0100010001000
0101010001000
1000011010101
1101000101001
?

Your on the Street Reporter

Uyless Black

Softistry

Softistry

Contents

Softistry

This is Your on the Street Reporter. We begin this report with a definition:

> **soph·is·try** [sóffistree][1]
> (*plural* **soph·is·tries**)
> *noun*
> 1. **flawed method of argumentation:** a method of argumentation that seems clever but is actually flawed or dishonest.

As suggested by the dictionary, sophistry describes fallacious reasoning. Another dictionary states that sophistry defines casuistry and illogicality. That description might be more understandable with simpler terms, such as: Sophistry is another word for bone-headed.

Borrowing from the word sophistry, I have coined a new word for this story.

> **soft·is·try** [sóftistree][2]
> (*plural* **soft·is·tries**)
> *noun*
> 1. **flawed method of software programming:** a method of writing computer software code that seems clever but is actually flawed.

Because some of the accounts in this story are strange, I emphasize at the onset that all events are true.[3] The names of the participants have been changed to protect their identity and associated sophistic and softistic natures. One more note, despite the weirdness and sometimes sadness of these tales, the final story provides a happy ending.

Softistry Example One

Infallible computers + infallible software + fallacious assumptions = the Black Swan

More often than not, people consider the output from a computer to be incontestable. After all, it came from a computer.[4] The supposedly irrefutable answers are displayed on computer screens or printed-out on hard copy. These descriptions, reflective of our toils and aspirations, have this added credibility: They were not presented to us by an actual human.

But one can protest and say, "Not I! I question a computer's output, especially if it claims I am overdrawn at the bank." Congratulations on your computational astuteness. I claim you are in the minority. This article will provide a single yet significant example of my claim.

The purpose of the example is to illustrate that in spite of all the extraordinary computers we have at our disposal, they are only as astute as we are. They offer only what we offer them. If we offer computers fallacious input, they will offer-back fallacious output.

My statements are so obvious that you may be wondering why I even state them. Read on. You could be in for a jolt.

Computers (specifically the software programs that direct them to do their work) are routinely employed to direct the decision-making of people whose choices often have an enormous impact on all of us. As one example, take the financial world.[5]

---

[1] Microsoft® Encarta® 2006. © 1993-2005 Microsoft Corporation. All rights reserved.
[2] With thanks to Bill.
[3] Conversations in quotes are paraphrased from recollections and notes.
[4] Most of us do not know about the extraordinary talent (sometimes genius) that went into creating that output.
[5] More details on this example are in *The Nearly Perfect Storm: An American Financial and Social Failure*, IEI Press, to be available on-line and in book stores soon…I hope.

The financial industry relies on computer-based software to provide data about the risks of making trades on Wall Street. The most widely used risk assessment program is called Value at Risk, or VaR. It is complex but its output is as simple as it can be. It expresses risk as a single number and as a dollar figure.

Behind this simplicity is a sophisticated software system. It models portfolios consisting of bonds and other financial instruments. It takes into account volatility and diversification.

Because of its flexibility and seeming accuracy, prior to the 2008 financial meltdown, VaR was used by banks and other financial institutions to decide what to place in their portfolios, as well as to establish their capital requirements.

It became such a common tool that a report was handed-out to finance personnel every day, just after the stock market closed. They used it to determine each trading desk's estimated profit and loss in comparison to the risks taken, and how each desk tallied-up for the firm.

Software Heaven! Suppose a software model could predict 99% of the time if we would lose or make money on a day's activities of selling and buying certain financial instruments? This idea sounds attractive and VaR did just that. It measured risk along a normal distribution curve (the bell-shaped curve).

It was a godsend for financial traders and their companies. Its value was aptly described by a modeling expert, "In a thirsty world filled with hydrogen and oxygen, someone had finally worked out how to synthesize H_20."[6]

Financial nirvana! The almost unbelievable complexities of Wall Street's toys were rendered into a simplistic correlation of the price behavior of stocks and bonds. "It is as if you had a formula for working out the price of a fruit salad from the prices of the apples and oranges that went into it."[7]

The Black Swan. The trouble with VaR is that it did not model events that could indeed come about, but were known to be so rare that it was not worth the trouble to write software code to model their effects. For example, VaR did not take into account the common-sense notion that granting millions of loans to people who could not afford them in the first place would cause a problem to millions of citizens, their creditors, and eventually, the entire economy.

Imagine. VaR's elite and affluent programming personnel---being impervious to the financial vicissitudes of life---could not envision a real world in which people actually existed who could not afford to buy a house in the first place.

These thoughts relate to the use of a term: the *black swan*; a metaphor for an event that lies outside conventional expectations but carries extreme impacts. These impacts are only explainable after the dust has settled, the casualties counted, and the black swan has swum past the wreckage.

The VaR software was flawed. It was a code-driven black swan. Black swans come by on occasion; always have, always will. But for this story, the software programmers chose to ignore them.

As stated earlier, infallible computers + infallible software + fallacious assumptions = the Black Swan.

---

[6] *The Economist*, January 24, 2009, p 12.
[7] *The Economist*, ibid.

Not quite; the programmers (at the direction of the financial gurus) wrote fallacious software based on fallacious assumptions. The results proved ruinous to millions of innocent bystanders.[8]

Softistry Example Two
Underselling the complexity of software programming

For this report, we make a 180 degree turn in the description of software and its complexity. Even for this writer, the story is strange. It harkens back to a past that will likely astound the readers of this essay who are software literate.

But something *now* must start from something *past*. After all, if something *now* were the same of something *past*, it could not have been started in the first place.

Nonetheless, I wrote this story, yet each time I read it, I remain bemused by those past days of software doing only "doing payroll" and some other bookkeeping functions.

Background. During the early days of the emerging mass-market computer industry---just before computers entered into the mainstream of our lives in the late 1960s and early 1970s---my vocation of computer software programming was treated as (a) the practice of a sophisticated, white collar alchemy and/or (b) the practice of a simplistic, low-level trade.

Regarding the latter view, I am still amazed when I recall the TV ads aired about computer trade schools in the 1970s. In addition to training on lawn motor repair, selling real estate, and wiring light sockets, several schools added computer courses to their curriculum and encouraged viewers to rush down to their campus and sign up for software programming classes.

It is not my intention to disparage those who repair lawn mowers, sell real estate, or fix lamps, but the fact remains that writing software (computer programming) is a bit more involved than what was suggested in those TV ads. Yet because of hard-sell advertisements and unscrupulous sales people, the software industry attracted droves of wannabe programmers who were ill-equipped for the profession.

Programming Assignment. In 1972, I took-on a part-time, evening position as a programming instructor at the Arlington, Virginia branch of the Computer Learning Institute (CLI), a school offering courses in various trades, including programming. This story focuses on the marketing and sales tactics of this company that took place in those days.

After two hours of lectures on the basics of the FORTRAN computer programming language, I assigned the students their first coding exercise, "Students (and future geeks (unstated joke)), your attention please. I am passing-out one IBM punched card. It has two numbers punched into it: 10 and 15. These values represent the length and width of a rectangular room. You are to write a program that reads the values from the card, calculates the square footage of the room, and prints your answer as, 'This room size is xx square feet,' where xx represents the computer's answer. Any questions?"

None. The problem itself was intellectually dim but irrelevant to the goal. The goal was to write software; to learn to code; to learn how to *submit* that code to the computer in a language the computer could *understand*. It was not intended to uncover the geometrical mysteries of square footage.

The students were eager to begin writing software. After all, they had paid CLI a hefty fee to learn programming, or as they say today, software engineering.

After the class, Joe (so-named in this story) approached me, "Mr. Black, I don't know how to calculate square feet."

[8] Not only because of VaR, see footnote 4.

Silence. ... I had to say something, but in order to form a response I needed to get my chin off the floor. Before Joe was granted admission to this school, he supposedly passed a screening exam. Among other skills, this test revealed his programming aptitude. As a programming instructor, I assumed my students would have a rudimentary knowledge of, say, the relationships of rectangles' dimensions to their area. Not of the simple formula, but of an ability to understand its abstraction.

Ha, you say. How often must a programmer write code for computing square footage? Let me explain. A programmer's lack of knowledge about how to write software to emulate the real world results in the software creating incorrect results. These results are distortions of that world. Thus, not only does the software do no good, it usually does harm. In the end, someone must have an understanding of what the lines of code in a computer software program actually do.

I formed an answer, "Joe, the square footage of a rectangle is computed by multiplying the length and width of two non-parallel sides of the rectangle. Your program needs three executable statements. One statement reads the two values from the card into the computer. The second statement performs the calculation. The third statement prints the answer. So, you need to punch-in to the cards those three instructions."[9]

"OK, thanks Mr. Black. I'll get to work on it."

Programming Problems. After this class had convened for its next session, I discovered the students had coded their program and most had successfully run it on the CLI computer. Joe was an exception. As I walked around the classroom, examining program listings and outputs, I noticed Joe had nothing on his desk except his notes, a programming manual, and the single punched card of the room dimensions---no listing of the program he was supposed to have coded. He was busy reading his notes and manual. He had his head buried in them, as if he wanted to avoid the other students and me. I sensed his discomfort and waited for the end of the class to approach him, "Joe, how is the exercise coming along?"

"Not so good, Mr. Black. I don't understand how to approach the problem. I don't know how to get those numbers into the computer…then get the answer back out of the computer."

I suspected Joe's problems stemmed from the school itself, and not Joe.[10] First, I thought he had likely received inadequate training during his enrollment in the prerequisite to my programming class. The earlier training was titled something on the order of, "Computer Fundamentals."

The instructor for Joe's introductory class had subsequently been fired. Prior to this firing, he should have failed Joe in this first class. Second, the salesman who administered Joe's screening test and who received a bonus for every successfully-enrolled student, had likely faked the results of the test. Rumors floated around the instructor staff at CLI about the less-than-rigorous admission requirements of the school. In my mind, Joe's problems with writing an almost comically simple program confirmed these rumors.

---

[9] For the software coders and programming historians, you know other non-FORTRAN punch cards were used to setup the card reader and the printer; the Job Control Language (JCL) cards. Also, a stop run FORTRAN card was needed. Joe knew about the stop run card, but a JCL manual---as far as Joe was concerned---could just have easily resided in the mystery section of a library.

[10] Don't mistake my empathy for Joe and my blame toward the school as suggesting I favor an educational institution that succors those who cannot meet its curriculum requirements. But it is another matter when the student's requirements for enrolling in the curriculum have been falsely advertised.

By a cursory examination of Joe's mental skills, it was obvious he had been taken-in by the school. Joe should not have been accepted for the programming training curriculum. Later, I learned he had been enticed into CLI's doors by TV ads promising a high-paying job and financial security.

"Joe, I've another class coming-up shortly. If you are free Saturday morning, let's meet here. I'll take you through a review of some of the material in your introduction class. Then, we can take another look at programming in relation to the earlier material. What say?"

"Great! I'll be here. Thanks, Mr. Black."

Joe showed up on Saturday morning, and we began a four hour tutorial on computers and programming. It was slow-going, and I am understating our progress. Joe had no ability whatsoever to grasp abstract concepts. He was only comfortable with concrete images---those he could see or feel. I had never encountered anyone like Joe. Of course, I had not spent much time in my life trading ideas with the Joes of this world about representing the numbers 10 and 15 as electrically-charged pieces of silicon and presenting them as symbols in a computer program.

For our encounter, it did not matter. As the morning morphed into the afternoon, and as I morphed from an empathetic teacher into a frustrated drill sergeant, I attempted one more time to help Joe bridge the gap between the physical world of the punched card and the abstract world of software.

Finally, after I had tacitly given up, and was looking for a way to inform Joe to find another line of work, my student's face brightened and he exclaimed, "Mr. Black, I get the idea now. I have the answer!"

He picked up the punched card, which of course, had the numbers 10 and 15 punched into the card. He also picked up a pencil and *wrote on this card*, to the right of the punched holes, the number 150. He put the pencil down, and handed the card to me, waiting for my critique.

Silence…more silence… still more silence. *Chin up…say something to your student. He's looking at you, expecting a response*...."Joe, your approach to solving the problem is not correct. Look, we have given it our best shot. I don't think you should try to be a programmer. I think your aptitudes lie elsewhere. Come Monday, I am talking to the school officials---your tuition fee should be refunded. It's not a big deal. Some folks have programming aptitudes and some do not.[11] Some minds work one way, some work another. For example, I know of some brilliant engineers who flunked their foreign language classes in school."

Joe responded, "Frankly Mr. Black, that's a relief to me. I've been miserable since I started this school. Nothing makes sense…and I borrowed the tuition money to attend these classes."

The following Monday, I paid a visit to CLI's administration offices, where I asked for Joe's file. It was handed to me; I studied it, making the following observations, and taking the following actions:

- Joe's admission test scores were well above the acceptance levels.
- I retested Joe, using the same test.
- He failed miserably.

[11] Some people think abstractly, some do not; some deal with representational symbols; some do not. Some humans play the piano; others play with themselves. Thank God for diversity, it's what keeps us humans interesting to one another.

The school and Joe took the following actions:

- The school admitted an "error" in its testing procedures.
- Joe was not given a refund.
- He was transferred to the computer operator curriculum, one in which he did not program the computer, but fed-in the IBM cards that did program the computer. Joe became a computer operator. He graduated from CLI with a "degree" in operating the IBM 360 computer. CLI kept Joe's money.

Epilogue. Thus, Joe learned to operate a first-generation, low-function IBM 360 computer (with the simple DOS operating system). He also received extensive "training" on punch card machines. Consequently, he punched cards for the students who were enrolled in the programming curriculum.

Joe was one among several of my students who were victims of the fallacious assumption that programming was a simple profession. I am not suggesting writing code in a programming language is beyond the mental capacity of most people. Certainly, an above-average mind, one with the ability to think abstractly, can "get along" in the software programming profession. Unfortunately, the early computer software programming trade schools exploited the largely unknown nature of software programming, and exploited an unwary and unprepared public.

By continuing to teach at CLI, and now aware of their admission ploys, I had become part of the conspiracy. I quit the job and looked around for other part-time teaching opportunities and found two fine programs around the Washington DC area that accepted me on their faculties.

Sidebar: The Past is Past. Now is Now.

A Web search reveals a company of the same name as CLI that is in business. I was under the impression it went under. I have no idea about the current CLI or if it is still the same CLI that I worked-for in the 1970s. If it is, I can only surmise that it has cleaned up its act. That being the case, I wish CLI well and hope you are training future software rock stars.

Softistry Example Three
Who cares if it has no output, it's not supposed to!

Frank (a fabricated name) was a brilliant man. He graduated with honors with an electrical engineering degree from a prestigious Pennsylvania university. His love of programming guided him away from hardware, into software, and into our office as my office mate. Frank and I were working with other programmers to debug (which means correcting design and coding errors) a software warfare simulation game. This system modeled navy submarine battles between China and the U.S; principally in maneuvering U.S destroyers to positions near Chinese submarines where depth charges could be released.[12]

In the 1970s, as computers became the "in thing," the Department of Defense took the stand that practically everything in its arsenal, and then some, had to be simulated on the computer. For example, another team in our company wrote programs to model the flow of traffic (people walking) in the hallways of the Pentagon---under a fat DOD contract (and our tax money).

The members of our programming team knew Frank was a bright man. During group meetings, he often came up with creative and insightful solutions to a problem. He was also well-

[12] It was a relatively simple set of software, because (at that time) China's submarine fleet was almost nonexistent.

versed in several computer languages (the JAVAs of those years) and computer operating systems (the Windows of those years).

But from a social standpoint, Frank was not very attractive. Not that he was ugly but he was ill-kempt. Unfortunately for me, especially because I had (and have) a highly developed sense of smell, Frank had an extraordinary BO countenance about him. During this time in my life, I became convinced the speed of odor propagating through air was faster than the speed of sound traveling through this same medium. I came to believe I could smell Frank coming down the hallway toward our office before I heard his voice or footsteps echoing in the corridor. His olfactory presence seemed to precede his aural entry. Perhaps I had been conditioned, like Pavlov's dog, to respond to a stimulus: I heard Frank's footsteps, and in my mind, I started smelling his body odors.

In addition to Frank's extraordinary armpits, his mind operated on a plane that did not intersect with the mental processes found in other brains on this planet. For certain, he was smart. Equally certain, he was oblivious to his day-to-day surroundings and forgetful about his project deadlines. Not that he was lazy, nor was he dismissive of our team's need for his work. His mind was just the opposite of the mind of Joe, the CLI student. Joe's mind was about the nuts and bolts of life. In contrast, Frank's mind operated in an ephemeral, abstract world, rarely positing itself in reality.

As our deadlines for completing the submarine warfare model came near, the project leader became concerned Frank's software would not be completed. Frank had finished parts of his assignments but he had yet to show us his code for the remainder of his routines.

One morning, a few days before our team was to demonstrate the results of our efforts to our customer (the Advanced Research Projects Agency, ARPA), I came into the office and found my officemate sleeping at this desk. This practice was not unusual for Frank. I didn't like the arrangement because his continuous, overnight presence did not allow the (otherwise vacant) office to undergo its nightly nosegay. Anyway, I nudged Frank. He groused a bit, then suddenly turned to me and almost shouted, "Uyless, I made a break-through in programming last night!"

"Great Frank. Did you finish the routine on the random walk generator for the submarines? We need it to test the emergency sortie program for the ships."

"Nope. I'll get the generator code done later. Last night I wrote a program that: [*note: To keep this narrative comprehensible to a nonprogrammer reader, I have omitted the details of the program.*][13]

"…And guess what? It executed correctly the first time it ran on the computer!"

I challenged Frank, "No way! I doubt you even compiled the first time you ran it. It's damn-near impossible to write a program of the complexity you describe that runs error-free on its first execution. Sorry, I don't buy it. Let's see the program's output."

Frank, "I coded no Write statements for the program…it produces no output."

Silence…more silence… still more silence. "Frank, the purpose of a computer program is to produce output. That is the reason for its creation in the first place. That is the only reason you and I are employed as programmers: to produce software that produces output! Even more, how can you be sure your computations and your conditional statements are correct if you can't check them?"

[13] For the programming reader, a few comments. The program contained approximately twenty conditional branching statements; some with branches to the middle of supposedly stand-alone functions; some with branches back to other conditional statements. It was complex.

Frank responded, "The computations are irrelevant if there is no intent to 'output' them. The conditional statements are correct and acceptable to the compiler if I received no system diagnostics on them, which I did not. And the program is correct if it is specifically designed to produce no output, and it indeed produces no output. Therefore, my program is correct *and* it ran correctly on its first attempt!"

Epilogue. Somewhere on this planet, there likely exists a philosophical debating society dedicated to confronting deep, abstract ruminations such as Frank's existential, metaphysical, discarnate, transcendental theories on the virtual output of a non-virtual machine. I hope I never come into contact with this society because I am almost Ben Franklinesqe in my approach to software design.

And somewhere on this planet, there likely exists other people like Frank, writing programs that produce nothing, or more probable, nothing of value. I hope their paychecks are directly proportional to their computer printouts.

Who knows? These geeks may be (read, *likely*) writing their software while inhabiting the think tanks along Massachusetts Avenue in our nation's capital. They may be (read, *likely*) collecting scads of taxpayer money by writing sofistic software that emulates the emulation of emulation; a Willy Wonka software factory, worthy of a government grant.

They could be those people on Wall Street who wrote code that supposedly predicted our nation's finances…with the minor exception of the Black Swan. It is code that cannot possibly be completely correct as the system it mimics is far too complex to be modeled. Yet, non-technical executives, sitting in high-rise offices in Wall Street read the print-outs of these systems and take them as gospel.

As for Frank? After completion of the project, he was transferred to another team and thanks to the nose gods, another office.

Softistry Example Four
Who cares if it's incorrect, it's fast as hell!

For one of my early programming projects, our team was competing with another group (from a competing software company) to write experimental code calculating the relative movement of a submarine to a surface ship. The input data to the program would come from an operator entering the results of radar pings. The ability to determine relative positions of one craft to another ship is very important in plotting escape paths, attack routes, as well as normal maneuvering operations.

In those early days of computing (the late 1960s) the project was spectacularly ambitious, but the U.S. Department of Defense had deep pockets and funded many "push the envelope" gambles. I participated in more than one of them; another being the destroyer/submarine model described in the previous story.

We were competing to win an overall contract, one considerably larger in scope than the relative motion software. Therefore, this initial competition was key to winning the overall deal.

The software programs had to be very "tight," a programming term describing efficient code, with no "lazy" statements (easy to write code that was inefficient partially because it was easy to write). In addition, the code could not consume a lot of memory. The programs were to run on a small UNIVAC machine installed on a submarine. This computer was also slow, so the

efficiency of our software had to compensate for slowness of the hardware's execution of our software.[14]

Both programming teams had completed their initial coding and testing. We were to demonstrate our respective solutions to our customer, a submarine warfare planning group, headed by a Navy Captain. After assembling in a conference room, we began our presentations, which included the results of our simulations and test runs.

The results (printouts) of our respective teams were examined by each other and our customer. As they were studied, the project leader for the competing team made this observation, "Your code is 30% slower in its execution than ours. It also consumes more memory; looks like about 5% more. We don't see why your software should be accepted over our code. After all, it has failed on two of our customer's key requirements: speed and memory consumption."

Our project leader responded, "True. We have some 'tightening-up' to do." But one other customer requirement, one we think is somewhat important, has not been satisfied by your system, at least by the printouts we see before us. ….Your answers are wrong! Your boats and ships are moving about as if they had no rudders. Given the navigational problems, our vessels are successfully maneuvering from harm's way. In your model, our vessels have sailed into a perilous position in regard to an enemy."

Their project leader responded, "We understand this *small aspect* (my emphasis, not his) of our coding is not complete. But we hold that the problem of correctness is secondary to the problems of speed and memory consumption. We think we attacked the most formidable problems first. On our next review, we will have the correct answers as well as the more efficient execution."

In exasperation, our project leader remonstrated, "But your fast and efficient system produces incorrect answers! It is essentially worthless. Ours at least works!"

Yep. Silence…more silence… still more silence. I sat through this debate in a stunned, silent mood. I could not decide if the other programming team was using a brilliant, unknown PERT chart for project management or was pioneering a new computer science discipline, yet to be published in the *Harvard Business Review*.

The ending to this story is more surreal than the meeting I just described. Our customer accepted the code from the other project team, with the "requirement" we turn our code over to them. We pleaded our case with the argument we could hone our software to meet the speed and memory requirements. To no avail.

Although I was in the U.S. Navy, my project team was administered and funded by a private company. In disgust, this company withdrew from the contract biding and our code went down with the ship.

Epilogue. Our project team was broken-up. I returned to a programming assignment funded and controlled by the ARPAnet---the forerunner to the Internet. Others went their ways as well. But we stayed in touch and learned about the results of the other software project team---those who coded the fast, efficient, and totally incorrect system. After completing the *small aspect* of producing correct output, their system required more memory and more computer power than the UNIVAC possessed. The project was scrapped, one of scores of such softistic failures I witnessed in the early days of programming.

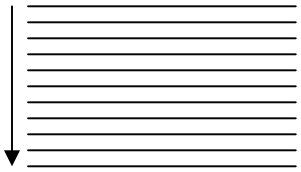
[14] The computer had only a few kilobytes of memory. Our code was written in machine language, with references to absolute memory locations. Later, we joked that the "smallness" of the computer was its location on a small vessel, which was actually partially true. Memory took-up a lot of space in those days and created tremendous heat.

Softistry Example Five
Who cares if it's incomprehensible, it's elegantly constructed!

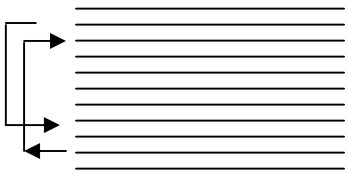
Perhaps because of my undergraduate work in psychology, I approached software design and coding based on visual and logical comprehensibility, attributes of several ideas on the theory of learning. My opinion, then and now, is that code is much more valuable if it can be easily read and comprehended by someone other than the programmer who coded the software. After all, an operational system is often maintained by programmers who did not write the code in the first place. If the code cannot be read easily, it cannot be easily maintained. Worse, the maintenance changes may create errors, because the effect of the changes cannot be predicted with complete accuracy. Try that idea out on the computers and software operating the brakes in your car, or the wing flaps on the airplane in which you are flying.

My first published work in the computer industry was a short article in a long-defunct trade journal, *Infosystems*, titled "Psychology Applied to Programming." It received more correspondence than expected by the magazine publisher because the article proved to be (at that time) controversial. (Many years later, I am happy to report a number of my ideas published in this article are still embedded in textbooks on software design.) The foundations for the article were built around Gestalt theories of learning, named after a psychologist who pioneered the ideas.

Comprehensible and incomprehensible code. I will use a simple example to explain the gist of the article and some ideas on programming. Let's pretend the figure below represents the text in this essay, the lines symbolize the sentences in paragraphs, and the arrows represent the order in which you read the text.



The order of reading is top to bottom; we begin at the beginning of the story and finish at the end. Brilliant! What other choice is there? Plenty. Let's look at the next figure. Notice the arrows show us moving around in the text of the story. We go to one part of the story, then we go to another, and so on. After a while, it is easy to become confused.



Poor (perhaps incomprehensible) software code executes in a similar manner to our reading this scrambled story. The execution has the program "branching-out" and creating complex paths through the code. If this simple illustration is multiplied many-fold, a software system can become very complex. Eventually, changes may create unintended and incorrect results, say, an aileron on your airplane failing, or heat shielding falling off space shuttles.

But you say, *why would a programmer want to create logic paths such as those shown in the second figure*? In today's environment, with fast computers containing billions of bytes of memory, the problem is not nearly as pronounced as it was thirty years ago. In the older days,

these loopback arrows signified the *reuse* of code (with slight alterations) or the *reuse* of memory---a very important practice when using slow, small computers.

Notwithstanding these needs, the ideas of simplicity still held in those days, and techniques were developed to adhere to the model shown in the first figure. Yet, some sophistic programmers continued with their softistry, and I suspect some (drone) programmers still code with this style.

Now, let's return to the mainline of our story.

Good, dumb luck. As a new employee of the Federal Reserve Board, I had the good fortunate to be chosen as the lead (and only) programmer for a new system to model the money supply of the United States. It would replace a manual procedure requiring several economists and clerks many hours each week to compute data and information for the Fed's Federal Open Market Committee (FOMC) deliberations. The data was used by the Fed authorities to increase or decrease the money supply, as well as to set interest rates. It is an understatement to say the system was important to the U.S. economy (and every citizen in the country).

I was selected for this critical programming assignment because: (a) no one else wanted the job, (b) I was new and had nothing to do. In the early 1970s, the economists in charge of the manual system detested the Data Processing Department. In the past, they had been given tardy systems that if they ran at all, produced poor results. In those days, spotting a programmer for the Data Processing Department in the economist area of the Fed building was as rare as spotting a happy data processing customer.

In a nutshell, I succeeded in writing the code and getting the system up and running prior to the deadline. What is more, the system had output! What is even more, the output was correct! I was lauded by the (now) happy economists, given promotions, and allowed to screw-up later systems with relative impunity (such is the importance of "initial impressions").[15]

The point of my Fed story for this essay on softistry is I was assigned to other duties, and my code was turned over to another programmer, William. This man espoused the opposite view of my philosophy: he loved programming complex, elegant code. (At least he believed, unlike Frank, that software should have output.) He was smart; he met his deadlines; for a while, his code produced the correct results---three valuable traits for a programmer to possess. However, his softistic approach led to serious problems.

Crisis at the Fed. You may recall that during the early 1970s, an OPEC-induced oil emergency created a national financial crisis. The Federal Reserve became a key player in this drama, because it was expected to keep the nation's economic and financial boat from floundering. The money supply program was a key tool used by the Fed during this crisis.

Unfortunately, since my departure from the project, William had made a number of user-requested enhancements, as well as his own "improvements" to my prosaic programming. The gradual result of these changes was unstable code---a scary way to describe software. Unstable code means it cannot be trusted. Its output may not be correct. For example, using our airplane example, the pilot directs the plane to go up into the sky, and the software directs the plane to go down into the ground. Certainly, this example is extreme, but I wager you have read news

[15] Don't mistake me for a financial or economics whiz. After the first meeting with my users, I was confused and intimidated. Lucky for me, I had a brilliant team leader who guided me through Money Supply 101, and some wonderful users in the Banking Section at the Federal Reserve Board, who showed immense patience during my learning about money supply principles.

articles of equally bizarre, perhaps catastrophic events that were precipitated because of unstable software.

During this crisis, the Board of Governors and the FOMC directed William (and now, some other programmers) to make changes to the money supply programs to reflect some of its contingency plans. The changes were made, the economists picked up the print-outs to analyze the results. The output looked strange. William was asked to check his changes. He did and discovered they had introduced bugs into the (heretofore) correct system.

Epilogue. The economists did not meet their deadlines to the Board of Governors and the FOMC, and as a result, the economists began to hate the Division of Data Processing again. In addition, this code was placed in a highly secured mode of operation. It could not be altered unless the code had been reviewed by a design committee, composed of users and (Ben Franklinesque-type) programmers.

Due to softistry, everyone in this tale got a black eye. But in fairness, in the 1970s, peer review of code was not yet an accepted way to spend expensive programmer time. Design disciplines and coding techniques had not yet made their way into the software industry.

Today, things have changed. Code reviews, proven design techniques, software execution modeling---all have improved the reliability of software. Yet, software sophistry and software errors continue to haunt our everyday lives. Just look at today's luxury lines of automobiles, cars that should run faultlessly. They are plagued with software errors, some simply stop running on the freeway. Recently, a $90,000 beauty, stalled on the side of a road because over-zealous software sophists have sold the car companies on the infallibility of software.

(Anti) Softistry Example Six:

Understanding the purpose of software *and* hardware!

During the time I was preparing for my first job as a programming instructor (and the subsequent encounter with Joe, the student of square footage fame), my girl friend (let's call her Irene) volunteered to help me prepare for my upcoming lectures. She said she would listen to my mock presentations and determine if they made sense to her. I was appreciative, yet skeptical of her suggestion. She did not know the difference between a computer and a toaster. Come to think of it, she was not much of a whiz with a toaster either. But she did have very fine personality and a lovely countenance. (Besides, I disagree with the idea that a way to man's love is through his stomach.)

Anyway, I did not wish to offend her by refusing the kind offer. So I bundled up my lecture notes and headed for her apartment. There, as we sat down on her sofa, I spread my notes before her, and began my introductory lecture about IBM software and hardware. During these first few moments, I became so immersed in my presentation I failed to realize she was formulating plans for a different delivery.

After I made some remarks about using software to manipulate hardware, Irene posed a question, "So, why is hardware called hardware?"

"Irene, as my programming student, you are supposed to know the answer to your question."

"OK, but I don't know. So, why is hardware called hardware?"

"Hmm. Because it is hard and rigid. Like steel, iron, things like that. A computer is made of hardware. It's not pliant."

"So, why is software called software?"

"Good question. I think someone made up the term to contrast how software behaves in relation to hardware. Software is pliant, flexible---really the opposite of hardware."

"So, software is soft, and hardware is hard?"

"Well, yes…hardware is literally hard, and software is figuratively soft. …Hm, what's your point?"

Then, it began to dawn on my obtuse mind that my "student" had other lectures in mind. It became crystal-clear with her next question, "Well, my point is….are you hardware or software?"

Silence…but not for long. "It depends on who is doing the programming."

"What if I am the programmer?"

And I told you this story had a happy ending.