So You Want to Write an App?

My motive

My first experience with a computer was in High School. In the math lab there was three or four ticker tape reading machines that were linked to a mainframe computer somewhere. I remember someone making me a copy of a computer game about a rocket ship that lands on the moon. The object was not to come in too fast to avoid blowing up on impact. I don't recall having played the game, let alone written any original computer code on those computers. Other students seem to have been more interested and productive on them than I.

A few years later, in 1979, when I was enrolled as an engineering student at UC Santa Barbara. One of the freshman classes was an introductory computer class where we learned a dialect of Fortran to write simple games. I was horrified at the amount of work required to complete a trivial program. For example, we had to create a horse-race game where each of the racing horses advanced so many lengths of track according to a random number to determine the winner. This sounded easy enough, yet to complete the assignment for the following week deadline required incessant daily toil—even an all-night session at the computer lab, while largely neglecting my other classes. I recall handing in my incomplete set of 50 punch cards on the day of the deadline, feeling both ashamed and utterly exhausted.

Despite my desperate straits as an engineering student, I demonstrated a curious programming aptitude: beating my dormitory fellows in game of logic called, "Master Mind." This was a game where the defender sets up an arrangement of colored pins that is hidden from the opponent, who must attempt to guess this order according to feedback by the defender—somewhat in the fashion of the better-known game, "Hangman." My mother had fortuitously given me this game as a holiday gift, although she had no idea what it would mean to my withering ego. Though just a game, I felt it would demonstrate once and for all whether or not I had a low IQ. Though failing as an engineer, I was peerless among my dorm-mates at this test of logic.

Before leaving the topic of my ill-fated experience at UCSB, I should mention that one of the students at my dormitory had purchased an Apple II computer by working a summer job in Cupertino. He booted it up using a cassette tape and the monitor was Sony black-and-white portable TV. This owner boasted that his Apple, though not as powerful, could out-perform the computer lab's Digital minicomputer insomuch as he didn't have to use a punch-card reader and wait in line to compile his program like the rest of us.

Thereafter, I steered clear of anything to do with the then burgeoning home computer phenomenon until about 1997. I was riding my bicycle back from a substitute-teaching job, when I passed by a garage sale with an Apple IIe for sale for $20. This immediately struck a chord with me, though the state-of-the-art had advanced astronomically since 1978, when it first came to my attention. At about this time I was matriculating at San

Jose State’s teaching college in order to become a junior high school teacher. At that time there were many proposals for revising our fledgling educational system, and I was intrigued by one called the Paideia Proposal, consisting of a hierarchy of three curricular levels. The highest level has the student demonstrate their competence in a subject by performing something or submitting a project. The middle level consisted a question and answer exchange between the teacher and students—a dialectical discussion. The lowest level dealt with building academic skills where the teacher coaches the student—such as how to spell. It occurred to me that computers would be ideal tools for succeeding at this third level. A student could, for example, be tested for their gaps in spelling competence. If they needed to memorize the spelling words of the week, they could employ a spelling drill software game, and so on.

My quest to become a middle school teacher, like my quest to become an engineer, also failed. Yet, I decided to fall back on substitute teaching and after-school tutoring. One of the advantages of this employment over a regular teaching job was that it allowed me time to explore teaching resources at the County Office of Education’s library. It was here that I spotted a binder called “The San Mateo County Spelling Study." I immediately recognized this work because the authors were next-door neighbors when I was in elementary school. My neighbors both had PHD's from the University of California. One had her degree in reading development, the other in linguistics. I new that they were assigned the task to compose a complete curriculum for spelling for grades 3 through 12 for San Mateo. I also knew that this work was widely adopted throughout the state and beyond. They had collected the most common English words used by Americans and organized them according to their graphemes (spelling parts) and grade-level suitability. Beside my personal connection to the authors of this work, I was attracted to the quality of the project. It seemed to have been crafted with Shaker-like devotion.

As I alluded, at about this time, I was undergoing a renaissance in my interest for computers and how they might be useful tools for learning skills. Not just that: I found myself passionate about programming. Here I now owned an Apple II and could perhaps create things on it without having to meet some implacable deadline. It occurred to me that one could take advantage of the word-processing power of the computer to capture a student's spelling corrections onto a list so the student retained the particular words they needed to master. Moreover, there was this wonderful spelling curriculum that emphasized important word parts, the mastery of which, a student would become a good speller for all words that contained those constituent patterns. I could write an application that analyzed the list of captured words addressing their particular kinds of spelling deficiencies. This was about 1994—the birthdate for SpellAware—the application that was to occupy so much of my free time for the next 20 years.

The Task

Yes, twenty years. The Great Pyramid took only 10 to 20 years; the Sistine Chapel, less than four; the Empire State building, less than two. A thought that haunts me is that in the same amount of time, I could have personally taught thousands of students the art of spelling. The biggest reason for why it took so long is that I actually completed the project several times over—once for each significant step I made in my hardware purchases. First there was the AppleSoft Basic SpellAware. Next, when I acquired an Apple GS computer, there was the Apple Pascal SpellAware incarnation. Upon buying my first contemporary computer, an iMac, I built a Microsoft Office version using its built-in Visual Basic Automation tools. This new computer also opened the door to composing an AppleScript version, and later augmenting this with an AppleScripts Studio interface. Then, about five years ago, I purchased a Macbook laptop and set about trying to coerce a SpellAware version out of Cocoa and it's related Carbon Universal Interface toolset. The evolution was driven by my desire to advance to the state-of-the-art programming from that of the antiquated late '70's technology where I began. See the following chart for my estimate of how the project grew in relation to the level of sophistication of the hardware and programming language:

Year	Hardware	Software	Lines of Code
1996	Apple IIe	AppleSoft Basic	450
1998	Apple GS	Apple Pascal	900
2000	iMac-G3	AppleScript	?
2004	iMac-G3	Microsoft Office VBA	2,000
2008	MacBook	AppleScript Studio	?
2014	MacBook Air	Cocoa	45,000

The technological stages along the way were probably a blessing. The first programming languages for the personal computer, though barren of features, were elegantly simple. With each upgrade in hardware and software, the size of SpellAware grew by leaps and bounds. This evolution allowed me to graduate along with the technology. Learning to program in AppleSoft Basic was hard. Apple Pascal was harder still, and so on, however, the most difficult jump was into the industrial-strength, modern programming language, ObjectiveC and Cocoa. Had I not competence with the simpler technologies, I don't believe I would have been able to complete a commercially viable SpellAware application.

I look back fondly on my nursery programming experience when writing in AppleSoft Basic. Though primitive by the standards of today, that simple language captured the essence of programming. Whatever you could imagine that lent itself to computing, you could probably craft if you were clever enough. Back in those days, nobody expected sophisticated dropdown menus, drag-and-drop file capability, or the ability to undo changes you make to the interface. Your application was focused solely on the simple tool you set out to provide. I suppose it can be likened to learning to fly an ultra-light airplane where you get the full thrill of flight without the overhead of a full-sized airplane.

There's more to programming than meets the eye

So the promise of more processing power and graphical features for my application drew me onward. Only after I had committed myself to this allure of professionalism did it dawn on me I had made a pact with the Devil. My fall began by taking a couple college courses covering the C language—the basis language for the formidable object-oriented ObjectiveC. Next, I carefully worked through two editions of a popular primer on the vast and thorny subject of the Cocoa API before enrolling in a class. While most of the exercises assigned were too difficult to complete within the school semester, I nevertheless revisited them afterword with dogged devotion. The transition from learning Basic or even plain C programming language to the current object-oriented language, such as ObjectiveC and Cocoa, is like going from Algebra to Calculus. For example, while AppleSoft Basic has about 6 commands with which you could manipulate a string of text, and C perhaps 12, today's Cocoa has over 200.

There is a chimerical aspect to programming: the end always seems near-at-hand when some new unforeseen detail is sure to arise. One reason for this is that with each additional line of code you write, the chances increase that the logical implications of this new line will clash with some other aspect of your code. I venture that the time required to complete a program is, generally speaking, in proportion to the square of the number of lines your program contains. The reason being, that code bristles with contingencies, for unlike the human brain that takes intuition and contextual queues for granted as it decides this or that, a computer is limited to bare Boolean logic where the programmer must anticipate every subtle difference where the data might be greater than, less than, or equal to some other bunch or bit of data in order to produce some practical result.

I was reading a comment of a reader responding to an article posted in the New York Times about a new trend in school curriculum for incorporating programming classes. This person told of an introductory programming class he took at Harvard where the professor wished to emphasize the non-intuitive aspect of writing code. The professor, using a baby doll to illustrate, asked the class to direct him on how to change a baby's diaper. The first procedure the class suggested was to place the baby on something flat, at which the professor took the doll and slammed it against the chalkboard. Of course, the point was the necessity of being exhaustively circumspect when you are instructing a computer without human-like understanding. I can attest that any student who programs will learn this lesson well.

Perhaps the human mind can be likened to a candle in a dark room. It "sees" objects within that room with respect to their proximity and the steadiness of the flame itself. Now suppose you were to place a metal pale over this source of light and you were to puncture this pale here and there to let the light out. With painstaking persistence you might skillfully project letters of the alphabet onto the far wall. Continuing, you could cause a word and even a short message to be written. Likewise, the programmer draws upon his intuition to set down a series of logical statements to form a system that performs some predictable manipulation of symbols. Unlike the programmer himself (except certain "gifted" individuals), his creation has the virtue of being fast, consistent, non-complaining, and narrowly specific.

Another reason your project grows is that you always seem to think of some clever new feature to add. You think, "wouldn't it be cool if…." Nor will today's technology allow you to rest on your laurels: the programming panoply of functions is forever being added to or diminished as the wisdom of its engineers see fit. While writing this article, I have been disappointed to find that one of my scripts used to copy the text of Apple's online word processor no longer works—probably in the name of increased user security. This is but one example of the technical impasses that constantly lurk about like hungry sharks, taking an occasional bite out of your hard-won catch.

To be fair, I should not blame the programming tools alone for my slow progress. After all, I did not pan out as an engineer student in my youth. My brain is probably polarized rightwards, making fastidious synthetic thinking an upward battle. Nor do I own a wonderful memory that is so important for remembering much of the sprawling API of todays programming environments. What I may be blessed with is: leisure time, a dauntless devotion to completing a working product, the love of problem solving, and—when the chips are down—recollection that I excelled at Master Mind.

It's hard to say what my average daily output is but I would guess something like 100 lines. What makes this assessment difficult is coding can be very unpredictable. If you are reworking a section, things go relatively fast. On the other hand, "groking" a new coding mechanism can take several days before you get your desired effect. The most insidious time-killer is tracking down a nasty "bug." For example, you may have wrong-headedly assigned a *mutable* text variable to an *immutable* one, expecting to be able to add characters somewhere else in your code only to find that it remains unchanged. Typically, after an hour or so of "coding Master Mind," and perhaps a walk around the block, you take a deep sigh of relief if you determined the problem. Coding bugs are by no means unusual. The first thing that surprises someone who has never programmed is how rare it is to write a line of code without triggering a complaint by the code compiler. Moreover, this is only the "grammatical" step to making your code do what you wish. The code must also behave according to the logical scheme you have in mind. Writing a function that simply sorts a list of words is a painstaking process despite (or because of) today's exhaustive programming tool kit. A fortuitous and indispensible, resource for the modern programmer is websites devoted to answering nut and bolt questions you have using a particular function. Other generous programmers will respond to your question, so long as you can articulate your problem concisely. More often than not, you have the same question another person already posted and your can read what others have already suggested. What these website do not address is higher-level architectural questions. Only through experience does one develop a nose for knowing the best way to execute some coding task: for example, is it a better design decision to write your application database in CoreData or using XML text files? Should you use AppleScripts to extract text from concurrently running applications or Carbon Universal Access functions? If you have not worked with these entities before and cannot get guidance from someone, you must proceed by time-consuming trial and error.

The programmer must also put on the psychologist hat in order to design a user-friendly interface for an application—the one aspect of programming that has arguably become easier with the advent of the newer programming tools. Back in the day, one would have to drudgingly write detailed code to generate those slick windows that scroll, the drop-down menus, and *undo support*. Nowadays most of these graphical objects are provided as templates that you visually manipulate with the mouse until you get just the look you want. These objects are then codified automatically for you so that they function seamlessly along-side your other hand-written tasks. Sounds cool, however this convenience comes at a price: those pro-looking graphical objects require that the rest of your code accommodates them by conforming to a complex protocol that is seldom intuitive and too intricate to commit to memory.

I wonder how different the world would be if exacting butt-heads—the likes of Steven Jobs (I know I'm biting the hand that feeds)—were obliged to write an application for someone else's approval. I do believe they would be humbled because programming seldom "just works." Rather it tends to behave according to Murphy's Law that if anything can go wrong, it will. Managing other humans has its set of problems. I know this well, having been a substitute teacher for 20 years. Yet there is nothing quite like programming that holds the mirror up to one's ego, exposing its arrogant presumptions. This is not to say that there is any sagacity in the finicky programming language and compiler. On the contrary, all too often things don't work the way they are supposed to and you are obliged to come up with some sort of work-around method (hack) to get the result you desire. For example, my application would have been much simpler to write if its mechanisms could react to the user's opening the auxiliary spell-check menu, as it should. Instead, I have to provide complex methods to continuously monitor the user's typing to get the same effect. More code—more contingencies.

SpellAware would probably never have been created if it were left to market forces. First, SpellAware is a massive application at 16Mb (about a third the size of Microsoft Word). It incorporates a dictionary of several thousand words with grapheme patterns that had to be prescribed by hand. Second, SpellAware requires a special framework of functions that interact with the user's manipulation of the mouse and menus of applications outside of SpellAware itself. To intercept a spelling correction when the user is writing in a Microsoft Word document, SpellAware needs to detect the user's keystrokes and spell-checker devices. Successful behavior of this interactivity is exceptionally difficult to do without interfering with the application's normal behavior. If that weren't enough, Apple discourages the writing of interactive applications by creating a protective "Sandbox" security layer requiring special permissions of the user during the software installation. Moreover, they bar all such interactive applications from being sold on their popular online Apple Store. Finally, academic software is a relatively small market.

They say you are never really done with an application: there are always bugs that need tending to, functions that need to be updated, features to add, and sections that need pruning. Nevertheless, at least for now, I feel like a python having ingested an impossibly big prey. I savor the moment with nary a care for the long and painstaking effort against a most unyielding quarry. Despite its limitations, SpellAware should appeal to the college or foreign language student who wishes to understand their personal word pattern omissions when they spell, for they will not likely want the prevalent academic software with cartoons and artificial games targeted at children to "make learning fun." No, they will want an unobtrusive application that runs in the background collecting their spelling errors. At a convenient moment they can learn from a succinct analysis of this collected data, thanks to the loving devotion of those two dear next-door neighbors and—as I hope I have made clear—my supreme effort in making this possible on today's computer.