the system was stable. However, a system in chaos is inherently unpredictable and therefore inherently uncontrollable, because there will always be a time/phase delay in reading  $X_n$  and in responding (think Plank constant and quantum physics - only the very next quantum instant is calculable).

*Figure 7 - The Butterfly Effect - from stable ( $G = 1.5$ ) to unstable ( $G = 3.5$ )*

$G = 1.5$ ;  $X_1$  varied from 0.50 to 0.57  
( $X_1$  increased in 2% steps)  
0% change in  $X_{100}$  from  $X_1 = 0.50$  to 0.57



$G = 3.5$ ;  $X_1$  varied from 0.50 to 0.57  
( $X_1$  increased in 2% steps)  
317,410% change in  $X_{100}$  from  $X_1 = 0.5$  to 0.57



The Butterfly Effect can be triggered by variance in tolerances - such as field strength of motor magnets, bearing friction (static and chordal), or thermal resistance. It also appears when signals are digitized, creating a minimum observable change. Cheap hardware amplifies the Butterfly Effect, possibly rendering hardware prototypes meaningless.

I refer the reader to the many fine books on Chaos theory, such as [The Turbulent Mirror](#), for a further description of this phenomenon and a pointer to the formal proofs. Richard Feynman and Henri Poincarre provided many of the formal mathematical proofs behind this behavior and the fact that, *until the system becomes unstable, its actual transfer function is not observable.*

## The Certainty of Uncertainty

The greatest problem is the difference between our mathematical model of what we *think* the reality is, and the Reality itself. (Assuming that we stay within the Nyquist frequency of the system. For the Nyquist frequency problem I refer the reader to the corpus of existing information about sampling theory.) The only question remaining is whether this difference is enough to kill the project. Early hardware prototypes are the only way to find out.

This also means that, as long as our system remains stable, we cannot even see the hardware issues that lay in wait to cause future problems, because we cannot actually see the hardware's real transfer function - we only see our model. The hardware's actual behavior is only visible as variance (such as variance in motor speed or delivery accuracy). Since we want our control algorithm, and not the hardware, to be the only visible behavior, we desire a system that cannot become unstable - and over the entire range of hardware components, including those not yet manufactured. However, in the absence of clairvoyance we cannot rely on our mathematical models to provide us with that information. After all, they are only *models* of what we *think* the real transfer functions are supposed to be and cannot account for every facet of every molecule in the system.

## There Are No Problems - Only Viewpoints

The hardware is always right. Our mathematical models are simply our mental models for describing our belief about reality - they are not Reality itself.

"As far as the laws of mathematics refer to reality, they are not certain, and as far as they are certain, they do not refer to reality." - Albert Einstein

## Common Sources of Problems

This is not meant to be an exhaustive list, but describes the problems that I've seen occur with sufficient frequency to be memorable.

- Variance in interrupt latency. Most common cause is a frequent need to disable interrupts for critical operations (such as Reader/Writer locks on critical sections). Another cause is variable execution time for machine language instructions, such as floating point operations.
- Variance in thermal resistance between heating/cooling element and the target. Causes variance in loop gain. Can cause loop instability, especially in systems with low thermal mass.
- Mechanical resonance frequency. This acts to store energy and introduce phase shifts. Can cause loop instability, especially in high mass systems.
- Static friction. Must be overcome before the motor moves; can result in excessive error signals and high overshoots, especially in low mass systems.
- Chordal friction in ball bearings. This is variance in friction caused by how ball bearings move and settle during rotation. Has the same effect as randomly varying the loop gain. Can cause loop instability, especially in low mass systems.
- Variance in sampling frequency. Most common cause is the real-time design of the software and the operating system. Any time variance between samples has the same effect as varying the system gain and changes the Nyquist frequency. Can cause loop instability.
- Inappropriate control algorithm. For example, PID is useful in constant velocity systems, but "fuzzy logic" is more appropriate for start-stop (bolus) systems.

## Conclusion

In summary, we can conclude that the most critical factors in systems control are:

1. Identify the hardware factors that are most subject to variance, especially in manufacturing and over time. Throw money at these issues to get better hardware. Software cannot overcome problems created by cheap hardware.
2. Identify the software factors that are most subject to variance, such as interrupt latency and instruction execution time. This becomes even more critical as more threads are added, each competing for certainty in time.
3. Hardware prototypes should be available as soon as possible.
4. Determine the actual frequency range (min and max) of the information needed to control the system. This sets the Nyquist frequency. Information beyond this frequency is aliased and results in system instability.
5. Control algorithms should be selected for their efficacy rather than ease of implementation. Control algorithms may need to be adaptive to cover the full range of system performance. For example, fuzzy logic is often a better choice than PID for control loops because the loop can be more adaptive.

6. Hardware variance over the life of the product can introduce critical, unforeseen problems, making boundary and life testing critical to a good system design.



# Zen of Validation

*“Conformity leads to mediocrity. To be different from the group or to resist environment is not easy and is often risky as long as we worship success.”*  
– Krishnamurti –

## Abstract

**Thesis:** Validation is pointless – and critical.

Once more, I’m making deliberately inflammatory statements that are designed to elicit an emotional response – being different as a way of generating controversy and discussion on a topic that is both important and basically dull. In this paper, I discuss why the concept of validation is both a necessary part of a product development process and why it is often the most useless part. So rather than simply “validate” something because “validation is good,” let’s rescue the concept of “Validation” and use it to good effect.

## “It’s ISO-Certified!”

My very first FDA sponsored training class for software validation was in 1994. It was hosted at our employer (who has not given permission to use their name, but if you’ve ever had general anesthetic then you’ve used their product, for which I wrote the production test software). The trainer had been trained at the FDA. The opening salvo from the teacher was this true statement:

“You can make life jackets out of lead and still have them ISO Certified. That’s because ISO (International Standards Organization) certifies that your processes and products are *consistent*. As long as your life jackets are made with a consistent and controlled process and materials, they can be certified!”

This led to a discussion of what the FDA was really trying to do with their new “Design Control” regulations.

## “It’s Validated!”

When we place the “Validated” label onto a product or process, we think it actually means something. So – let’s validate our lead life jackets. We will throw a non-swimmer into the water. They begin drowning and must be rescued. Send in Kevin Sorbo! Swimmer is rescued. We throw another non-swimmer into the water, but this time wearing our wonderful lead life jacket! They drown. Being dead, they do not require rescuing. Therefore, people who use our lead life jackets do not require rescue! We stamp “Validated” on our product and start selling them!

So what is it that we are really trying to say when we say “Validated?” Our expectations for this term are what gets in the way and transforms it from a valuable process into a meaningless waste of time. So let’s try to figure out just how we can show that it’s really safe to go into the water.

## Proofs of Correctness

In the 1970's the U.S. government began exploring the concept of "proofs of correctness" as an alternative to the lengthy software testing process. The idea was to use a formal analysis of the specification and validation of the tools to prove that the resulting program had to be correct. (Reference the ADA programming language.) The attempts were abandoned in the late 1990's after they were forced to admit that the problem was insolvable. Today it is difficult to find any references to "proofs of correctness," though the book The Way of Z, published in 1997, still attempts to use some of the "proofs of correctness" concepts to justify formal analysis of the specification as an *alternative to testing* (pg 11). Here is one example that shows why "proofs of correctness" do not work. This also illustrates the Turing Halting Problem, by showing a system that may or may not halt while trying to see if a statement is a theorem in the system.

### MIU system

The MIU system was created by Douglas Hofstadter, author of Godel, Escher, Bach (1979). It is a simple, formal system. You have one MIU axiom, four MIU rules, and an MIU goal.

#### MIU Symbols:

There are only three symbols in the MIU system: M, I, and U. Valid strings in the MIU system need not contain all three symbols, but will contain no other symbols.

#### MIU Axiom:

There is one axiom in the MIU system:

- You begin with the string MI.

#### MIU RULES:

There are four production rules in the MIU system:

1. Given a string of the form Mx, you may produce Mxx. ('x' is a string variable)
2. If a string ends in I, you may append U to it.
3. Given I I I in an MIU system string, you may replace the three I's with a U.
4. Given UU in a string, you may remove them.

#### MIU GOAL:

The MIU system has one goal:

- Get from MI to MU. (i.e. – is MU a theorem in the MIU system?)

This may be facilitated by using the MIU production rules.

**Example:** From Axiom MI, if we apply Rule 1 we get MII. If we apply Rule 1 again we get MIII. If we then apply Rule 2 we get MIIIIU. If we then apply Rule 3 we get MIUU. If we next apply Rule 4 we get MI (back to where we started). On so on...

I encourage you to go through and try to get to the goal before going to the next page. When you've given it a good try, go to the next page and read the solution.

## Solution

You can't reach the goal in the MIU system. See GEB for an interesting proof of this, as well as to see the hidden meanings in the MIU system. Here is one proof of why it's impossible to reach the goal:

1. The goal contains 0 I's (0 is a multiple of 3)
2. We begin with 1 I (which is not a multiple of 3)
3. Rules 2 and 4 do not change the number of I's.
4. Rules 1 and 3 will only result in a multiple of 3 I's.
5. It is impossible to go from a state without a multiple of 3 I's (start state) to a state with a multiple of 3 I's (goal state).

Therefore, it is impossible to reach the goal in the MIU system.

## One Meaning Behind the MIU System

This is a concrete example of the key concept behind the Turing Halting Problem, the Church-Turing Thesis, and Godel's Incompleteness Theorem. By following the rules within a formal system (rules 1 – 4 in this case) one can explore an infinite number of pathways and not yet reach the goal (MU in this case). The process never halts because it never reaches its goal, and just because it hasn't halted *yet* does not *prove* that it will *never* eventually halt. Therefore, one can never be certain that it is actually *impossible* to reach the goal (i.e. within the system, it is impossible to know if MU is actually reachable in the MIU system).

Godel gave a more general lesson: that in any formal system it is possible to state theorems (MU in this case) which can be neither proven nor disproven within the system (in this case, rules 1 – 4). One must go outside the system to determine the correctness of the theorem (as we did in the solution above, which uses rules of logic outside of the MIU system). In other words, we must rely on the meta-rules of the system and the meta-meta-rules of that system, and so on until we reach the level of natural language.

Its relevance to software engineering is that it shows that even if it is possible to create a program P (e.g. a compiler) that correctly translates source code S into executable code C, it is impossible to show that the program P is, in fact, correct in itself because you cannot submit it to anything to verify its correctness – the fact that any verifier has not revealed a defect may simply be its failure to detect the defect in P. In other words, the verifying system only halts when a defect is found. So if no defect is found yet, does the verifier keep looking until it does? When is the failure to halt taken as evidence that no defect exists? And to what program do you submit the verifier itself to verify that *it* is correct?

One can never know if the input for program P was itself correctly translated into a correct program. One must always go outside the system (e.g. by testing or code review) to determine if C is actually correct. Proving the correctness of the source S has the same problem – we cannot prove that it is a correct translation of the requirements. We must go outside the system (e.g. reviews, testing) to determine the correctness of S.

So, what does “Validated” actually mean?

## Parts is Parts

I have a thermometer. It's a nice thermometer. I take it to the calibration lab. They put it into an ice bath – it reads 0<sup>0</sup> centigrade. They put it into boiling water. It read 100<sup>0</sup> centigrade. Perfect! They slap a calibration label on it. Happily, I run outside and try to use it to measure the air pressure in my tires.

Good thermometer.

Bad intended use.

## Verification

The calibration lab “verified” that my thermometer was working correctly. How? They had a written specification describing what a thermometer should do. Through testing, they verified that the thermometer matched its specifications. (Gee – I can do that with lead life jackets! I just need a specification!) Unfortunately, nowhere did anyone say that an object meeting those specifications is actually good for anything.

## Validation

In empirical science it is impossible to prove anything. Proof is what you did in Geometry class. Only Analytic statements can be proven, and Analytic statements have nothing whatsoever to do with Reality. Analytic statements are statements about concepts (such as “All red houses are red”). Analytic statements are true a priori (in the absence of evidence).

By contrast, all statements from empirical science are Synthetic (true only if demonstrated by evidence); for example, “My house is red.” I can verify that by showing you my house. (It's yellow, so my statement would be disproven.) I will not attempt to recap the entirety of my statistics classes, but Synthetic statements *cannot be proven true – ever!* They can only be proven false by the demonstration of contrary evidence. That fact kept the Dark Ages dark for centuries, because it was taken as proof that science could not be trusted to invent the light bulb. However, with Sir Francis Bacon was born another idea – that although unproveable, it may be useful to accept a statement as true for practical purposes. This is the heart of “Validation.” It may or may not actually *be* true, but there may be practical value in *accepting* it as true.

Does it have to meet its specifications for there to be a practical value in accepting it? NO! This was the biggest surprise I had in the FDA sponsored training. Even if verification fails, there may still be practical value in accepting the product as safe, effective, and useful! They only expected an analysis to defend our position, which was often to remove the failed requirement as unnecessary or to be fixed in a later version, or to limit the intended use of the product.

**Verification** = Meets specification. Consistent. But, it might not be of any value at all.

**Validation** = It does the useful thing we intended. But, it might not actually meet specification.

Of course, consistency combined with usefulness makes for a much better production process, so our goal is to achieve both verification and validation. It also shows that, to be considered validated, we must have a statement of “Intended Use,” else we cannot know if it actually does anything useful. We still need to be able to say “We planned it that way!”

## To Validate or Not To Validate?

In 2001 I was part of a national teleconference seminar with the FDA for the upcoming release of their “General Principles of Software Validation; Final Guidance for Industry and FDA Staff” (officially released in Jan 2002). While most of the conference went smoothly, QA groups from several companies were at odds with the FDA over one key question: should software development tools be validated.

The QA answer from every company in the conference: “YES!”

The FDA answer: “NO and don’t even try!”

The FDA was quite emphatic about the “don’t even try.” The FDA rationale was a well thought out formal argument showing that, since there is no solution to the Turning Halting problem, the label “validated” means nothing what-so-ever when placed onto any product that either *generates* or *tests* software. The label “validated” only has meaning when placed onto the software *created by* the tools. However, by placing the “validated” label on the tool it gives a false sense of security and encourages a reversion to the defunct Dept of Defense attempt to enact proofs of correctness. The FDA cited their own Code of Federal Regulations, Chapter 21, Part 820 (Quality System Regulation):

### 21 CFR 820.30(g) Design Validation.

- Each manufacturer shall establish and maintain procedures for validating the device design.
- Design validation shall be performed under defined operating conditions on initial production units, lots, or batches, or their equivalents.
- Design validation shall ensure that devices conform to defined user needs and intended uses and shall include testing of production units under actual or simulated use conditions.
- Design validation shall include software validation and risk analysis, where appropriate.
- The results of the design validation, including identification of the design, method(s), the date, and the individual(s) performing the validation, shall be documented in the Design History File.

From § 820.3(z) – Definitions. Validation means confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use can be consistently fulfilled.

The phrase “defined user needs and intended uses” makes sense when applied to the product, but not to the tools used to create or test the product. The FDA also noted that the definition of Validation includes “can be consistently fulfilled;” however, it is possible that a latent defect in a tool may not affect one product but could affect a different product, so that part of the definition cannot be met by validating compilers or test tools (Turning Halting problem).

### 21 CFR 820.70(i) Production and process controls.

Automated processes. When computers or automated data processing systems are used as part of production or the quality system, the manufacturer shall validate computer software for its intended use according to an established protocol. All software changes shall be validated before approval and issuance. These validation activities and results shall be documented.

During the conference, this section of the regulation was cited by QA officials to support a need for compiler and tool validation, on the grounds that these tools were used to produce

the software. The FDA disagreed – the FDA says their regulation means that you must validate the software *created by* the compiler or tool – not the tool itself.

### 21 CFR 820.75(a) Process Validation.

Where the results of a process cannot be fully verified by subsequent inspection and test, the process shall be validated with a high degree of assurance and approved according to established procedures. The validation activities and results, including the date and signature of the individual(s) approving the validation and where appropriate the major equipment validated, shall be documented.

This was the FDA’s counter-argument to tool validation – that the output of any compiler, code generator, or automated test tool should always be examined and determined to be correct – it should never be blindly accepted just because a validated tool said it was good. To make matters worse, if the label “validated” is ever placed on the compiler and tools, then according to the FDA’s own regulation *we need never test the software at all!* And they already had vendors doing that – refusing to test on the grounds that they had validated the compiler! This is a legacy from the Dept of Defense’s earlier attempt at establishing proofs of correctness. In contrast, the FDA’s position was that a refusal to validate the tools would force companies to test their products, which is what they really wanted in the first place. They wanted the objective evidence that the product itself was safe, effective, and met its intended use. That evidence would include testing of any features provided by compilers, etc, so the evidence supports that the software produced by that specific tool chain is safe, effective, and meets its intended use.

## So – What Is “Validation”

The FDA said it best: “Validation means confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use can be consistently fulfilled.”

It’s up to us to decide, for any particular product, what constitutes good objective evidence for our particular system. Remember – Garbage In, Garbage Out. I may have a cool and validated unit test tool, but use it to write ineffective or outright defective unit tests. I will trust the tool, because it’s validated. Well ... really, to be in compliance I have to validate the unit test code that I wrote, not the tool. But ... how? If it seems to work, is it because the defect in my tool didn’t catch the defect in my code? For any given test result we have four possible conditions:

	Code has defect	Code has no defect
Test has defect	Test may pass or fail	Test may pass or fail
Test has no defect	Test fails	Test passes

So, if my test passes, why did it pass? What does it mean? This is an illustration of the fundamental premise behind all Synthetic statements – they cannot be proven, only disproven. Using the test success as proof of success is the formal logical fallacy of Affirming the Consequent. Only a test FAILURE means anything at all (Denying the Consequent, or Modus Tollens). It’s only the continued failure to get the tests to fail that constitutes a body of evidence to support a claim that the product is safe, effective, and meets its intended use. In other words, reject or fail to reject the Null Hypothesis by trying

to show that there is no relationship between the software's intended behavior and its actual behavior. This forms the foundation of the field of Hypothesis Testing.

So all testing should be focused not on trying to prove that it works, but in failing to prove that it doesn't. When we fail to prove that it's broken, then there is born a practical value in accepting that it works. The label "Validated" now means something.

## Labels Are For Cans

In the end, what does the label "Validated" really mean? Consider the following syllogism:

If a Product has feature set F Then that product will benefit society.

Product P has feature set F.

Therefore, Product P benefits society.

This is a modus ponens<sup>8</sup> argument and is therefore a formally valid argument. If the premises are true, then the conclusion must follow. The only question is how to determine if the premises are true. The premises are both Synthetic statements; therefore the truth of the premises can only be determined by a body of evidence and can only be accepted as true if it is practical to do so.

This is the whole point of validation: to decide if we will accept the truth value of the premises. Validation only has meaning when linked to the practical value of accepting the premises as true. It must be linked to a benefit to society. Anything less is verification, not validation. Even if we call a component or subsystem or process to be "validated," from the product perspective it is a "verification" and not a "validation."

So now we have the fallacy of equivocation. I may validate a production process, but from the product perspective it is verification. So the term "validation" has changed meaning. I believe that this fallacy of equivocation lies at the heart of the dispute over whether components and tools require validation. What they really require is actually verification, not validation. The only way to escape equivocation is to attach adjectives to the term "validation," such as "process validation" or "design validation." But these are not different adjectives for a single concept – they are compound nouns and are entirely different concepts. These are verifications because they support the following syllogism:

S is a specification for a class of Objects (could be processes, things, or whatever).

The properties of instance O match specification S.

Therefore, O is a member of the class S.

Testing an object or process to confirm that it meets specification is a verification process. It says nothing about what benefit may exist by having this object or process. We test software against its requirements. The requirements form the specification. If it passes, it only means that the software meets the requirements – not that software built to those requirements is of any real value. The most one can say is that, as a component, it is suitable for use in the larger system of which it is a part. Building upwards, component by component, we can use verification to determine if our components are suitable for use in the larger system. But that is still a verification process and not a validation. Calling it "validation" is equivocation.

---

<sup>8</sup> Modus Ponens:  $p \rightarrow q, p, \therefore q$

In the manufacturing environment, common types of verifications include “Installation Qualification,” “Operational Qualification,” and “Performance Qualification” (IQ/OQ/PQ). These may be components of a larger “process validation,” which in turn is part of a larger system validation. Again, from the product perspective, they are all verifications. I’d prefer to avoid equivocation about the word “validation” and call them what they really are. For example, “Tool Validation” would more appropriately be termed an IQ/OQ/PQ process, especially since a tool may be installed and used on many systems.

## Analyzing the Premises:

Back to the original syllogism:

If a Product has feature set F Then that product will benefit society.  
 Product P has feature set F.  
 Therefore, Product P benefits society.

There are some questions to answer about the premises before we can accept them as true. Let’s first consider the major premise:

**If a Product has feature set F Then that product will benefit society.**

1. Why must it benefit society? The answer is because that’s the only purpose for a medical industry in the first place. Therefore, only products that provide some form of benefit should be considered. The type of benefit could be improved health, lower cost, ease of use, or anything else that is deemed a benefit.
2. What is it about feature set F that creates the benefit? After all, I could say:  
 If the ocean is blue then Mars is inhabited by Smurfs.  
 The ocean is blue.  
 Therefore, Mars is inhabited by Smurfs.  
 This is clearly *non-sequitor*. The link between the feature set and the benefit must be established somehow.
3. Are there additional features that should be in F but that we didn’t think about?

Next, let’s consider the minor premise:

**Product P has feature set F.**

1. What makes us believe that the product we created actually has these features?
2. Does the product contain additional features that remove the benefit? For example, a random chance of exploding would be a significant safety risk.
3. Even if one particular instance of product P has these features, what about the n<sup>th</sup> product made on the production line? Does it also contain F and only F?

The *entirety* of the evidence supporting these two premises constitutes the Validation. This is why the FDA considers validation to be everyone’s responsibility – because the body of evidence required extends from the initial user needs to design to testing and to manufacturing. Only then does the label “Validated” begin to carry meaning.

*“When we yield uncomprehendingly to environment, any spirit of revolt that we may have had dies down, and our responsibilities soon put an end to it.” – Krishnamurti -*



# Zen of State Machines

Lamps emit light.  
Feathers are light.  
Therefore, lamps emit feathers.

## What's in a Name?

It's easy to create a definition for a word such as "light." It's easy to appropriately use that word in many different places. It's also easy to create the logical fallacies of equivocation<sup>9</sup> and amphiboly<sup>10</sup> when we are not precise and consistent in our usage of terms. So it is with the terms "state" and "event." These terms often seem to have precise meanings, but imprecision becomes apparent when trying to design a "state machine."

What is a "State Machine," really? According to a definition on Wikipedia, a state machine is "an abstract machine that can be in one of a finite number of states." I've rarely seen a more useless definition.

## Why Have "State Machines?"

Suppose I asked you to build me a "base"; what would you build?

- If we were at Wrigley Field, you'd build me a canvas bag for a baseball base.
- If we were at the Pentagon, you'd build me a military base.
- If we were in a lamp store, you'd build me a lamp base.
- If we were in a cosmetics store, you'd mix a cosmetic foundation base.
- If we were in a chemistry lab, you'd build me something that has an -OH hydroxyl, as opposed to an acid (e.g. sodium hydroxide).

The meaning of the term "base" is contained nowhere in the word itself - it only acquires meaning when combined with the context in which it is used. That context can be termed the system's "State." A "State" then becomes the context in which a set of data is given meaning. Until then, the data has no meaning.

There are several different ways of defining what constitutes a state machine. The way that makes the most sense to me is to contrast a state machine's "logic dominated" approach with its opposite: a "data dominated" approach.

---

<sup>9</sup> Equivocation: a word or phrase that is used in multiple places, but with different meanings, as in my opening syllogism.

<sup>10</sup> Amphiboly: a single sentence or phrase (as "nothing is good enough for you") that can be interpreted in more than one way.

## Data Dominated Approach

In this context it means that the system is dominated by the amount of data, rather than the decisions being made. Consider saving or reading a file - the amount of data is large and the number of decisions is small (open, copy, close, check CRC). In such a design, memory management may be the most important factor. Logic is simple can be accommodated with the usual array of for, if, and switch statements.

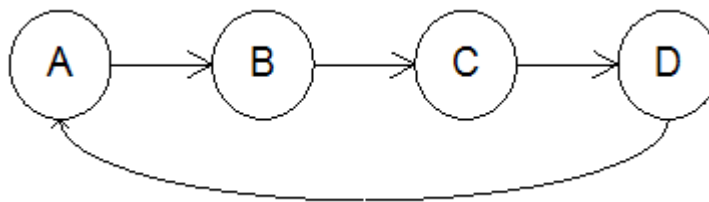
## Logic Dominated Approach

In this case, the amount of data is small compared to the array of decisions being made. These decisions may be arbitrary and it's the decision logic that dominates the design. The ability to code decision logic may be the most important factor.

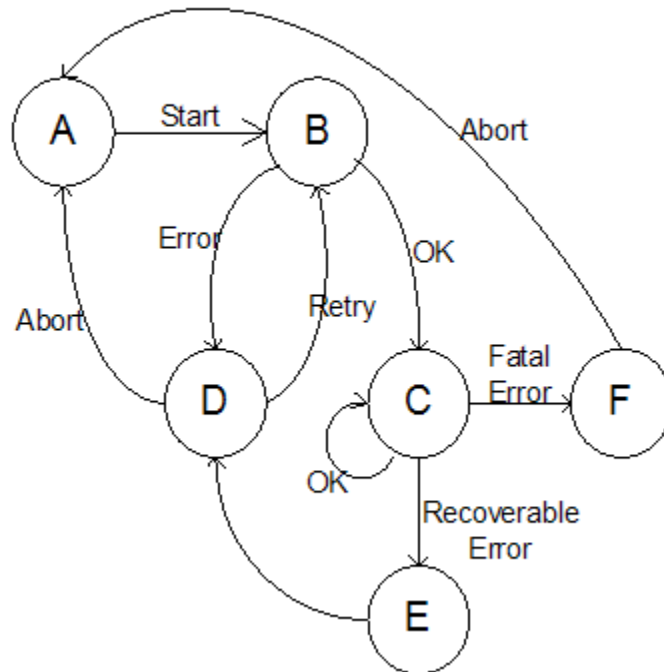
There are many ways of implementing a Logic Dominated Approach, such as a series of if-else statements. State Machines are simply one method of implementing a logic dominated system. State Machines have the advantage of readily representing arbitrary decisions with a single design architecture. State Machines can be described either by their structure or by their purpose.

### Common purposes of State Machines:

- **Acceptors** (aka Sequence Detectors or Recognizers): these accept a sequence of data and give a binary output (Yes/No) to answer whether the input is accepted by the state machine or not. Acceptors typically begin in a Start state, then switch to a Scanning state once the start of a sequence is detected. It then steps to a Final state (Accept or Not Accept) once a decision is made. Acceptors are common in lexical analysis and data transfer, as a way of determining if a valid token or frame of data has been received.
- **Transducers.** These generate output actions based on given inputs. These are used to control actions in a system. There are two kinds of Transducers:
  - 1) **Moore Machine.** The output depends only on the state. For example, a door may only have two commands: "open" and "close." When the state is "open" and a "close" command is received, the door closes.



- 2) **Mealy Machine.** The output depends on the current state and the incoming data. The next state also depends on the current state and the incoming data. For example, a motor in a "stopped" state will start when a "start" command is received. Once in the "running" state, it may stop if a "stop" command is received or if a fault is detected by a sensor.



So far, I've only considered the class of State Machines termed "Discrete Finite Automata" (aka Finite State Machine). These are characterized by their deterministic behavior, in that every state has exactly one and only one transition for each possible input. By contrast, in non-deterministic automata an input can lead to one, more than one, or no transitions for a given state. Push-Down Automata are one such form, which are often used in the analysis of regular expressions and in compilers. Non-deterministic Automata<sup>11</sup> will not be addressed in this document. This document covers only FSM's.

Table 1 – State Machine type by language type being parsed

Chomsky Grammar	Language	Example	State Machine
Type-0	Recursively enumerable	All formal grammars	Turing Machine
Type-1	Context-Sensitive	C#, C++, Java	Linear bounded Non-Deterministic Automata
Type-2	Context-Free	Assembly Language	Non-deterministic Pushdown Automata
Type-3	Regular	Regular expressions	Finite State Machine

<sup>11</sup> In general, the class of non-deterministic automata are used where prior decision information is required to create a complete context. For example, parsing a Type-1 grammar, which is a context-sensitive grammar.

## A Formal Definition for Finite State Machine<sup>12</sup>:

Flow charts are not state machines. Flow charts are tools for illustrating the flow of logic. However, that does not make the boxes "states."

A basic deterministic finite state machine or acceptor deterministic finite state machine is a quintuple  $(\Sigma, S, s_0, \delta, F)$ , where:

- $\Sigma$  is the input alphabet (a finite, non-empty set of symbols).
- $S$  is a finite, non-empty set of states.
- $s_0$  is an initial state, an element of  $S$ .
- $\delta$  is the state-transition function:  $\delta : S \times \Sigma \rightarrow S$   
(in a nondeterministic finite automaton it would be  $\delta : S \times \Sigma \rightarrow P(S)$ , i.e.,  $\delta$  would return a set of states).
- $F$  is the set of final states, a (possibly empty) subset of  $S$  (e.g. Accept or Not Accept).

$\delta(q, x)$  does not have to be defined for every combination of  $q \in S$  and  $x \in \Sigma$ . For example, if an open door receives an "open" command, it could either be treated as an "error" condition or simply ignored as Not Applicable."

A finite state transducer is a sextuple  $(\Sigma, \Gamma, S, s_0, \delta, \omega)$ , where:

- $\Sigma$  is the input alphabet (a finite non empty set of symbols).
- $\Gamma$  is the output alphabet (a finite, non-empty set of symbols).
- $S$  is a finite, non-empty set of states.
- $s_0$  is the initial state, an element of  $S$ . In a nondeterministic finite automaton,  $s_0$  is a set of initial states.
- $\delta$  is the state-transition function:  $\delta : S \times \Sigma \rightarrow S$ .
- $\omega$  is the output function.

If the output function is a function of a state and input alphabet ( $\omega : S \times \Sigma \rightarrow \Gamma$ ) that definition is a Mealy machine. If the output function depends only on a state ( $\omega : S \rightarrow \Gamma$ ) that definition is a Moore machine.

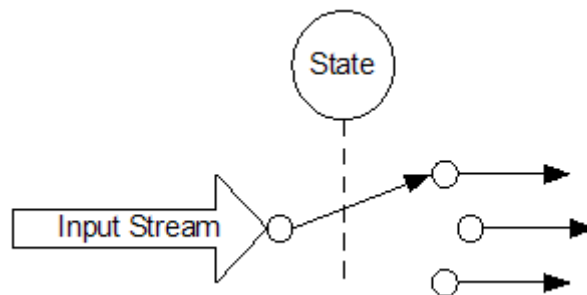
---

<sup>12</sup> Discrete Automate or Finite Automata or Deterministic Finite Automata are all considered synonyms for Finite State Machine.

## "State Machine" vs "Lexical Analysis"

Lexical Analysis is the process of analyzing an input sequence of characters and parsing them into tokens. Lexical Analyzers generally contain an Acceptor type of state machine to signal completion of a token. However, Lexical Analysis itself is still largely a data dominated approach, with the state machine being merely a component. Tokens are typically set aside for later processing.

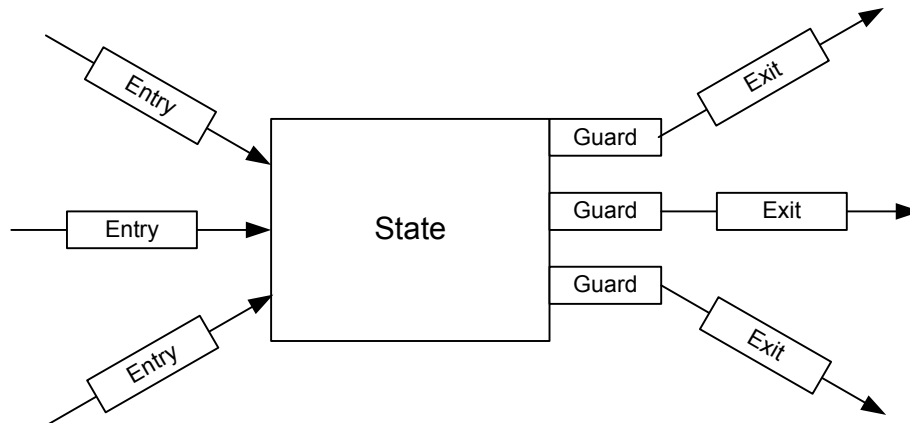
For example, an input datagram may contain a frame header, a payload, and a CRC. The Acceptor would be used to separate these tokens, passing the payload to a receiving application only after it has accepted the complete datagram.



## Implementing State Machines

There are about as many ways to implement state machines as there are programmers. However, some work better than others. Some convenient definitions are:

<b>State</b>	The context in which input data is interpreted.
<b>Data</b>	Input data that is processed according to the current state.
<b>Transition</b>	The action of changing from one state to another.
<b>Event</b>	Anything that triggers a transition. This may be triggered by something external (like a sensor) or by the state itself in response to the input data. For that reason, an "Event" is a semantically different concept than "Data." In the Mealy State Machine, an Event causes a Transition.
<b>Entry</b>	Function called when a Transition causes entry to a State.
<b>Guard</b>	Function called to allow or disallow an Event from causing a Transition.
<b>Exit</b>	Function called when a Transition causes a State to change.



The sequence to fire an event is then:

Event occurs.

The tuple {Current\_State, Event} determines the Transition.

IF Guard Accepts the Transition, THEN

Current State's Exit function is called.

The State is changed.

The new State's Entry function is called.

END IF

Not all State Machines will require Guards, Entry, or Exit events.

The primary considerations when designing a state machine are:

- 1) Readability. Arbitrary logic is not the easiest thing to read. 30 pages of nested switch and if-else statements are not readable.
- 2) Maintainability. Closely linked to readability. State machines, being arbitrary logic, are subject to frequent change. To be maintainable, it helps to code an architecture that abstracts the State Machine structure from the state functions themselves.
- 3) Handling simultaneous or unexpected events.
- 4) Initializing on start up (setting the initial state).
- 5) Must be an always-halting Turing Machine<sup>13</sup>. Since the Turing Halting Problem is unsolvable, this dictates the need for a clear and understandable design for the underlying architecture of the state machine itself, independent of the state functions being performed. The State Design Pattern<sup>14</sup> is one such example of an architecture that abstracts the state machine concept from the state functions.

---

<sup>13</sup> Ref "Halting Problem" - [http://en.wikipedia.org/wiki/Halting\\_problem](http://en.wikipedia.org/wiki/Halting_problem)

<sup>14</sup> Ref "State Design Pattern" - [http://sourcemaking.com/design\\_patterns/state](http://sourcemaking.com/design_patterns/state)

## Defining State Machines

State Transition Tables are probably the most common method of defining state machines. While graphical drawing tools can be used, the tools can often obscure details. State Transition Tables can also be directly coded, allowing the creation of a generic state machine architecture. For example:

Current State → Input ↓	State A	State B	State C
Input X	...	State A	...
Input Y	State B	...	State A
Input Z	...	State C	...

## Bad State Machines

"Bad" is a pejorative and is used to identify state machines that, while they function correctly, should never have been coded that way. Changes have a high likelihood of introducing unexpected defects. Here are some ways to make "bad" state machines:

- 1) Use one long nested sequence of if...else statements.
- 2) Use one long nested set of switch() statements (maybe add some nested if...else's in the cases).
- 3) Copy-and-paste code (I've seen an automated code generator that uses copy-and-paste for entry/exit code to each case in the huge switch statement).

## Good State Machines

"Good" means that the code is readable and maintainable. If the state machine is small enough, then perhaps a switch() statement will be quite readable. However, that should only be considered for very small state machines.

The State Design Pattern is probably the best example of how to implement a state machine. If using the "C" language, the pattern can still be used as a guide by either creating your own V-Tables or by creating a State-Event table. State-Event table is very maintainable, though it takes a little extra work to accommodate entry/exit/guard conditions.

Using a "C" table for the above state table (an NA state and event is defined for ease in deciding if a state change occurs – it ensures an always-halting Turing Machine):

```
typedef enum STATES { StateNA, StateA, StateB, StateC };
typedef enum EVENTS { InputNA, InputX, InputY, InputZ };
typedef struct EVENTTABLE { STATES CurrentState,
                           EVENTS Event,
                           STATES NextState };
```

```
struct EVENTTABLE EventTable[ ] = {  
    StateA, InputY, StateB,  
    StateB, InputX, StateA,  
    StateB, InputZ, StateC,  
    StateC, InputY, StateA,  
  
    StateNA, InputNA, StateNA // Table terminator  
};
```

To process an event, scan the table to find a match for the current state and event, then retrieve the next state from that table entry. To accommodate entry, exit, and guard conditions, the EventTable can include function pointers to those functions. If done in an object-oriented language, such as C++, each state can own its own table, rather than having one long table for the entire state machine.

The “NA” states and events provide a positive way of ignoring something while guaranteeing an always-halting machine, as an alternative to creating a comprehensive table that covers every {State, Event} combination (which could be a very large number).

## References

Chomsky Hierarchy. “Formal Grammars”  
[http://en.wikipedia.org/wiki/Chomsky\\_hierarchy](http://en.wikipedia.org/wiki/Chomsky_hierarchy)

Wikipedia - [http://en.wikipedia.org/wiki/Finite-state\\_machine](http://en.wikipedia.org/wiki/Finite-state_machine).

Rebecca Wirfs-Brock and Brian Wilkerson. Responsibility-Driven Design.



# Zen of Alarms

*“It is not the answer that enlightens, but the question.”* – Eugene Ionesco

Software is not clairvoyant – it is unable to “know” the current condition of the hardware on which it is running. Instead, it must rely on information provided to it by hardware sensors and hope that the information is sufficient to give the engineers the ability to create an algorithm that is capable of determining the current condition of the hardware.

The purpose of this document is simply to revisit the fundamental logic behind the concept of using data inputs to detect abnormal conditions in the system. The goal is to maintain perspective on what is and is not within the realm of possibility for detecting abnormal hardware conditions. Software defect investigations over the last several years have led me to believe that it is appropriate to revisit the fundamental issues before going further into new projects. So be prepared to see material that you already know, but possibly presented in a different way so as to refocus our efforts.

Abnormal conditions such as data corruption may have hardware causes, but are essentially software in nature and are outside the scope of this discussion. For purposes of this discussion abnormal conditions are real time events, such as:

- Air in a fluid line.
- Occlusion of a fluid line.
- Lack of motor movement.
- Sensor failure.

## The Fundamental Problems

The fundamental problems in detecting abnormal conditions are (in increasing order of severity and difficulty of overcoming):

### Order of Execution

Real-time operating systems are not capable of guaranteeing order of execution of different threads. This means that detection methods that cross thread boundaries (and they undoubtedly will) are constrained to have variable and unpredictable latencies between the time a sensor is read (Time A) and the time the information is processed and a decision made (Time B). This makes it possible for the sensor reading to be different at Time B such that a different decision would have been made. The maximum latency determines the Nyquist<sup>15</sup> frequency for this decision. Events occurring faster than this frequency may produce false results (either failure to alarm, false alarm, or an extraneous alarm other than the intended alarm).

---

<sup>15</sup> Nyquist Frequency is half the sampling frequency of a discrete signal processing system. Signal frequencies higher than the Nyquist frequency will encounter a "folding" about the Nyquist frequency, back into lower frequencies. For example, if the sample rate is 20 kHz, the Nyquist frequency is 10 kHz, and an 11 kHz signal will fold, or alias, to 9 kHz.

## The Certainty of Uncertainty

Similar to the Heisenberg Uncertainty Principle, the hardware may not allow the system to *simultaneously* obtain multiple pieces of information. This means that if two or more sensor inputs are used for a decision, the time between the two samplings also affects the Nyquist frequency of the information being used for the decision. Events occurring faster than this frequency will produce false results (either failure to alarm, false alarm, or an extraneous alarm other than the intended alarm).

## Insensible Sensors

The use of sensors to measure system behavior implies a correlation between the *intended behavior being measured* and the sensor's output. I emphasize the implied correlation with what the engineers *intend* to measure, because there is a separate correlation between what we *intended* to measure and what we are *actually* measuring. This correlation can be affected by factors such as sensor technology and sensor position, as well as the assumption that measuring A gives us usable information about B. For example, we could assume that measuring changes in motor position gives us information about flow rate. However, a leak in the line will invalidate this assumption.

It may be that the sensors themselves are unable to obtain the right kind of data for making the appropriate decision. This can be a limit on the sensor, or on the limits of science itself. Many sensors use indirect measurement to give an indication of the system's behavior. For example, many air sensors use ultrasonic transducers, relying on the change in signal level to indicate air vs fluid. However, this assumes a predictable and monotonic relationship between the ultrasonic transmission and the fluid level under all possible fluids and conditions.

Indirect measurement carries ambiguity in the information, which can result in inappropriate decisions. Indirect measurement is also the greatest driving factor behind using multiple sensors, as a way of improving the correlation between the actual system behavior and the measured data. Multiple sensors may even be needed to monitor the condition of the sensors themselves.

## Limitations of Observation

The single greatest limitation of a system's ability to detect abnormal conditions is simply this: we are *assuming* that data A indicates the presence (or absence) of condition B. We are never able to directly detect condition B itself. This places constraints on the system's ability to detect abnormal conditions. (For purposes of this discussion we will assume that, once detected, the system is able to take an appropriate response.)

The most basic limitation of observation is the assumption that for any possible detection algorithm there is a function  $f(x)$  (where  $x$  is the sensor input), which is a *surjective function*; that is, for any input  $x$ , there is one and only one possible  $y$  (alarm or no alarm) AND if for any value of  $y$  (alarm or no alarm), there is an input value  $x$  which will produce that result. An example Leak Detection algorithm for one system provides a good example of this limitation.

The exact force required to produce a leak varied widely by manufacturing tolerances, as it was a molded part. However, the leak detection algorithm in question was only a surjective function over a certain range of devices - those whose force was within a specified range.

Outside that range, it became impossible to distinguish a leak from a non-leak based on the pressure value (there was no value of  $x$  that could produce a  $y$ -value of alarm). Within the given pressure range, the algorithm was effective at the 95% confidence level; however, outside that range the algorithm was unable to generate an alarm, even if the component was entirely leaking all of its fluid.

To further complicate matters, testers often tested the algorithm by cutting the part to produce a leak. However, in the original investigation that led to the algorithm, the leaks were traced to a jammed inlet valve - not a hole. During the development of the leak detection algorithm, a “leak” was defined as “a jammed inlet valve.” But later testers simply cut the tubing to create a leak; a “leak” was redefined as “a cut line.” While both conditions can produce a leak, it may not be true that “a jammed inlet valve” = “a cut line.” For purposes of testing this algorithm, a cut line can be considered a “leak” if and only if it is equal *salva veritate*<sup>16</sup> to a jammed inlet valve. This equality must first be verified in testing before a cut line can be used. Without such verification, a failure to alarm can just as well be used to demonstrate that a cut line is *not* equal to a leaky inlet valve. So failures to alarm in these tests could not be used to reject the efficacy of the algorithm.

## The Logic of Alarm Detection

For any particular abnormal condition, we define the following terms:

- Cab This is the abnormal condition that the sensor(s) are trying to detect. This is a binary condition – it either exists or it does not exist. The inverse  $\neg$ Cab means that the abnormal condition is not present. NOTE: This must comply with the Laws of Non-Contradiction and the Excluded Middle.
- d One set of data from the sensors, which are used to make a decision. Each data set  $d$  may be vectors, scalars, sets of vectors, etc. It may represent a single point in time or an interval of time.
- D Set of all data sets from all sensors. Essentially, all permutations of all signals from all sensors for all devices for all conditions - forever.  
 $\forall(d)(d \in D)$ .
- B Set of data  $b$  such that  $\forall(b)[(b \in D) \wedge (b = \text{Cab})]$ . (aka “Bad”)
- G Set of data  $g$  such that  $\forall(g)[(g \in D) \wedge (g = \neg \text{Cab})]$ . (aka “Good”)
- An( $d$ ) A specific algorithm ( $A_1$ ,  $A_2$ , etc.) that accepts input data  $d$  and determines if the abnormal condition exists; i.e. determines if  $d$  is in set B or G. The output is always a binary value  $\in \{\text{Alarm}, \neg \text{Alarm}\}$ .

From these definitions we create the following axiom, which must be maintained for the detection to be effective:

**Axiom1:**  $\forall(d)(d \in D) ( [(d \in B) \wedge \neg (d \in G)] \vee [\neg (d \in B) \wedge (d \in G)] )$   
(in natural language: All data sets in  $D$  are either in set B or G, but not in both or neither. This axiom can only be established by the sensors.)

---

<sup>16</sup> (Latin, ‘saving the truth’) Two expressions are intersubstitutable *salva veritate* if the result of substituting one for the other always preserves the truth-value of any sentence in which they are used.

For any given data set  $d$  we then have four possible conditions:

	$(d \in B)$	$(d \in G)$
$An(d) = \text{Alarm}$	True Positive	False Alarm
$An(d) = \neg \text{Alarm}$	Failure to Alarm	True Negative

These conditions are represented as a series of Research Hypotheses, which cannot be proven deductively but that can be verified empirically at a given confidence level:

- Hypo1:**  $\forall(b)\exists(An)(An(b) = \text{Alarm})$   
(True Positive – all “bad” data produces an alarm.)
- Hypo2:**  $\forall(g)\forall(An)(An(g) = \neg \text{Alarm})$   
(True Negative – no “good” data produces an alarm.)
- Hypo3:**  $\forall(b) \neg \forall(An)(An(b) = \neg \text{Alarm})$   
(Absence of the Failure to Alarm condition – no “bad” data fails to alarm.)
- Hypo4:**  $\forall(g) \neg \exists(An)(An(g) = \text{Alarm})$   
(Absence of the False Alarm condition – no “good” data produces an alarm.)

Meeting Hypotheses 1 & 2 ensures no failures to alarm or false alarms; however, since these are hypotheses and not theorems they are subject to Type I and Type II errors (Failure to Alarm and False Alarm respectively). In other words, we can never be *certain* that Axiom 1 has actually been met or that the algorithm is effective, because the axiom itself can only be tested empirically – hence we can only have a level of confidence in Axiom 1. It is only stated as an axiom to illustrate the fact that, if not met, it is impossible to meet any of the research hypotheses, even in theory.

The reality is, we only achieve a level of confidence in Axiom 1 as the *result* of testing our research hypotheses. This means that, in practice, we can never be confident that the fundamental system of sensors is adequate until after we have already reached a level of confidence in our algorithms. Subsequent field results could always demonstrate a new data set that meets one of the following Exclusion conditions:

- Excl 1:**  $\exists(b) (b \in G)$   
(in natural language, there are some abnormal conditions that still produce good looking data. E.g. wrong sensor choice, etc.)
- Excl2:**  $\exists(g) (g \in B)$   
(in natural language, there are some normal conditions that still produce bad looking data. E.g. wrong sensor choice, uncalibrated, etc.)
- Excl3:**  $\exists(d) [(d \in B) \wedge (d \in G)]$   
(in natural language, there are some data sets that can exist in both “good” and “bad” conditions. This is a violation of Axiom1.)
- Excl4:**  $\exists(b) \neg \exists(An)(An(b) = \text{Alarm})$   
(in natural language, there are some “bad” conditions for which no algorithm exists that can see it as an Alarm condition.)

## Impact of Multiple Alarm Algorithms

In some cases, it may not be possible for a single algorithm to make a decision; it may have only a low confidence level in declaring an alarm. Note that this is not the same as several algorithms combining to produce a single decision – that would still be considered a single algorithm. Rather, multiple algorithms exist when two or more algorithms exist *either one of which can declare the alarm by itself*.

Revisiting Hypotheses 1 & 2 (True Positive and True negative), what we have is this:

- $P_n(\text{Alarm} \mid b)$   $P(\text{An}(b) = \text{Alarm})$   
(in natural language: the probability of Algorithm n producing a True Positive, given data set b.)
- $P_n(\neg\text{Alarm} \mid b)$   $P(\text{An}(b) = \neg\text{Alarm}) = (1 - P_n(\text{Alarm} \mid b))$   
(in natural language: the probability of Algorithm n producing a Failure to Alarm, given data set b.)
- $P_n(\text{Alarm} \mid g)$   $P(\text{An}(g) = \text{Alarm})$   
(in natural language: the probability of Algorithm n producing a False Alarm, given data set g.)
- $P_n(\neg\text{Alarm} \mid g)$   $P(\text{An}(g) = \neg\text{Alarm}) = (1 - P_n(\text{Alarm} \mid g))$   
(in natural language: the probability of Algorithm n producing a True Negative, given data set g.)

We can see that, since  $P_n$  will always be a positive number, adding more algorithms will increase the probability of a True Positive and, conversely, reduce the probability of a Failure to Alarm (which is the most common reason for adding multiple algorithms). In this case, the Alarm is given if any one algorithm produces an Alarm.

$$P(\neg\text{Alarm} \mid b) = (1 - P_1(\text{Alarm} \mid b)) \cdot (1 - P_2(\text{Alarm} \mid b)) \dots \cdot (1 - P_n(\text{Alarm} \mid b))$$

$$P(\text{Alarm} \mid b) = 1 - P(\neg\text{Alarm} \mid b)$$

Since  $P(\neg\text{Alarm} \mid b)$  gets progressively smaller as more algorithms are added,  $P(\text{Alarm} \mid b)$  grows larger. However, adding more algorithms also increases the False Positive rate and reduces the True Negative rate:

$$P(\neg\text{Alarm} \mid g) = (1 - P_1(\text{Alarm} \mid g)) \cdot (1 - P_2(\text{Alarm} \mid g)) \dots \cdot (1 - P_n(\text{Alarm} \mid g))$$

$$P(\text{Alarm} \mid g) = 1 - P(\neg\text{Alarm} \mid g)$$

Since the True Negative rate  $P(\neg\text{Alarm} \mid g)$  gets progressively smaller, the False Alarm rate  $P(\text{Alarm} \mid g)$  grows larger. This makes the False Positive rate of an algorithm more critical than it may seem at first glance. Even if a single algorithm is not effective over the entire range of B, multiple algorithms should only be added if they each have low enough False Positive rates that the overall performance is an improvement. Otherwise, a single algorithm with a high enough False Positive rate will adversely affect the entire system. This also shows that False Positives cannot be reduced by adding algorithms, but rather by increasing the confidence level of existing algorithms, possibly by adding more sensor data or with additional calibration to improve the confidence of the sensor readings.

## Validity of Indirect Measurement?

Ultimately, the root problem is simply that the use of sensors to detect abnormal conditions is not valid under formal logic, making it an empirical problem whose effectiveness must be verified by observation. In some cases, the indirect measurement may be several levels removed from the actual abnormal condition. Consider the problem in formal logic:

$p \rightarrow q$  (in natural language,  $p$  implies  $q$ , or If  $p$  Then  $q$ )

" $p = \text{true}$ " represents the abnormal condition Cab. " $q$ " represents the consequent (sensor data). The sensor data is the consequent and then implies the presence or absence of  $p$ . In other words:

1.  $p \rightarrow q$   
 $q = \text{true}$   
 $\therefore p = \text{true}$

2.  $p \rightarrow q$   
 $p = \text{false}$   
 $\therefore q = \text{false}$

Unfortunately, these are both formal logical fallacies - 1) error of affirming the consequent, 2) error of denying the antecedent. This means that the presence of  $q$  is only evidence that  $p$  *may* be true. In addition, due to the sensor itself, the relation  $p \rightarrow q$  may not even be true for all possible abnormal conditions being sensed. Therefore, the absence of  $q$  is only an indication that  $p$  *may* be false. There may be causes of an abnormal condition for which the sensor is not adequate. Finally,  $p$  may not be the only condition that makes  $q = \text{true}$ . "If  $p$  Then  $q$ " is not the same as "If and Only If  $p$  Then  $q$ " ( $p \leftrightarrow q$ ).

At best, sensor data can only be taken as a probability that condition  $p$  either exists or does not exist. Only empirical testing can reveal the level of confidence in an algorithm's conclusion about the presence or absence of the abnormal condition.

## Parts is Parts - or Not

Notwithstanding the above, the greatest challenge is not the designing of a system that meets the requirements of Axiom 1 and Hypotheses 1 - 4; rather, the greatest challenge is designing a system (including the algorithms) for which these statements will hold true for all situations, *including those not yet seen*, and for all hardware components, *including those not yet manufactured*.

Despite the best attempts to cover the range and combination of component tolerances and the anticipation of all possible future events, tolerances will drift over time and unforeseen events will occur. We can only have an unknown level of confidence that our data sets are a fair sampling of all possible data sets that will occur in every instrument over the life of the product. Events may occur for which the existing hardware or algorithms are unsuited or even antithetical.

## Laws of Non-Contradiction and Excluded Middle

Binary logic has its origins in Aristotle. According to Aristotle, a proposition must either be "True" or "Not True" - never anything else. In symbolic logic this Law of the Excluded Middle is written ( $A \vee \neg A$ ).

By taking the DeMorgan inverse of TRUE, the following statement is also true: "A AND NOT A = FALSE". In symbolic logic:  $\neg (A \wedge \neg A)$ . This is the Law of Non-Contradiction.

For our purposes, this means that the system has either an abnormal condition or a normal condition - never anything in between. For any given data set, there is no situation under which it would be normal but another under which the same data set would be abnormal. It's either normal or it isn't. If there exists a data set that does not comply with these two laws, then no sensor or algorithm exists that can properly detect the alarm condition.

2000 years later, mathematician Bertrand Russell challenged Aristotle with a barber paradox:

There is only one barber in town.

The barber cuts the hair of everyone who does not cut their own hair.

Does the barber cut his or her own hair (Yes or No)?

IF the barber cuts his own hair, then it creates a contradiction – he cuts his own hair, yet his hair is cut by the barber, who only cuts the hair of people who do not cut their own hair ( $A \wedge \neg A$ ). However, if he does not cut his own hair, then he must go to the barber (himself) and thus cuts his own hair ( $\neg A \wedge A$ ). Either way, it is a paradox.

Binary logic is useful in computers, which has hardware that ensures that each bit is either a 1 or a 0 – never neither and never both. However, in real life, binary logic can produce paradoxes. The correct answer to the barber question lies in multi-valued logic (the birthplace of "fuzzy" logic):

The statement “the barber cuts his or her own hair” is 50% true.

While fuzzy logic has its place in a number of computer applications and control systems, it does not have a place in the detection of abnormal conditions. The conditions "Alarm" and "Not Alarm" are mutually exclusive and also exhaustive - meaning that the system has no states other than "Alarm" and "Not Alarm," all data belongs to one and only one of these two conditions. We must be able to either give an alarm or not give an alarm - not give a 50% alarm.

To that end, consider another real world example:

Q: Given an apple that is  $\frac{3}{4}$  red and  $\frac{1}{4}$  green, is it a red apple or a green apple?

A: The statement “This is a red apple” is 75% true and 25% false.

However, hypothesis testing requires a binary result. To accommodate this, we must set a threshold value:

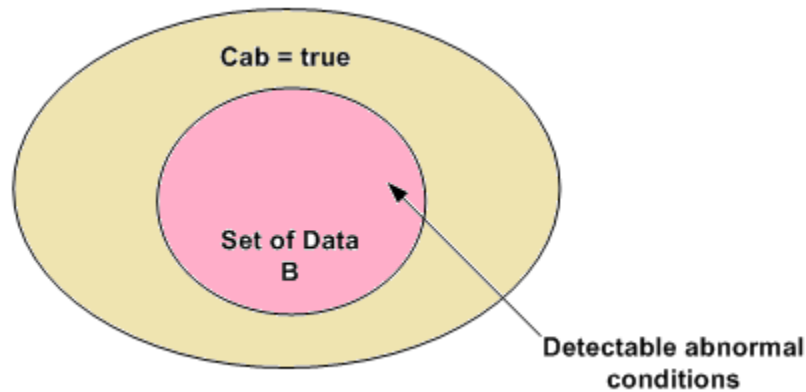
A: IF 75% or more of the area is red THEN it is a red apple ELSE it is NOT a red apple. This test condition produces a binary result.

Since the system is either producing an alarm or it is not, the use of thresholds is a common method of taking real world data and creating a binary result. However, in doing so we must make certain that the Laws of Non-Contradiction and the Excluded Middle are obeyed (which is codified formally in Axiom 1).

## Examples From the Real World

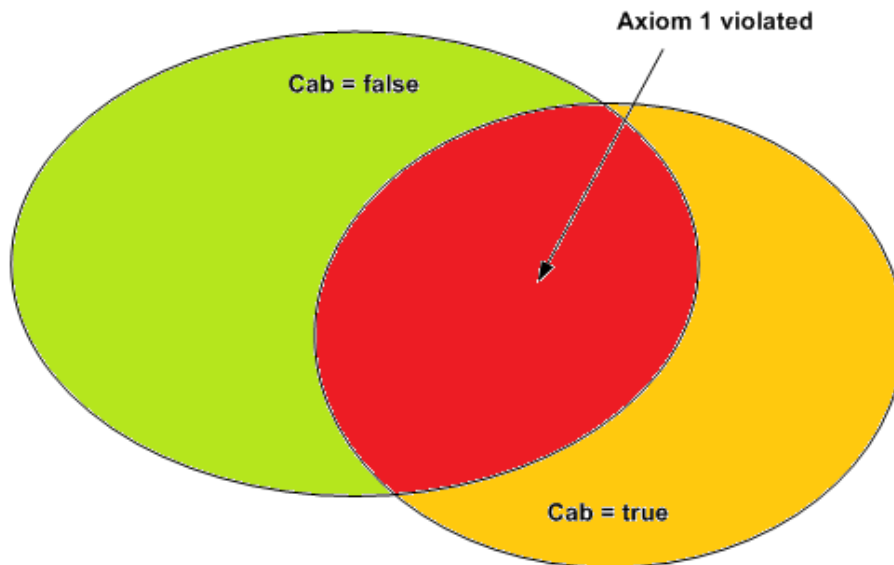
Here are some examples of the types of sense data situations that have been found during investigations of real system issues.

*Figure 8 - Sensor Unable to Detect All Abnormal Conditions*



This condition violates Axiom 1 and can occur when a sensor is not appropriate for the task. For example, in one investigation it was learned that the object being sensed often hit the sensor pin at nearly a right angle, producing no perceptible pressure change.

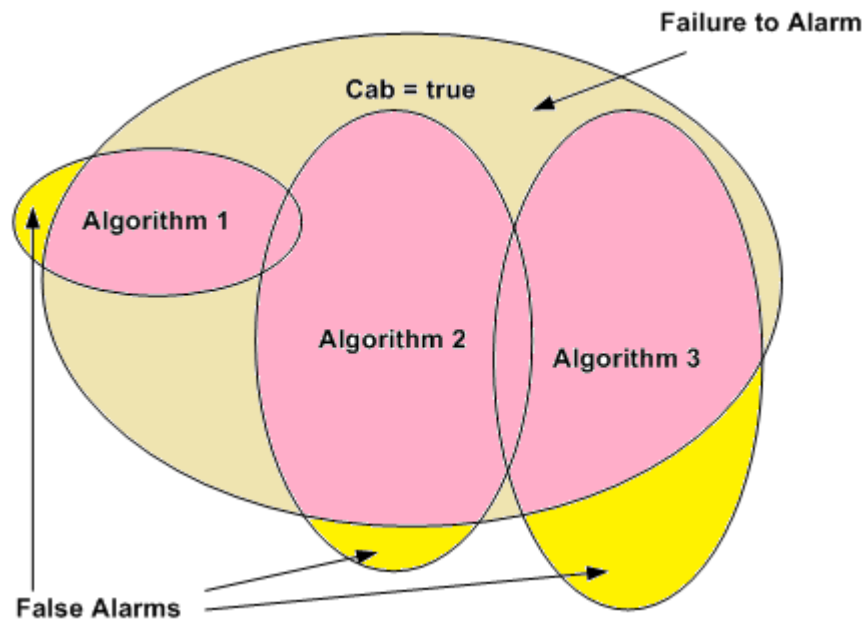
*Figure 9 - Sensor Cannot Distinguish Between Normal And Abnormal Conditions*



This is a violation of Axiom 1. This situation may occur when a sensor is used to detect a condition for which it was not originally designed. For example, in one case the use of the pressure sensor was expanded to detect leaks caused by a jammed inlet valve. Verification testing shows that this leak detection method has only a 42% confidence level because most of the data sets overlap (58% of leak data sets are indistinguishable from non-leak data sets).



*Figure 10 - Multiple Algorithms Required for Coverage - Increasing False Alarms*



This is a possible response to the situation in figure 1 above. However, while increasing the number of algorithms reduces the Failure to Alarm rate, the False Alarm rate rises.

## Conclusion

In summary, we can conclude that the most critical factors in a system's ability to detect abnormal conditions are (highest importance first):

1. The hardware's ability to comply with Axiom1 over the range of all foreseeable component tolerances and situations.
2. Having a low False Positive rate. If multiple algorithms are required for a single abnormal condition, then the False Positive rate becomes even more critical.
3. Having a sufficiently low Failure to Alarm rate that multiple algorithms for a single Alarm are not required.

# VVISSIIOONN

I recently discovered an unusual Christmas gift on my chair at work: a book titled the way of Z, subtitled “practical programming with formal methods.” I soon discovered that this was being considered as a new standard for creating software design specifications at my workplace. I wasn’t sure whether to be excited or appalled, so I decided to do something different – I decided to read the book before forming an opinion.

Actually, this is impossible, since whenever new data conflicts with a pre-existing belief, the resulting cognitive dissonance creates a fear reaction. The new data is immediately viewed as a potential threat and is attacked. This isn’t a malicious act – it’s just the brain’s way of saying “different = possible threat = bad until proven good.” By the time I was even consciously aware of the nature of this book, I knew that it was somehow “wrong.” Fortunately, the conscious mind can examine its reactions and determine if they are justified. So overcoming my resistance to change came before reading the book.

I’ll ignore “Part I – Why Z?” It’s largely the opinion of the author along with his supporting arguments. I perused it enough to realize that, while we both worked in safety critical industries, we had formed entirely different opinions about the role of verification and formal methods. That’s ok, Brahms and Beethoven had diametrically opposed views on how music should be organized – yet both are considered to have been great composers. Brahms was staid and stolid, his work conforming to numerous rules laid down by German schools of music. In contrast, his nemesis Beethoven was a radical who broke as many rules as he could. He even changed keys in the middle of a composition (as we hear near the end of the first movement of his 5<sup>th</sup> Symphony)! At least when J.S. Bach changed keys, he did so according to formal rules of his own invention (pun intended).

Here, then, are my observations based on a first reading of the book.

## Pardon My Fallacy

My first formal observation comes from page 35, where the author commits the fallacy of “equivocation,” in which one meaning of a word is confused with another meaning. My favorite example of equivocation is the syllogism:

```
Lamps emit light.  
Feathers are light.  
Therefore lamps emit feathers.
```

On page 35 the author presents the following Z statement:

```
iroot : N → N
```

This is presented as being the Z equivalent of the C statement:

```
int iroot(int a)
```

The next several paragraphs are devoted to explaining how the Z description is stronger than the C language statement. However, his assertion that an integer N in Z is the same as an integer (int) in C is not true. In Z, the letter “N” stands for Natural Numbers {0, 1, 2...} and are always positive. However, in C, an “int” is a *signed* value and can be negative. The C equivalent of “N” would be “*unsigned* int”, not “int”. The next several paragraphs rely on the reader not catching this equivocation. Once “unsigned int” is substituted for “int”, the author’s main argument falls apart.

The author next claims that Z clearly states that a single value is returned, while C does not. In fact, C does. The C language specification states that each C statement always has a single value and that functions either return a single value or nothing (void). To return multiple values, C must return a single value that may be a pointer to an array or structure of values.

The only remaining point in his argument is that the C function does not define its behavior when passed an invalid value for “a”. However, neither does Z. Both C and Z rely on the meta-rules of their language to give us an interpretation of their symbols. The meta-rules of Z tell us what to make of its symbols and how they are to be manipulated and interpreted. While the meta-rules of C do not tell us what to do with invalid parameters, the rules of engineering do just this. The practices of software engineering and coding standards tell us that it’s the job of a function to determine the legitimacy of parameters before using them (e.g. check for NULL pointers) and to return only valid values.

While this refutes the argument that the author presents for claiming that Z is stronger at defining conditions than C, it does not mean that his claim is invalid. The author is simply comparing apples to razor blades and claiming that apples are more nutritious. I don’t think he needs an argument to do that.

## “Say It Again, Sam”

On page 36 the author makes an argument that Z is stronger at defining behavior than natural language. Personally, I think he makes an argument against Z itself. “Stronger” does not mean “better.” Here is his first statement:

**Statement 1:**  $iroot(a) * iroot(a) \leq a < (iroot(a) + 1) * (iroot(a) + 1)$

Now, here is another statement, which he presents a few sentences later:

**Statement 2:**  $iroot(a) * iroot(a) < a \leq (iroot(a) + 1) * (iroot(a) + 1)$

Both statements are examples of well-formed Z statements. Both statements may be perfectly valid statements in a design specification. Both statements may even be true. Here is my challenge: match the statements to the intentions below:

**Intention 1:** Take the square-root of “a”, rounding up to the nearest integer.

**Intention 2:** Take the square-root of “a”, rounding down to the nearest integer.

I have no doubt that you can figure it out. However, how much effort did it take to figure out which statement correctly represents each intention? Did you do this purely with reason, or did you perform “testing” by taking a few values of “a” and trying them out on each statement? Both Z statements are correct representations of an intention; but which is easier for a human being to understand?

For that matter, which of the following statements is clearer?

**Statement 3:**  $true \wedge ss' \langle x? \rangle \wedge s = ran \quad ss \wedge s' = ran \quad ss' \Rightarrow s' = s \cup \{x?\}$

Or:

**Intention 3:** “Insert object  $x$  into list  $s$ ”

Oops – there are syntax errors in one of the above. Where? (Check page 249 to see.)

**$7 \pm 2$** 

The standard test for memory is to memorize five words and be able to recite them after 15 minutes of unrelated conversation. Why? In 1956, psychologist George Miller published a paper showing that a human being can mentally manipulate no more than  $7 \pm 2$  mental objects at one time. He says that this forms a concrete limit on our ability to transmit information. Miller concludes by saying that, to manage complex ideas, we must aggregate complex objects into “chunks,” which then become a single mental object.

Z does just the opposite. It takes single concepts and breaks them into multiple symbols, whose meaning requires close examination to understand. Consider the statement

$$\text{iroot}(a) * \text{iroot}(a) \leq a < (\text{iroot}(a) + 1) * (\text{iroot}(a) + 1)$$

Treating parentheses as a single “concept,” this statement uses eight concepts to represent a single idea:

*iroot*  
( )  
a  
\*  
 $\leq$   
<  
+  
1

By contrast, the prose statement uses only four:

Square-root  
a  
round up (or down)  
integer

Z adds to the complexity of the specification and obfuscates the intelligent interpretation of its statements. By breaking a concept apart into more chunks, it becomes harder for a human mind to comprehend. If I had an intention but wrote the wrong Z statement, only testing of the Z specification itself would ever reveal the error. In other words – we have the Turing Halting Problem in another form: there is no method of determining that a well-formed series of Z statements is a correct representation of a given problem, since any program that examines the statements may itself contain errors.

Human testing of the specification is needed to verify the specification. And testing itself requires a set of test cases; else we might forget to test something. And the test cases may contain errors. And even after testing the specification, we still don’t know if a program *written* to a correct specification was *correctly* written from the specification – it may still contain errors.

The only thing Z does is remove the possibility of equivocation and amphiboly (where the same sentence may be interpreted in multiple ways). This leads me to the one place where Z may be useful – designing state machines.

## State of Z Union

One of the biggest problems with state machine design is the handling of multiple or unforeseen events. Specifying state machines requires more attention to correctness and completeness than most other aspects of software design. By “correct” and “complete” I mean:

**Complete:** *Every* desired behavior is captured in the design specification, and the specification does not include any *undesired* program behavior. (You mean – you didn’t *want* it to crash when you pressed those keys at once?)

**Correct:** Every *actual* program behavior is specified in the design statements. This means that *everything* the program does was specified from the beginning (including the crash when you press those keys at once).

With state machines, it is easy to think of state-event-action. However, it is harder to think of every possible state-multiple simultaneous events-action. Sometimes this can be handled by adding guard conditions in the hope that only one of the events will be valid. But what if several simultaneous events are valid? What does the state machine do next? This is where formal design languages can help – if they can ensure a consistent interpretation and identify contradictions. Of course, there is limited value in having a specification language that is *capable* of being complete and correct if we have no way of *knowing* when that happens! (Gödel’s Incompleteness Theorem Strikes Back!)

## What Did He Say?

“Did you hear the story about the three holes in the ground?”

“No.”

“Well, well, well.”

My little brother once tried to tell this joke, but made an error:

“Did you hear the story about the three wells in the ground?”

“No.”

“Hole, hole, hole.”

Both are examples of well-formed statements; but only one is funny. What rule exists to say which one is the “correct” joke, since both are well-formed and comply with the same rules of grammar? By contrast, here is a well-routed message from the Internet. A human mind can read it, though it defies all rules. I’ve presented it to 5<sup>th</sup> graders and they can all read it; even many of my 3<sup>rd</sup> graders could read it. That’s because the human brain contains optimizing pattern recognition hardware. However, if this were a series of Z statements, how could we ever make sense of it, or even know if it was wrong in the first place? A few missing or transposed symbols may never be noticed – until it’s too late.

I cdnuolt blveiee taht I cluod aulaclyt uesdnatnrd waht I was rdanieg. The phaonmneal pweor of the hmuan mnid, aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it dseno't mtaetr in waht oerdr the ltteres in a wrod are, the olny iproamtnt tihng is taht the frsit and lsat ltteer be in the rghit pclae. The rset can be a taotl mses and you can sitll raed it whotuit a pboerlm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe. Azanmig huh? yaeh and I awlyas thguhot slpeling was ipmorantt!

## 2B<sub>∨</sub>¬2B?

One of my favorite T-shirts reads:

“There are 10 kinds of people – those who understand binary, and those who don’t”

So it was that on page 235 I saw this line:

$$data' = data \oplus \{r? \mapsto vm \ p?\}$$

At first I interpreted  $\oplus$  as “exclusive-or”, until I remembered that in Z, this means “override operator”. This leads to the most confusing part of formal languages: *the symbols themselves are not unique*. The world has a plethora of formal languages, but not enough unique symbols for them. So we reuse the same symbols across multiple languages, but with different semantic meanings. This means that I not only have to know several formal languages, but I have to be able to juggle multiple meanings of the same symbol. I think that only the Electric Monk<sup>17</sup> could manage this without error. Here are just a few examples of the multiple meanings of symbols:

Symbol	Formal Language	Semantic Meaning
< >	Predicate Calculus	Bounding symbols
< >	Algebra	Less than, Greater than
( )	Algebra	Bounding symbols
$\oplus$	Z	Override operator
$\oplus$	Logic	Exclusive Or
$\sim$	C/C++, Predicate Calculus	Negation (NOT or Inverse operator)
$\sim$	Z	Relational Inverse
$\neg$	Z, Logic	Negation (NOT operator)
$\supset$	Predicate Calculus	If...Then
$\Rightarrow$ or $\rightarrow$	Logic	If...Then
$\Rightarrow$	Z	If...Then
$\rightarrow$	Z	Total functions
$\Leftrightarrow$	Z	Logical equivalence
$\Leftrightarrow$	Logic	If and Only If ... Then
S'	Predicate calculus	Variable
S'	Z	State schema after an operation
S'	Differential calculus	First derivative of S

Now, was Range Restriction an  $\triangleleft$  or an  $\triangleright$ ? Or are those Domain restrictions?

<sup>17</sup> From the Douglas Adams book Dirk Gentley's Holistic Detective Agency.

## “This Has Been a Test”

While I’m sure that Z has some practical uses, I believe that the author of this book will fail in one of his original goals – to replace testing with *proof* of correctness without having to run the program. This concept was all the rage in the 1980’s, but has since been largely abandoned by the Department of Defense (its original champion) because, although proofs can be written, there is no method of proving that the proof itself is correct! (That Turing Halting problem just keeps coming back.) Proofs are unable to detect defects in the specification itself, or whether a program *claiming* to comply with the specification really does so under all possible inputs. Proofs cannot prove that defects do *not* exist, nor can they prove that the design correctly expresses what the user really wants. However, you *can* prove that loops terminate, types never mismatch, parameters are never NULL, and so forth.

## Can I Prove Anything?

*Nothing* can prove the absence of defects. “The way of Z” proposes to replace testing with proofs of correctness. That mindset still existed in some places at the time the book was first published, but has since been discredited except in one area: algorithms. An algorithm is a mental construct that lends itself to formal proof. For example, a communication protocol could be diagramed and analyzed formally prior to any implementation. In a case like this, formal proofs are actually more effective than testing, since testing cannot look for occurrences of things like deadlock or process starvation. However, to facilitate understanding I would still consider flowcharts and UML diagrams to be more effective tools than Z, since they can better capture the dynamics of the system.

## Conclusion

The design process is our primary method of ensuring that defects do not creep into our design. But who is the design really for in the first place? Design specifications are written for people, not computers. Formal languages are meant to be understood by people – computers don’t care. The purpose of formal languages is to enforce rigor in the design process and to provide a means of ensuring that equivocation and amphiboly do not creep into our design (also known as *vagueness* and *ambiguity*). To the extent that a language like Z can accomplish this, formal languages are a good thing. However, they come at the expense of increasing the complexity of the design document. We need to make sure that these languages are used in a manner that improves the quality of the design and the ability to communicate the design to others.

What came out of the Dept of Defenses’ failed attempts at *proofs of correctness* were improvements in the design process itself, such as test-driven design, object-oriented design, and formal ways of specifying things that are better described using formal languages. Can a design become a *better* design by *including* elements that are specified in Z? Probably. The author of the book admits that Z does not describe program structure, which is still an important component of software design. And we all know that what the user wants today will change tomorrow, so the structure had better be amenable to change! So I see a good specification as *including* formal behavior descriptions (e.g. Z), structural elements (e.g. UML or Data Flow Diagrams), formal definitions (e.g. Backus-Naur Form) and prose.

We must also realize that a design document may contain statements in more than one formal language, and those languages may reuse the same symbols but with different meanings. This is why some formal languages (like UML) try to be all-encompassing, so that they and only they are used in the design specification. In creating design specifications we must be careful to avoid the use of symbols whose meaning may be ambiguous. We also cannot rely on every reader understanding the meaning of symbols, so the formal statements must always be accompanied by the meta-statements that interpret them.

By the way, did you realize that the meaning of the title “VVISSIIOONN” is “Double Vision?”



# Nature Walk in the Big City

I once had the pleasure of being one of several sponsors taking a group of teenagers on a nature walk in central California. This was easy because there was still some Nature left in that region. However, I wanted to find a way to do a nature walk here in San Diego. I was in a quandary, until I realized that nature still exists in the big city – it just looks different.

## Ironwood

Ironwood is a curious adaptation to a human-altered environment. It doesn't grow in the wild – only inside human-made towns and cities. Ironwood grows straight and tall, with no branches (technically, this makes it a bush). It has generally a single leaf at the crown – such as a large, red leaf bearing the word "STOP," though there are many other varieties, such as "One Way," "Do Not Enter," and the yellow triangular "Yield." This curious adaptation allows Ironwood to thrive in a city environment, where it is often planted at street corners as a way of controlling traffic flow.



Ironwood is not unique to North America; it grows in many countries around the world, adapting its size, shape, and lettering to match the local language and environment.

This is just one of many examples of how nature adapts to our human environment.

## Plightning Bugs

If you grew up in the American Midwest, you recall those warm, summer nights filled with lightning bugs ("fireflies"). Their unique signature of flashing tail lights identified the particular sub-species of insect, allowing them to locate suitable mates.

However, the incursion of humankind into the region has caused a most curious adaptation – Plightning Bugs! These unique insects have developed the ability to parasitically attach themselves to existing plants as a way of enhancing their ability to send forth their pattern of colored signals in a human-rich environment.

Just as there are many subspecies of lightning bugs, there are also many subspecies of Plightning Bugs. One of the most common varieties sits atop a tall Ironwood plant, often displacing the ironwood's characteristic leaf, allowing it to sit high atop the pole while sending out its unique pattern of red, yellow, and green flashes. Some varieties of ironwoods have adapted to this parasite by sprouting a single, long branch – a suitable resting place for numerous Plightning bugs.



The Plightning Bug is clearly one of the more interesting adaptations of nature, in that its unique pattern of flashes is apparently an attempt to send mating signals to human vehicles, which often respond by flashing their own tail lights and making honking sounds.

## BellWeathers

The spider is perhaps one of nature's greatest curiosities. Having eight legs, it is not an insect, but an arachnid. Most spiders are unaware of this distinction and continue to be born with eight legs. However, nature has evolved numerous varieties of spiders to offset the fact that spider parents have to buy an extra pair of shoes for each of their children. We have trapdoor spiders, jumping spiders, tarantulas, daddy long legs, and water spiders, just to name a few. But the oddest spider of all is the BellWeather.

In many human-infested regions trees have become increasingly sparse. Many trees simply lose all their branches and foliage – especially along roadways where they are constantly exposed to automobile exhaust. BellWeathers are uniquely adapted to spin their webs on these denuded trees.



In this picture, we see a denuded tree festooned with BellWeather webbing. In adapting to a human-rich environment, the BellWeather webs are metallic and therefore well suited to carrying electricity. By making their webbing useful to humans, they avoid having people sweep away their webs.

Despite the fact that they are a web-spinner, BellWeathers do not sit on their webs waiting for prey. Instead, they lurk inside utility vans, waiting to attack passers-by or small vehicles.

Changes in the telephone industry have spawned entire subspecies of Baby Bells, which now roam the country freely, spinning their metallic webs with ease. Southern California is almost entirely infested with PacBells, one of the more voracious and predatory of all the BellWeathers.

Despite the apparent annoyance factor, the webbing is often useful to other creatures. It is not uncommon to see avian creatures such as birds and Plightning Bugs roosting on the BellWeather's unique webbing.

# Charge of the Developer's Brigade

(by Alfred Wyatt Tennyson)

1.

Half a byte, half a byte,  
Half a byte onward,  
Into the CPU of Death  
Rode the programmers.  
"Forward, Developers!  
"Charge for the bugs!" he said:  
Into the CPU of Death  
Rode the programmers.

2.

"Forward, Developers!"  
Was there a man dismay'd?  
Not tho' the coders knew  
Someone had blunder'd:  
Their's not to make reply,  
Their's not to reason why,  
Their's but to do and die:  
Into the CPU of Death  
Rode the programmers.

3.

Requirements to right of them,  
Requirements to left of them,  
Requirements in front of them  
Volley'd and thunder'd;  
Storm'd at by tests and QA,  
Boldly they rode and well,  
Into the jaws of Death,  
Into the mouth of Hell  
Rode the programmers.

4.

Flash'd all their source code bare,  
Flash'd as they coded in air,  
Fixing the defects there,  
Charging an army, while  
R&D wonder'd:  
Plunged in the V&V-smoke  
Right thro' the line they broke;  
Testers, Tech Writers  
Reel'd from C# stroke  
Shatter'd and sunder'd.  
Then they rode back, but not  
Not the programmers.

5.

Requirements to right of them,  
Requirements to left of them,  
Requirements behind them  
Volley'd and thunder'd;  
Storm'd at with tests and QA,  
While files and heroes fell,  
They that coded so well  
Came thro' the jaws of Death  
Back from the mouth of Hell,  
All that was left of them,  
Left of programmers.

6.

When can their glory fade?  
O the wild code they made!  
R&D wondered.  
Honor the code they made,  
Honor Developers,  
Noble programmers.

# The Asylum

Doctor Green walked into his office to find his next patient standing by the window. He hated it when the orderlies simply left the patients in his office. As the junior doctor in the *Nemo Memorial Hospital for the Incurably Insane* he felt that the orderlies often ignored him. While most of the inmates were non-violent, he was always afraid that someone would go through his things or possibly hurt themselves.

"Hello? Are you my next patient?" he called. The man at the window jerked around suddenly as if burned, his wild eyes holding a mixture of confusion and fear. Doctor Green knew better than show fear so he strode calmly to his desk and sat down in his high backed leather chair.

"Please, have a seat." he motioned his distraught patient to a straight backed chair opposite his crowded desk. The man shuffled nervously, his eyes darting quickly around the room, never resting long on the doctor. By his face and his nervous gait the man looked to be quite infirm. He sat down slowly, gingerly testing the seat to make sure it was solid.

"Mr..." Doctor Green pulled the top medical record off the stack and opened it. "Mr. Gruen? How are you feeling today?"

"It's pronounced 'Groon'." The man's nervous eyes avoided Dr. Green's gaze, flicking around the room like an insect in flight.

"I see. Well, Mr. 'Groon', this is your first day in our fine institution. How do you feel?"

"You're wrong doctor. This isn't my first day at all. I've been here longer than you think!" Suddenly the man lunged forward, grabbed the front edge of the desk and pulled himself up tall; his frail figure looming over the cluttered desk. "There's been a horrible mistake! I've been trapped here for longer than time can count and I desperately need your help!"

Doctor Green flinched at the man's outburst and his hand dropped down to the panic button under his desktop. The patient, seeing the alarm on Dr. Green's face, drew back and resumed his frightened look.

"Please don't push the button, doctor. You see, I'm perfectly sane. I understand now what happened and I really need to tell you my story." He stared wildly into Dr. Green's eyes, who found it oddly hypnotizing.

"All right, sit down please and tell me your story, Mr Gruen. Before you begin, would you like something to drink?"

Mr. Gruen paused, staring into the doctor's eyes, trying to fathom if he would be heard or merely patronized. At last he sighed and slowly deflated back into the chair.

"Iced tea please, if there's any left. Please call me Frank."

"Why yes, there should be plenty left..." Doctor Green had risen and was halfway to the refrigerator when he stopped, turning back to Frank. "Why did you say 'if there's any left'?"

“You always keep iced tea in your refrigerator. I know more about you than you think. I told you, I’ve been here a very long time.”

Doctor Green, determined to hide his uneasiness, opened the small refrigerator and searched among the scattered bottles of soda and packets of lunchmeat before finding a half-full pitcher of iced tea. He retrieved two paper cups and filled them each with tea. He returned the pitcher, closed the door, picked up the cups, and then walked back to his desk, his hands trembling slightly. He handed one cup to Frank, who instantly seized his hand.

“Doctor, please, I’m not mad! You must listen to me! Please!”

The doctor looked down at the emaciated hand holding him in an iron grip, which belied Frank’s apparent frailty. There was a tense silence, then Frank released the doctor’s hand and took the cup of tea. Doctor Green let out a small sigh of relief and hurried back behind the safety of his desk. He couldn’t help spilling the tea as he set it down on the desk, adding to the small pond of old tea stains.

“I’ll bet you spilled it again. You always do that when we talk.”

Doctor Green looked up into Frank’s eyes and saw there not the wild stare of lunacy but the calm gaze of serenity. He decided that Frank had seen the other stains on his desk and concluded that he frequently spilled tea. Dr. Green had learned that insane people often experienced periods of complete lucidity; he resolved not to let Frank shake his self-confidence.

“Mr. Gruen, Frank, I mean, what shall we talk about today?”

Frank leaned forward with a slow, deliberate movement.

“The same thing we talk about every day. Time travel.”

“I see. Time travel. Do you think that time travel is possible?”

Frank laughed aloud. “Oh yes! Not only possible, but I have done it! I have a time machine and now I’m stuck in this...” he made a wide sweep of his arm, “this *place*. Ironical, isn’t it?”

Doctor Green began to relax. He had let Frank get to him at first, but now things seemed to be under control. Frank was clearly delusional and probably highly intelligent; he must avoid falling into the trap of sharing Frank’s delusion. This was not the first time he had met an intelligent patient who could cold read a doctor, much like the psychics and side show mind readers could read a person. There was no mind reading at work here, just a disturbed, perceptive patient. Doctor Green leaned back in his chair, assumed what he thought was a scholarly pose, and let Frank tell his tale.

“Time travel isn’t like the movies. You don’t go back in time to stop assassinations, or warn people about global destruction. You can’t bring people or whales into the future to re-populate the earth. It just doesn’t work like that. Time is not reversible.”

“That’s an interesting viewpoint. How, then, did you come to invent a time machine?”

Frank’s face soured in exasperation. “I never said I *invented* a time machine; I said I *have* a time machine. I don’t know anything about physics - I’m a doctor by trade, and a mathematician.”

Doctor Green leaned forward with interest. Being an avid mathematician himself he saw the possibility of a more interesting conversation than a mere doctor-patient session. He picked up a pencil and pad and began scribbling notes.

“Do you still draw little stick men when you take notes?”

Doctor Green dropped his pencil. “Excuse me.” He gave a slight cough, leaned down, and retrieved his pencil with trembling fingers. He must not allow Frank to read his reactions like this.

“Mr Gruen, Frank, I am interested in why you believe yourself to possess a time machine. Is it here in the asylum?”

Frank smiled evilly. He leaned forward and whispered “It’s here, all around us. We are *in* the time machine. However, time travel isn’t really possible. I can’t go back in time and kill your mother before you’re born. I know, I’ve tried that and it failed.”

Doctor Green stood bolt upright, spilling his tea over the desk. He frantically tried to wipe up the spill but only succeeded in getting more papers wet. Finally he sat down, shaking visibly. Thirty years ago someone *had* tried to shoot his mother, she was pregnant with him at the time. The bullet nearly missed him and left his mother bound to a wheelchair. How had Frank known?

“You can’t imagine how many times I’ve tried to kill you, to erase your existence.” Frank continued speaking while Doctor Green was busy blotting the teas stain on his desk. “Twice I tried to run your mother over with my car, and once I even gave her a poisoned apple at Halloween! I guess that was a bit melodramatic, but her sister ate it and *she* died instead.”

Doctor Green was visibly upset now. It was true that his aunt had died from poisoned Halloween candy about ten years before his birth, but how could Frank have known? Frank must have researched his background before his committal here. This was clearly a dangerous man! He reached down for the panic button.

“Before you press the panic button,” Frank continued, “you should know why I know so much about your family.” The doctor’s hand paused on the button. “You see, time travel isn’t really possible. Time is not reversible. Do you remember that TV show where people went in and out of alternate realities? Time travel is like that. Once I go back, I don’t go back to anyone else’s time. My universe becomes a new universe - a new alternate reality in which I can interact freely. That’s why paradoxes won’t happen; the universe just keeps spawning new instances, each different from the old. That means that if I change something it only changes in *my* universe, not in anyone else’s. Time travel only affects the traveler.”

Doctor Green regained his composure long enough to point out the obvious hole in Frank’s story. “If time travel only affects the traveler, then why did you think that you could kill me by killing my mother? Why did you think you could kill me at all? I would only be dead in *your* universe.”

Frank grinned. “Don’t you see? I tried with mother first, before I realized that time travel only affects the traveler. After that, I tried killing myself directly, but it never seemed to work. I think that would have created a paradox internal to my own universe, so the universe always shifted before I died. It’s as if, once time is violated, my new universe

cannot be unmade; it always cycles but never repeats, each instance slightly different from the last. That's why I'm never sure how much tea is left in the pitcher. It keeps changing with each trip."

"Each trip?" the doctor queried nervously.

"By traveling backwards in time I had inadvertently created a time loop. I can't escape, and each round trip causes another parallel *me* to come into existence, each one slightly different from the previous. That's what I've been trying to tell you all these years. I know now that I can't change what happened; all I can do is act out my part and pass my memories on to the next *me*."

This conversation was getting too strange! Doctor Green gently pressed the panic button. The orderlies usually acted quickly, but this time they delayed.

"Look outside," Frank demanded, "and tell me how large this asylum was when you came here. Each time I'm carried away to a cell, the asylum grows; it *has* to grow so the new universe can accommodate another *me*. I don't even know how many of *me* are locked away screaming in the cells below!"

Doctor Green stood, his legs wobbling as he sidled to the window, never taking his eyes from Frank. He cleared his throat and hoarsely croaked out his next question: "By your own words ... your time travel cannot affect *me*, since it only affects the traveler. It can't affect myself or my office or the tea in my refrigerator." He could hear the orderlies running this way. "So why have you tried to kill *me*?"

"Your argument would be sound, if we were separate people."

The orderlies rushed into the room, snatched Frank from his chair and pinned his arms behind his back. One orderly asked "Shall we take him to a padded cell?" The doctor nodded numbly, unsure what to do next. As the orderlies dragged Frank from the room, Frank cried back: "*Welcome to Hell, doctor!*" The door slammed shut, closing out Frank's cries for help as the orderlies dragged him to a padded cell.

Doctor Green turned away and looked out the window; he could faintly hear the screams and moans of the insane locked in their distant cells. From this viewpoint the building looked larger. He had never noticed before, but the west wing extended all the way to the woods. He stood at the window musing for several minutes when the door opened behind him. A voice spoke.

"Hello? Are you my next patient? My name is doctor Mills."

Doctor Green whipped around as if burned. He saw a younger man standing in the doorway, wearing *his* white coat; except the nametag read "Dr. Mills." His legs grew weak as the memories flooded back.

"And you would be..." the newcomer looked down at his clipboard, "Mr. Groon? Did I pronounce that correctly? Please have a seat. Would you like some tea?"



# The Project Manager

*My thanks to Franz Kafka, whose short story The Metamorphosis provided the inspiration for this story; and also to all those project managers who made their work look so important.*

Gregory Samson awoke one morning to find that he had metamorphosed into a project manager. There could have been worse things he could have morphed into, such as a hideous vermin; however, becoming a project manager was still a very traumatic experience! Had he been born a project manager, he didn't doubt for a moment that this experience would seem perfectly ordinary. But, going to bed as a hard-line computer programmer, then awakening to find oneself turned into a project manager – well, that was more than most people could handle. But Gregory had always prided himself on having an open mind, and he resolved to make the best of this unfortunate transformation.

As he lay in bed, wondering what to do next, he could hear Greta, his wife, clattering about in the kitchen, probably making breakfast. At least he hoped that it was breakfast, though he wasn't sure what project managers should eat. He doubted that the potato chip and soda diet that had gotten him through many software projects would suffice. Gregory made a mental note to call his doctor and ask if project managers required some kind of special diet.

As Gregory pondered this sudden change in his life, he felt a cold chill. Had Greta left the window open again? She often left it open on warm nights, but last night had been quite cool. He tried to turn to see the window, which is when he noticed that he could no longer swivel his head in the old fashion. Silently, he chided himself: "Of course! I should have realized that, as a project manager, I will be restricted to tunnel vision!" Even the mere act of turning his head to see a different point of view would now involve cumbersome bodily movement. Oh well, he could check the window later. What he needed now was a way to get out of bed. That was when Gregory realized just how much trouble he was in.

Gregory had never, not even once in his entire life, written a plan to get out of bed. In the past, he had always risen in an ad-hoc fashion, and probably not even the same

way two days in a row! Now he chided himself for such short-sighted behavior. Clearly, if he was to rise from bed, and be certain that he had arisen in an intended and reproducible fashion, he would have to devise a plan. He lay in quiet thought, devising his plan and anticipating pitfalls. First, what was his goal in rising from bed? Clearly, if he was to expend the effort there had to be some return on his investment. Since today was a workday, and a Monday at that, he must rise soon or he would be late for work. Good! Now he had a goal and a justification! Secondly, how long should it take to rise from bed? He had never thought about how well he rose from bed, or whether the process could be improved. But, as a project manager, he had to be concerned about adherence to schedule and continual process improvement. Obviously, this process could not be started without a schedule, so Gregory began to run through the steps in his mind, estimating how long each step would take.

“AH!” Gregory exclaimed. The clatter in the kitchen paused and Greta called out “Are you alright, dear?” Gregory didn’t want to alarm his wife, so he quickly replied that all was well. But all was definitely NOT well! Gregory had been mentally designing the implementation of how he would rise from bed before he even completed the plan! Now if that wasn’t putting the cart before the horse, he didn’t know what was! “Well,” he chided himself, “I’m new at this, so it’s to be expected that I would make a few mistakes at first.”

He could hear Greta walking towards the room. His spirits lifted as he realized another of his mistakes: he had been trying to do this task all alone. “There’s no ‘I’ in ‘Team’!” he reminded himself. “All it takes is teamwork!” He heard Greta open the door, but could not see her as he had not yet turned himself to see the door. Then he heard her sharp gasp, and the sounds of the breakfast tray hitting the floor.

“OH! Oh ... my GOD!” Greta gasped.

“Yes, is anything the matter dear?” Gregory inquired, suddenly pleased that he had an issue to investigate – to wit, the reason why the breakfast tray had fallen.

“Is...is that you, Gregory? Oh, my GOD! What happened to you?”

“Nothing to worry about, my dear. I’m a project manager now. We always knew that it could happen. Now that it has, I feel oddly collaborative. Perhaps you could help me draft my plan for getting out of bed?”

Greta quietly closed the bedroom door. Now that Gregory was in the kitchen, eating his hastily repaired breakfast, this was the only room where she felt safe. She recalled an incident last year, when a neighbor's husband had metamorphosed into a congressman. The poor woman had tried to hide it as long as she could, but once the rumors got out they had been forced to move into a leased penthouse suite and accept bribes. Greta didn't think she could take such a change, and hoped that her husband's condition was only temporary.

She could hear Gregory start washing the dishes. In the past he had always refused to do any kind of housework, saying that it wasn't his job to do such tasks. "Oh no!" Greta panicked! "What if he starts doing housework all the time! He'll be underfoot and impossible to manage! Well, if that happens, I'll have no choice but to talk to a physician."

Greta jumped as the phone rang! She was in no mood to talk to anyone, but to her surprise she heard Gregory answer the phone. Gregory had always ignored the phone, treating it as an unwanted interruption. She sat dejectedly on the edge of the bed, the walls muffling the ensuing phone conversation. She flopped onto her back, staring blankly up at the ceiling, wondering what the neighbors would think when they discovered what had happened. Her mind churned with the possible outcomes. Perhaps if she locked him in his room, all would be well. After all, programmers were known for holing up for long periods while working. She was sure that her friends would believe her if she told them that he was diligently working on a new program. Of course, there was Gregory's boss – she wasn't sure how long a telecommuting excuse would be accepted.

"Greta dear!" Greta jerked upright as Gregory burst into the room, uncharacteristically exuberant. You'll never guess who just called! That was Ansil, at the company! There's just been a new job posting for a software project manager! They've finally decided to create the position! Do you realize what this means!"

Greta only stared at her husband, trying hard to hold back the tears at seeing the project manager that her sweet, programmer husband had become.

"It means that now I'll be able to synergize and leverage the combined capabilities of a cross-functional development team by using an integrated systems approach! Isn't that fantastic! It's odd, you know, to think that yesterday I wouldn't even have known what that meant; but today I finally realize just how important it all is!" Gregory was beaming with pride! Greta only nodded dumbly, vowing to make an appointment with the family physician that very morning.

---

Greta had been whispering on the phone for fifteen minutes. Gregory had no idea who she was talking to, but Gregory had never shared the same set of friends with his wife, so he was unconcerned. In fact, his friends were all coworkers – fellow programmers who shared the same goals. He wondered if his friends would understand his transformation, which clearly put him outside their normal clique. But he could deal with that when the time came.

Gregory was starting to get the hang of this now, having written the assembly instructions for his business suit in less than twenty minutes. Of course, next time he would have to use ink instead of pencil. Perhaps tomorrow he would have time to create templates for all of his commonly performed tasks, such as shaving and eating. He couldn't help feeling a sense of shame at the way he had repaired his breakfast earlier – all without any rework instructions. Still, he was new to this life and he was certain that whatever Power had wrought this change upon him would also be understanding and make allowances while he adjusted to his new life.

He only had one suit – his interviewing suit – but with his transformation he would now have to buy more. That might strain their budget for awhile, but now that he had a better understanding of the capital budgeting process, he was confident that it could be handled. It was really just a matter of planning the acquisition of capital resources.

"Gregory, dear," Greta stood at the bedroom door, her eyes reddened, as if she had been crying. "You ... I mean, I need you to take me to the doctor. Right away, please."

"Is something the matter? Was this a planned illness?"

"No, dear, it came rather – suddenly. I ... just, please take me to the doctor right away. And, well, you may want to have him look at you too; you know, just to see..."

her voice trailed away; her words choked as she found herself unable to speak openly to her husband about – his new condition.

“Just a moment, dear. I’ve had some difficulty, and had to make some modifications to my dressing procedure. Could you please review the procedure, then initial and date my changes?”

---

The doctor’s office only had a few patients, for which Greta was thankful. The nurse clucked sympathetically as Greta checked in. “Poor thing,” she overheard one of the office staff mutter. Greta tried to hold back her tears – resolved to braving her way through this ordeal. The nurse said that the doctor would see her in a few minutes and would she please sit down and Gregory won’t hurt anybody will he it’s safe to take him out in public like this isn’t it?

As Greta sat down, she saw Gregory begin organizing the magazines on the waiting room table. She could see the shocked looks of the patients, and one woman pulled her young daughter back in horror as Gregory approached. “How could she!” Greta heard the woman mutter to the man next to her – probably her husband. “She should be ashamed to take ... that... manager ... out in public!” The woman spat the word manager, as if by merely uttering the word she could become tainted by Gregory’s condition.

“The doctor will see you now.” Another nurse stood at the hallway entrance, holding a packet of papers. She put her hand over her mouth and gasped silently as Greta led Gregory to the examination room. The nurse didn’t enter the room, but instead dropped the bundle of papers into a holder on the door, then scurried away. Greta could hear the faint, shocked mutterings back in the waiting room. Well, she had endured much in her five years of marriage, she supposed that she could get through this ordeal. Her mother had often wondered why she hadn’t married a nice doctor, instead of a computer programmer. To her mother, computers had always seemed so unnecessary, even parasitic at times; whereas the world would always need doctors.

Franz, the doctor, entered the room, looked at Greta, and asked what was the matter. He then looked at Gregory and, with stoic composure, said that he understood now why Greta had wanted an appointment so quickly.

“Doctor,” Gregory interjected, “I know that your office is constrained by limited resources, making it difficult to plan, control, and execute your daily tasks; but Greta insisted on seeing you as soon as possible. Honestly, though, I can’t imagine what the problem might be. However, as a stakeholder in your medical practice, I must commend you on your rapid response. Now that I understand the importance of time management, I can understand what a strain her request must have placed upon your system of project controls.”

Greta nodded meekly to the doctor, silently saying “see what I mean?” The doctor nodded back, and began to inquire as to when and how this remarkable transformation occurred. Greta answered as best as she could, while Gregory, still unaware that he was the patient being examined, began mentally sketching outlines of possible system development life cycles for designing his daily commute.

Greta told Franz how the transformation had occurred suddenly this morning, though in truth she had not turned on the lights when she rose to make breakfast, so it could have occurred anytime during the night. Yes, he had been perfectly normal the day before. He had spent all day Sunday mounting his new satellite dish on the roof which, in a two story apartment complex, was not a simple task. She had worried the whole time that he would fall and break something. Gregory was always impulsive like that – or had been, before the transformation. Being a programmer, he had no practical tools, but had managed to borrow a long ladder from the company where he worked; the facilities department had lent it to him for the weekend. He was supposed to return it today. He had borrowed the other tools from the apartment manager.

Franz stared at the ceiling in thought during Greta’s discourse, absorbing the details, and finally formulating a conclusion. “Ah ha!” he stated in a scholarly voice, “your problem is evident. You say he had borrowed the ladder from his company?”

“Yes, they sometimes allow people to borrow company equipment for personal use.”

“And the apartment manager had no objections to putting a satellite dish on the roof?”

Greta winced as she recalled the arguments between Gregory and Harold, the apartment manager. “Not after Gregory offered to give him a free satellite feed from the dish. Something about amplifiers and splitters and ESPN. Anyway, the manager was helping him install the dish.”

Franz nodded approvingly. “I see now what has happened. The problem is simple – your husband has climbed the corporate ladder, and this is the inevitable result!” Franz beamed proudly at his diagnosis.

---

Gregory sat proudly in his new cubicle, carefully laying out his plans for the next big project. His cubicle was not too large – just enough cost effective space to hold his desk, his computer, his project documents, and some books on project management. The promotion at the company had certainly helped with the finances, and while Greta was still having difficulty adapting at home, Gregory felt certain that she would come around in time. She only needed to see the improvements that his metamorphosis would bring to each project’s schedule, budget, and quality. In only his first week, he had finished benchmarking his team’s performance and, with this new information, he was now ready to adjust the original project schedule. Yes, Gregory was at home in his new form, with his new responsibilities.

Greta knocked on the cubicle wall. “Dear, are you ready for supper now?”

Gregory had no idea why the night staff always bothered him this way. Still, it was nearing 6 o’clock, and his doctor had insisted that he take regular breaks to avoid repetitive motion injury. He sighed, put the finishing touches on his project risk analysis, and said that he would eat now.

Greta brought in his supper tray, setting it gently onto the desktop. “Please, eat everything. Otherwise, I’ll have to throw it out.”

Gregory nodded assent. He already felt stronger on his new diet, so he really shouldn’t resent it when this nice lady on the night crew brought his meals. She seemed

to take such an interest in his well being, and she always kept his cubicle tidy. He thought perhaps he should send her supervisor a letter of commendation.

Greta sighed, and walked back towards the kitchen. Her mother, who had been sitting on the sofa, rose to follow her, offering what consolation was able provide.

"I'm so sorry that it turned out like this, Greta dear. Gregory was so much nicer when he was just a programmer." She cleared her throat, as if hesitating to ask her next question. "Is ... is the living room going to stay that way, with that awful - *cubicle* - taking up half the room?" She tried not to show disgust from her tone of voice. Greta nodded dumbly.

"I'm afraid so. The doctor says that he must be kept in his natural habitat. I guess I'll just have to adjust ... somehow."

Greta had already been looking for a new apartment. The neighbors had been whispering about her dilemma, and she could no longer bear how they turned away whenever she came near, as if Gregory's disease could somehow rub off by association. And she could no longer bear the sad looks at the supermarket when she bought Gregory's food. Still, the extra income had been helpful, allowing her to look into some of those new condominiums on the hill. The company had promised to give him a good credit reference. At the newest development complex, the homeowner's association took Gregory's kind, and she was certain to find some sympathetic friends among the wives there.

"Excuse me," Gregory called from the living room. "I need some Quality Assurance input on my latest calculations." Greta sighed, saw her mother's mournful look, and resignedly walked towards her husband's home habitat.



# Stone Age Programming

Oog looked at the stone he had shaped – it was a good stone. A soft piece of limestone with sharp corners where he wanted them and soft curves where they belonged. Oog was proud of his stone, his best effort yet. Proudly, Oog took his stone to the Clan Woman, who was collecting stones to make the clan fire pit.

“Oog make good stone.” Oog’s leathery hand extended the finely crafted stone. Clan Woman hefted the rock and scowled.

“Oog, I need big granite chucks to make fire pit. This no good! This not fit! Also, you late by hours, and this not even right kind of rock!”

Oog was incensed! How dare the clan woman rebuff his fine effort! While it was true that she had asked for rocks to make the fire pit, Oog knew that his rock – his work of art – was better than any rock found on the ground! How dare she belittle his fine artistic talent! How dare she tell him how quickly to create art! Oog knew that maintaining his integrity as an artist was more important than Clan Woman’s imagined needs.

“Make it fit!” Oog shouted as he stormed away.

Oog’s spirit was reincarnated as a computer programmer. As a programmer, he loved crafting finely tuned programs. Sometimes, when he did not have the time to write the code himself, he would buy a program; although he always knew that he could have coded it better. Oog knew that a well written program was always in need and that his customers would appreciate the time he spent ensuring that each line of code had just the right amount of elegance.

Clan Woman was reincarnated as Oog’s project manager in the Information Systems department. Clan Woman was concerned about meeting the needs of the company – on time and within budget. However, Oog’s programs remained 90% done for months at a stretch, and many of his programs were merely solutions that were looking for a problem to solve. Clan Woman tried to explain to Oog that computer science is not the same thing as software engineering, and that discipline

was needed to ensure that the solution fit the problem, but Oog never seemed to get the point. She talked about the budget and the schedule, but Oog only replied that programming was an art and that it was not possible to put a time constraint on art.

One day, Oog presented clan woman with a finely crafted database. It was true that the company already had a database, but this database was far better because he had coded it with the right amount of elegance. Oog explained that this was the ideal solution to the company's information problems. Clan Woman, fearful of angering Oog, installed the database onto the computer network.

Installing the new database required buying a new server, since the new program could not coexist with the old database. All of the information had to be transferred manually from the old database to the new one, since Oog had not written a data converter. This caused numerous transcription errors, which Oog knew were entirely the fault of the underpaid transcribers that had been hired. The new program also had a plethora of annoying defects.

"The program crashes when I do this!" one person complained to Oog.

"Then don't do that!" Oog replied.

"I don't understand this user interface!" complained another.

"I don't see what's so confusing about it." Oog commented. "It looks perfectly simple to me."

"This feature doesn't work right!" shouted a third.

"That's because you didn't install yesterday afternoon's software patch!" Oog retorted. "You should have asked me if I had fixed that!"

---

Some programmers are like Oog – carefully crafting software while naively believing that computer science is the same thing as software engineering. Like Oog, they create solutions where there are no problems, or present solutions without analyzing the problem. Sometimes, solutions are presented before problems are even identified. They tell Clan Women that the program is "90% done"

throughout half of the project, and then deny that software schedules are even possible. Oogs view software engineering and systems engineering as simply the act of writing larger and more elegant programs – rather than defining the problem and designing the solution. As a result, many projects begin with “we need this database” and end with “why didn’t your system work?”

Today, managers are often presented with solutions – Oog’s finely crafted stones – and simply instructed to implement them. Oog’s approach to engineering is oxymoronic – implementing a computer program without defining the problem or designing the solution. The answer is establishing the discipline of software and systems engineering, and forcing the Oogs of the world to act as Clan Women; then perhaps they will see the error of their ways and advance out of the stone age of programming.