Syntax, Semantics, Micronesian cults and Novice Programmers

03/01/2004 • 6 minutes to read

I've had this idea in me for a long time now that I've been struggling with getting out into the blog space. It has to do with the future of programming, declarative languages, Microsoft's language and tools strategy, pedagogic factors for novice and experienced programmers, and a bunch of other stuff. All these things are interrelated in some fairly complex ways. I've come to the realization that I simply do not have time to organize these thoughts into one enormous essay that all hangs together and makes sense. I'm going to do what blogs do best -- write a bunch of (comparatively!) short articles each exploring one aspect of this idea. If I'm redundant and prolix, so be it.

Today I want to blog a bit about novice programmers. In future essays, I'll try to tie that into some ideas about the future of pedagogic languages and languages in general.

Novice programmers reading this: I'd appreciate your feedback on whether this makes sense or it's a bunch of useless theoretical posturing.

Experienced programmers reading this: I'd appreciate your feedback on what you think are the vital concepts that you had to grasp when you were learning to program, and what you stress when you mentor new programmers.

An intern at another company wrote me recently to say "*I am working on a project for an internship that has lead me to some scripting in vbscript. Basically I don't know what I am doing and I was hoping you could help.* " The writer then included a chunk of script and a feature request. I've gotten requests like this many times over the years; there are a lot of novice programmers who use script, for the obvious reason that we designed it to be appealing to novices.

Well, as I wrote last Thursday, there are times when you want to teach an intern to fish, and times when you want to give them a fish. I could give you the line of code that implements the feature you want. And then I could become the feature request server for every intern who doesn't know what they're doing... nope. Not going to happen. Sorry. Down that road lies cargo cult programming, and believe me, you want to avoid that road.

What's cargo cult programming? Let me digress for a moment. The idea comes from a true story, which I will briefly summarize:

During the Second World War, the Americans set up airstrips on various tiny islands in the Pacific. After the war was over and the Americans went home, the natives did a perfectly sensible thing -- they dressed themselves up as ground traffic controllers and waved those sticks around. They mistook cause and effect -- they assumed that the guys waving the sticks were the ones making the planes full of supplies appear, and that if only they could get it right, they could pull the same trick. From our perspective, we know that it's the other way around -- the guys with the sticks are there **because** the planes need them to land. No planes, no guys.

The cargo cultists had the unimportant surface elements right, but did not see enough of the whole picture to succeed. They understood the **form** but not the **content**. There are lots of cargo cult programmers -- **programmers who understand what the code does, but not how it does it**. Therefore, they cannot make meaningful changes to the program. They tend to proceed by making random changes, testing, and changing again until they manage to come up with something that works.

(Incidentally, Richard Feynman wrote a great essay on cargo cult science. Do a web search, you'll find it.)

Beginner programmers: do not go there! Programming courses for beginners often concentrate heavily on getting the syntax right. By "syntax" I mean the actual letters and numbers that make up the program, as opposed to "semantics", which is the meaning of the program. As an analogy, "syntax" is the set of grammar and spelling rules of English, "semantics" is what the sentences mean. Now, obviously, you have to learn the syntax of the language -- unsyntactic programs simply do not run. But what they don't stress in these courses is that **the syntax is the easy part.** The cargo cultists had the syntax -- the formal outward appearance -- of an airstrip down cold, but they sure got the semantics wrong.

To make some more analogies, it's like playing chess. Anyone can learn **how the pieces legally move**. Playing a game where the strategy makes sense is the hard (and interesting) part. **You need to have a very clear idea of the semantics of the problem you're trying to solve, then carefully implement those semantics.**

Every VBScript statement has a meaning. **Understand what the meaning is.** Passing the right arguments in the right order will come with practice, but getting the meaning right

requires thought. You will eventually find that some programming languages have nice syntax and some have irritating syntax, but that it is largely irrelevant. It doesn't matter whether I'm writing a program in VBScript, C, Modula3 or Algol68 -- all these languages have different syntaxes, but very similar semantics. **The semantics *are* the program.**

You also need to understand and use **abstraction**. High-level languages like VBScript already give you a huge amount of abstraction away from the underlying hardware and make it easy to do even more abstract things.

Beginner programmers often do not understand what abstraction is. Here's a silly example. Suppose you needed for some reason to compute 1 + 2 + 3 + .. + n for some integer n. You could write a program like this:

```
n = InputBox("Enter an integer")

Sum = 0
For i = 1 To n
    Sum = Sum + i
Next

MsgBox Sum
```

Now suppose you wanted to do this calculation many times. You could replicate the middle four lines over and over again in your program, or you could **abstract the lines into a named routine**:

```
Function Sum(n)
    Sum = 0
    For i = 1 To n
        Sum = Sum + i
    Next
End Function

n = InputBox("Enter an integer")
MsgBox Sum(n)
```

That is **convenient** -- you can write up routines that make your code look cleaner because you have less duplication. But **convenience is not the real power of abstraction**. The power of abstraction is that **the implementation is now irrelevant to the caller**. One day you realize that your sum function is inefficient, and you can use

Gauss's formula instead. You throw away your old implementation and replace it with the much faster:

```
Function Sum(n)
    Sum = n * (n + 1) / 2
End Function
```

The code which calls the function doesn't need to be changed. If you had not abstracted this operation away, you'd have to change all the places in your code that used the old algorithm.

A study of the history of programming languages reveals that we've been moving steadily towards languages which support more and more powerful abstractions. Machine language abstracts the **electrical signals** in the machine, allowing you to program with **numbers**. Assembly language abstracts the **numbers** into **instructions**. C abstracts the **instructions** into higher concepts like **variables, functions and loops**. C++ abstracts even farther by allowing variables to refer to **classes** which contain both **data and functions that act on the data**. XAML abstracts away the notion of a class by providing a **declarative syntax** for object relationships.

To sum up, Eric's advice for novice programmers is:

- **Don't be a cargo cultist -- understand the meaning and purpose of every line of code before you try to change it.**
- **Understand abstraction, and use it appropriately.**

The rest is just practice.

Comments

- **Anonymous**

February 29, 2004
The comment has been removed

- **Mike**

February 29, 2004
My next piece of advice would be: Learn to use your debugger.

I see it so often on message boards where a novice's code isn't working right, and they