The Minimum You Need to Know

About Logic to Work in IT

By Roland Hughes

Logikal Solutions

ISBN 0-9770866-2-3
ISBN-13 978-0-9770866-2-7

Acknowledgments

I really would like to dedicate this book to the students about to embark on a career in IT. Having spent nearly 20 years in this industry myself, I felt I owed it to the following generations to put back some knowledge that has been lost in this industry. It is an obligation of those who become seasoned in this industry to pass that knowledge onto the following generations. This book is my attempt to pass some of the most important, yet least taught, knowledge onto you, the reader.

Welcome to IT.

Source Code License

Unlike the other books in this series, there will be no source code for you to worry about. While we will create diagrams and pseudocode, there will be nothing you can actually compile and run.

Table of Contents

Introduction

I.1 Why Logic?

I could start playing the game with you and respond with "Why not logic?", but it is too soon in this book to tick you off; there will be time enough for that later. This book came about for many reasons. It is not one I originally intended to write as part of this short series, if ever. Many circumstances coalesced to allow time enough for it, but there still had to be a need for this book to justify writing it. The need came from a very odd place (at least for those of you unfamiliar with the ways of IT professionals): conversations in bars.

Oh, this wasn't conversations with a bunch of grizzled coworkers swilling suds and having a gripe session. These were conversations with college students when I actually ran into one who was enrolled in some aspect of IT. My current consulting contract has me staying near a college town, so when you head out for a beverage or eight, you inevitably run into college students even in the old fogey bars. What was appalling wasn't the fact that in a bar with more than 300 people you only ran into, at most, one IT student, but what they were being taught. Logic isn't required any more at a lot of campuses. Many others don't even offer it as a class.

In today's pointy-clicky world, logic has been tossed aside. The trouble with tossing aside logic is that you toss aside what is the fundamental core of application and system design. Once you understand that logic is no longer being taught, it is easy to understand the complete lack of design so visible in today's PC and Unix products. Those students who graduate today and actually accept a job working with a 3GL writing non-GUI back end applications flounder miserably. Were there any justice in the world, they would be able to get their money back from the university which gave them the degree.

When I obtained my first degree some two decades ago, I saw the beginning of the end for logic in college courses. They started teaching Pascal in the logic class. The excuse was that students needed to implement the logic tools they were being taught. The second excuse was that they needed to know Pascal to survive the data structures class later in the curriculum. Neither argument was worthy of pushing *any* language into the logic course. Being in the absolute last group of students to take the logic class in its pure form, I'm in a unique position to criticize the following classes. It also helps to have gone back a decade later and heard much the same criticism from former instructors. They claimed that none of the newer students did very well later in the curriculum. That a lot of students changed their major when they started hitting the more coding intense classes.

The reason those students floundered isn't because IT got harder. Indeed, IT had gotten easier by then with good 3GL compilers and syntax checkers. It was because they had burned almost their entire logic class learning a 3GL (Pascal) rather than learning the depths of logic. They were focusing on the syntax of a language. The larger portions of logic, those portions that take you above simple program design into complex application design and simple systems design, were never covered. When they got to the more advanced programming classes, they had no frame of reference. They did not know how to design a control break report, or even what a control break report was. The last part is really sad considering how most college students have a credit card. Every month they get a statement for their credit card account that is a control break report.

Why logic? Because you cannot hope to succeed or even survive in software development without it. Why logic? Because most of you have went through or are going through a college curriculum that didn't teach you logic. Why logic? Because the drive to be the absolute best in software development cannot be taught, but logic can, and you cannot be the best without logic.

In years gone by, hiring a new programmer to your staff was much like buying a new car. You got to pick and choose the features, but you never had to ask if the car came with an engine. Programmers from the old curriculums all had been taught logic, you didn't have to ask. In today's market, you have to ask if the car comes with an engine.

I.2 What is Logic?

That is the fundamental question. There are many answers. Depending on the situation surrounding the question, some answers are more correct than others. When you complete this book and begin working in the real world of IT, you will constantly use a variant of this question with every program and system you design. "What is logical?"

No, I'm not trying to lay a bunch of academic babble on you. People don't buy a book they aren't forced to read because they like having academic babble shoved down their throats. I have found few things more boring than listening to what career academics have said or reading what they have written. There are several periodicals that handle that market. There was a time in my career when I believed I could obtain something useful from them, so I paid expensive fees to read those magazines. What I obtained from them was a substitute for Nytol most of the time.

Logic is the fundamental tool of IT. It is the tool from which all other tools are created. There are many kinds of logic in the world of IT. Scholars would like you to believe that there is only one definition of logic and it always yields the same outcome. That statement is incorrect. They will try to proof their statements with truth tables and lectures that glaze over your eyes inside of 15 minutes. Had there been even the most remote grain of truth to those proofs AI (Artificial Intelligence) would have been more than a mid-80s flash in the pan.

For logic to provide the same answer twice, given the same set of truths, you must start in the same place each time. Logic isn't a set of gears producing the same output every time you turn a crank handle. A single instance of a logic path will provide that type of output. In the world of IT, we tend to refer to that single instance as a program. When a program doesn't provide a consistent set of output, we say it has bugs. The term bugs became shorthand for implementation failure. Every program fails at some point. Because we don't like to say we exist to create failure, we say we exist to create software and some of that software has bugs. It is probably a good thing we call them bugs so the tool to remove them can be called a debugger, otherwise we would be talking about running our programs through "failure removal" instead of running them through the debugger.

Back in the beginning days of computers, they were a collection of tubes and programmed by either wiring or tossing a series of switches. There was only one type of logic: hardware. For most computers it was the "hard-wired" truth. A switch was either on or off. Some computers, though few exist today, used "frequency logic" for lack of a better term. They were analog, not digital. Most of you reading this should be familiar with the classic sine wave from your math classes: the perfectly symmetrical S laid on its side and a line drawn exactly through the middle of it. Everything above the line was positive, or truth, while everything below the line was negative, or false. Analog was much like real life. It allowed for varying degrees of true and false. Some of you using digital cameras may have heard a term called "fuzzy logic." Fuzzy logic tends to get used in a lot of automatic focusing. (Those of you reading this who are intimately familiar with analog computers and fuzzy logic, please allow me some literary freedom here. Conceptually, they are similar as a car and truck are similar. We don't want to go too far down a bunny hole at this point.)

There are still some analog computers around, thought not many. I don't remember the exact problems they were the best at solving, but there is/was a niche of software development which could use no other form of computer. I have a nagging thought in the back of my mind that stellar drift calculations were among the applications best served by analog boxes. If any of you reading this are old enough to have owned the older cell phones, then you have a shining example of analog technology. When you were in a bad spot you could still make out pieces of what the other person said even with all of the static. With modern digital cell phones you simply get a dropped call. Analog keeps going as long as there is any degree of truth; binary stack dumps. Analog knew there was interference and allowed for it; binary requires perfection.

Logic has taken many turns during the past two decades. The study of AI and the creation of truth tables lead to the creation of rules-based systems back in the mid- to-late ‘80s. Truck loads of dollars were poured into companies claiming they would create thinking machines and personal helpers with all of the intelligence of a hired servant in just a few short years once the tables were constructed. There was a “market correction” when none of these start-ups bore the tasty fruit investors were looking to savor. Financially, it was not as horrific as the DOT COM flame out, and it did not have quite as many criminals with their fingers in the venture capital grab bag. It did have one very ugly side effect though — logic got a bad wrap. Colleges pretty much stopped teaching it and IT started a downward spiral.

Logic taken too far gives you problems like the AI investment debacle. What the industry failed to realize is that logic, like cholesterol, must exist. There is good logic and bad logic just like there is good and bad cholesterol.

The short answer is that logic is the most fundamental of IT tools. Logic is what allows you to get from input to output, no matter what. Logic is the framework upon which every computer application is developed, even the object-oriented applications.

I.3 Prerequisites for This Book

This book only requires that you have some interest in computers and software development. It will help if you have poked around with some software development tools or attempted some fundamental programming on your own. By “fundamental” I’m not talking about some GUI development product where you only clicked and dragged things together. I’m talking about older, lower level, C or BASIC programs where you printed “HELLO” to the screen and tried to work out how to print columns of asterisks from 10 down to one so they looked like a crude bar graph. If that last problem sounds horribly simple, try doing it without having had a logic class. Try doing it in under an hour. If you feel like adding insult to injury, try printing numbers going down the left side, along with as solid a bar as you can print and letters going across the bottom with a solid bar above them so it looks even more like a bar graph.

Don’t worry if all you have is a casual interest in computers or software development. We won’t be writing any programs, only mapping out their logic, so you do not even need a computer. You might want to surf the Web or visit an office supply shop and see if you can find a “flowcharting template.” This is a little green plastic thing with different shaped holes cut in it. You use it and a pencil to draw a flowchart on paper.

I.4 Approach of This Book

Most books on programming logic tend to be less than 100 pages in length. The reason for that is they focus on either flowcharting or pseudocoding, then stop. They give you one or two problems that could relate to the real world, then leave you to figure out things on your own. If I'm going to slam those books that hard here, then you can be assured it will not be my approach.

I will cover the boring and dry components of flowcharting and dabble a little with pseudocode. From there we will move forward to some of the standard programming situations. This will let you actually "see" logic in action. It will also allow some "images" to sink into your brains. From there we will move forward into the general approach of application design.

This may be a book whose later chapters you wish to read two or three times, especially if you are new to IT. Nothing you do in your career will save you more time and anguish than allowing the career advice chapter to fully settle in. Trust me on this one. My first year in college I had to take a class on logic. We had two little paperback books. One for flowcharting and one for pseudocoding. Needless to say we completed those in the first month. The entire rest of the semester was class participation. Each session would start with a new problem – usually – and the entire class would be spent with students calling out or suggesting the steps of logic required while the instructor drew with chalk. Admittedly, he had it rough for the first few days of that, but once the bit was set in our teeth, all he had to do was draw and try not to inhale too much chalk dust.

I don't have a classroom setting for you, nor do I wish to create one. I'm sure someone reading this book will do that in the end. What I do have is nearly two decades of software development experience to draw on when creating problems for you. There will be no choice but to start you out with some of the fluffy ones, but I plan to leave you with some really good ones toward the end.

One thing that was always done in the past was to separate analysis and programming logic classes. Now that I'm "long in the tooth," so to speak, I don't agree with that separation. I won't go into all of the high-minded and far-reaching concepts of application and systems analysis. There are an awful lot of other diagrams and specification refinement methods. If you learned them all, each project would have you burning a week up front deciding which method was going to work best for the current project. I will take you through the "common sense" approach, naturally extending program logic to application logic to systems design logic.

I.5 Who Should Read This Book?

Anyone who has even the slightest interest in software development or needs to manage the software development process should read this book. If you are contemplating a career in software development, then this should be one of the first books, if not *the* first book, you read before going too far down that career path. Understanding the most fundamental concepts of the software development process is critical to success in either of those career paths.

I.6 Why is Flowcharting and Pseudocoding Shunned?

I'll be honest. I haven't drawn a program flowchart in nearly two decades. The only time I write what could be called pseudocode for a module is when I have either a really big chunk to chew or I'm writing a bid for approval. Notice that I specifically stated "program flowchart" and "pseudocode for a module." The devil is in the details. When doing application or systems design work, I either draw or participate in the drawing of flowcharts for each project. Once you get into VLA (Very Large Application) or system level designs, you have absolutely no choice. The human mind was not meant to remain stretched around designs of that magnitude for prolonged periods of time. Normally, projects of that size take six months to a year before they even get close to a testing stage. You need really good diagrams to remember what it was you intended to do six months ago so you can set up useful test conditions.

As you progress in the field if IT, you will begin to understand why companies will spend in excess of $10,000 for an ink jet or plotter that uses rolled paper three to four feet wide. One system I helped develop in the past few years had a system flow diagram done from a very high level so management could understand it. It took two sheets of that paper six feet in length taped on the long edge to contain the diagram. When it was scaled down to print on regular letter size paper on a normal ink jet it took close to a dozen sheets of paper taped edge to edge, and that was using fonts no larger than six points inside of the symbols. You couldn't read it if it was more than a foot from your eyes.

Program flowcharting has been shunned because it is a lot of work. A flowchart is only good before you write your first line of code. Once you start testing, you realize there are problems with your design and start making changes to the program. The flowchart never gets updated. Bad documentation is worse than no documentation at all because it leads people in the wrong direction. In the early '80s many shops still endeavored to keep their flowcharts current. Once management got involved and started chanting "cut costs," flowcharts were the first thing to go.

Pseudocode was simpler. You could write it with any text editor or word processor. In many ways it was much like COBOL. Some shops used to write pseudocode with such detail that you could almost put a period at the end of each statement and get it to compile. This level of pseudocode was impossible to maintain and also died after hearing the "cut costs" mantra. High-level pseudocode still gets written in quite a few shops. It usually gets wrapped in a word processing template that calls it a program specification.

In days of old, a new programmer would be given a program specification from a systems analyst, or just an analyst. That specification would range in quality from incredibly detailed specifications you could begin coding from, or just a few lines scribbled on a napkin with a sweat ring from a beverage on it. Laugh all you want at that statement. The most complex systems you will ever get involved with start out as just a few lines on a napkin. They generally take about five years to settle into a company and become stable. The reason you end up working on them is they look so simple and innocent being only a few lines on a napkin.

Once a programmer received a specification for a new program, they would sit down with pencil and paper drawing out a very detailed flowchart. This flowchart would document close to every line in the initial version of the program. They would not be allowed to start coding until the analyst reviewed the flowchart completely and agreed it was correct. When the analyst was too busy to bother with you, they would reject the flowchart outright with the phrase "Needs more detail." This statement was designed to tick you off for a day or two until the analyst had time for you. It really didn't have much to do with what you had done. How do I know this? I was once one of those flowchart drawing programmers who got a flowchart rejected with that exact phrase. A week later I submitted that exact same flowchart and it was accepted. During that week I did flowcharts for the other analysts because they had time to chat with me.

Developers who paid their dues got to be programmer analysts. These were the cherished job titles. Once you became a programmer analyst you never had to draw another program flowchart. You got to code straight from the specification. When the specification was only a few lines on a napkin you got to work with an analyst to flesh out the specification. Developers weren't allowed to reject a specification from an analyst with the phrase "Needs more detail."

The result was that everyone wanted to be a programmer analyst. Everyone wanted to just "throw code at a problem and see what sticks." That was the fun job in IT. Some geeks prided themselves on how close to the metal they could code. Even if the shop had chosen BASIC or COBOL, they would find a way to toss in some Assembler code just to prove how much of a geek they were. The fact that the Assembler code could never port to a new machine, even in the same family of computers, never stopped the geek of geeks from throwing it into the application.

During your stint as a programmer analyst, you would get to cut your teeth writing program specifications. Because you weren't a systems analyst, your specification could never be three lines on a napkin. Normally you would be the one receiving three lines on a napkin and be told to flesh it out. Then your specification would have to be reviewed by an analyst or systems analyst before it would be assigned to a programmer.

Life would then take an ugly turn for you. After one to five years as a programmer analyst you would be promoted to analyst. It was the only way to get further pay increases. You would also be relieved of writing any more code. Instead, your life would be application specifications and programmer management. While you would toggle between writing systems specifications and application specifications, you would never really do systems design. The task of systems design fell to the systems analyst. When it came to bidding on contracts or very large applications, they would have to draw a system flow diagram showing how all of the pieces were to fit together. Filling in the actual details would fall to those below. The only code you would ever get to write was the occasional operating system command script.

Most programmers went into IT not only for the money, but because they really liked the idea of coding up solutions to problems. Once they got to the skill level of programmer analyst, most would change jobs. Indeed, the average length of stay when I was at that level was two years. Developers who loved to program simply didn't want to give it up. The only way to avoid it was to leave the company.

Never let it be said that marketing won't sell you a product that will kill you. Along came case tools followed by pick and drag code generators. They would market these products claiming great leaps in productivity. The tools would have an interface so appealing that even an MBA could generate the sample contact manager program in under half an hour. That poisoned apple sure tasted sweet to upper management. They threw tractor trailers full of money at the vendors of those products. Once PCs with graphical desktops came out, an even bigger slew of "visual" type tools flooded the market. Every one of them promised to make programmers more productive. Every one of them cost the industry more than will ever truly be known.

The argument was made on college campuses that students would never have to flowchart in real life, so why make them do it now? Grading those flowcharts certainly took a lot of time. Drawing flowcharts was a lot of work. It was agreed that flowcharts would be dropped. Because pseudocoding was taught in the same class as flowcharting, it died a death by association.

Perhaps the saddest part of this story is that nobody with any clout stood up and answered the question "Why make them do it now?" Had anyone answered that question, it wouldn't have been dropped. Specification writing would have been added to each and every language/tool class added to the curriculum and IT would have remained a shining star instead of a downwardly spiraling industry.

Flowcharting is shunned because it is a lot of work. College students want to do nothing but party when they first get away from home. Nobody wanted to spend their evenings with a table full of eraser crumbs, which is pretty much what a class requiring flowcharting made of their evenings.

Flowcharting is shunned because management didn't want to eat the cost of training a developer who was only going to spend two years at their company. They expected colleges to do that. When colleges failed, they threw money at tools to eliminate the need, or so they thought.

Flowcharting is shunned because this is a Visa/MasterCard society. Buy now, rack up enough debt to declare bankruptcy, and avoid paying altogether.

Pseudocoding at the program level died with flowcharting. It survives in a watered down form at the much higher level of program specification. Until you see actual application pseudocode, you won't understand just how watered down it is.

Nobody wants to pay their dues. That first year of flowcharting is horrible. If you don't drink, you will find yourself starting. The first few months are bad because you aren't any good at logic then. You understand some of it, but don't really know how to put it all together. By the end of the first year, you were approaching the level of quality the IT industry needed: The kind of individual who could hear a problem and begin the design in their head. Not an entire application or system problem, but a single module or program problem. You would find yourself looking forward to hearing newer developers ask, "How am I supposed to do this?" Turning around with a smile you would say "It is accomplished this way" and give a detailed verbal explanation. You looked forward to that day because that was the day all would know you had paid your dues in full.

Flowcharting is shunned because nobody wants to pay their dues.

I.7 Flowcharting and the Current State of IT

Admittedly, this section probably belongs in one of my "Ruminations" chapters, but it fits here. There are few uglier tasks in the world of IT than being asked to draw a flowchart. While it may take you months to track down a bug in a particularly nasty piece of code, there is an immense sense of accomplishment that follows fixing it and pointing the fix out to all of those programmers who shied away from the task. When you are asked to draw a flowchart for a large program, it can feel like being asked to lug rocks from one pile to a new pile ten feet away, then lug them back again.