Choosing Languages

Stevey's Drunken Blog Rants™

Have you ever noticed that writing is as hard as programming?

Welcome to my 10th attempt at writing this month's Stevey on Languages column! At *least* the 10th. I've got dead articles galore.

Part of the problem is that I subscribe to the Miami Herald online articles, specifically the Dave Barry ones. If you ever want to get into writing, don't read Dave Barry. You'll write an article that you think is perfectly good, maybe even a little bit funny, and then Dave's column will come out, and you'll be compelled to print out a copy of your now-dead article, just so you can burn it and eat the ashes.

This month, I wanted to give a Whirlwind Tour of all the languages we use at Amazon, saying something funny and yet still insightful and relevant about each of them. Unfortunately, given that we use something like 30 languages here, all I had room for was one-liners, like: "C++: an octopus made by nailing extra legs to a dog. *(Steve Taylor)*." And: "When C++ is your hammer, everything starts to look like your thumb. *(Amazon interview candidate)*". Insightful perhaps, and relevant, but not the makings of a good article.

I also wanted to talk about static vs. dynamic typing, and pattern matching syntax, and a bunch of other techie stuff. Yeah, yeah, be still your beating heart, I know. But something kept nagging me, something I couldn't quite put my finger on, but it might have been *remotely* related to the fact that every time Mike C. sees me, he says: "Hey there, OCaml boy!" I just can't put my finger on it, but *something's* definitely nagging me lately.

So my articles kept meandering into self-defense territory, and it finally dawned on me: my problem is Mike! (Har, har.) No, actually my problem is that I've been writing all these late-night blog entries, usually after too many, er, beverages, and now everyone thinks I believe the company should switch to OCaml. Ouch.

So I'm gonna write about languages, and popularity. In particular, I think your choice of language should be based on cost optimization. Next month we can talk about techie stuff. Promise.

OCaml Considered Unpopular

Let's get this out of the way right up front. Here's a fictitious dialogue between an utterly fictitious person who is not intended to bear any resemblance to real persons, and myself:

Mike:	Should Amazon switch to OCaml, or use it in any capacity?
Me:	Um... no.
Mike:	What about all your blog-ranting about its time-machine debugger and all that stuff?
Me:	*(hopeful)* If I told you to jump off a bridge, would you do it?
Mike:	Um... no. But you *did* say it was the coolest language since (and I distinctly remember this) "Cobol".
Me:	Um, that might have been "Algol". But I did say OCaml was cool, yes.
Mike:	Well then! If it's so cool, why shouldn't we use it here?
Me:	'cuz it ain't popular.

But why *did* I write those blogs about OCaml?

Well, it's got some nice features, sure. And its runtime performance is amazing. But OCaml has strengths and weaknesses, as do all languages. If I were writing a native Windows/Win32 application, one that I had to compile into an executable to run on client desktops, then I'd seriously consider using OCaml. But I'm not doing that, and (presumably) neither are you.

OCaml's primary use to you and me is as a learning tool. High-level languages generally offer direct support for high-level abstractions, which just means you can express something complex without using very much code.

It can be a challenge to figure out what the heck some fancy new abstraction is actually doing, and why you'd ever need it. But once it sinks in, you'll start seeing places in your own code where you're doing... well, something *like* that abstraction. Except your version is all watered-down and mixed in with the rest of your code. It's there, but it's not as clear.

The same thing happened when Design Patterns came out, didn't it? That book just gave names to things you were probably doing already, except maybe you hadn't always encapsulated the idea as well as they did in the book. The book elevated certain useful programming idioms to the status of first-class citizenship.

New programming languages do *exactly* the same thing, except they usually offer syntactic support for these idioms, so it might look a little "weird" at first. That's the thing about syntax - it obscures the idea initially, because you have no clue as to what all the squiggly symbols mean, but after you learn them, it can make the code easier to read and write.

OK, I think we need one techie detour before we move on.

OCaml, SML and Haskell are all research-y pinhead type languages, but they're generally well-regarded. At least by pinheads. They all offer support for pattern-matching on data structures. OK... what's that?

To illustrate, let's consider the following snippet of Java code:

```
public void processPeople (List<Person> people)
{
  if (people == null || people.isEmpty()) {
    System.out.println("Sorry bub:  empty list.");
    return;
  }

  int length = people.length();
  if (length == 1) {
    handleOnePerson(people.get(0));
  }
  else if (length == 2) {
    handleTwoPeople(people.get(0), people.get(1));
  }
  else {
    System.out.println("Woah! We got " + length + " people!");
  }
}
```

(Sorry about the code being lit up like a bad Christmas tree. My syntax-highlighting program is a bit over-zealous.)

This isn't the greatest code in the world, but it gets the point across. Basically we receive a list of Person objects, and we need to do one of four things, depending on whether the list has 0, 1, 2, or more than 2 elements in it. It's a pretty common thing to want to do: we're examining a data structure and dispatching based on its contents.

OK, let's look at the same thing in Perl, which is usually less verbose than Java:

```
sub processPeople
{
  my @people = @_;
  my $len = scalar @people;
  if ($len == 0) {
    print "Sorry bub: empty list.\n";
  }
  elsif ($len == 1) {
    handleOnePerson(@people[0]);
  }
  elsif ($len == 2) {
    handleTwoPeople(@people[0], @people[1]);
  }
  else {
    print "Woah! We got $len people!\n";
  }
}
```

Nope, no help there. Looks basically the same. We're still getting the length of the list and testing it for 4 cases. There *are* a few differences, of course. For one, Perl appears to be great at interpolating variables into strings, but its built-in length() function doesn't work for lists. So it goes.

Note: Please don't email me saying that you wouldn't have coded it this way. I wouldn't have either. I'm keeping these examples simple and non-idiomatic to make them as readable as possible.

Let's try Ruby, which is a lot like Perl, but cleaner, more modern, and usually less verbose:

```
def processPeople people
  if people.nil? || people.empty?
    puts "Sorry bub: empty list."
    return
  end

  len = people.length
```

```
    if len == 1
      handleOnePerson people[0]
    elsif len == 2
      handleTwoPeople people[0], people[1]
    else
      puts "Woah! We got #{len} people!"
    end
  end
```

Nope! Even good ol' Ruby, a lovely Perl substitute if there ever was one, looks about the same in this example. Cleaner, maybe. But if Ruby's better than Perl, it's clearly not from doing this kind of thing.

Incidentally, I avoided using case- or switch-statements in all three snippets above, because in general, you can't use them for this kind of problem. We got lucky in this example because we're dispatching based on the length of the list, which is constant. But we could have been dispatching based on the structure and/or contents of the list, requiring us to dive into it and grub around. So in the general case, you'll use a cascading if-then-else.

Anyway, the Haskell/SML/OCaml designers realized that you do this kind of thing *so often* in your code that they made a special syntax for it. In OCaml (the most verbose of the three) it looks like this:

```
let processPeople people =
  match people with
    []         -> print_endline "Sorry bub: empty list."
    | [p1]     -> handleOnePerson p1
    | [p1; p2] -> handleTwoPeople p1 p2
    | _        -> print_endline "Woah! We got " ^ (length people) ^ " people!"
```

HMMMmmmm... this looks different. What's happening here? It looks a bit like a case or switch statement, but... not.

What's going on is just pattern-matching, a bit like you do on strings with regular expressions. You're drawing little ascii pictures of what you're looking for in each case, and inside the pictures, you're creating variable names to hold the pieces if there's a match.

Each "case" in the pattern matcher consists of a pattern, a little arrow-sign ("->"), and some code to run if that case matched. The code doesn't have to be on one line; it just worked out that way in this example. The cases are separated by "or" ("|") symbols. Patterns can be almost arbitrarily complex, and dive way deep into the data structure, even though they're simple in this example.

Our first case is an empty list: []. The next case tries to match a list with one item; if there's a match, it names it p1 and passes it to the code on the right. Similarly with the next case, but with 2 items. The last case uses a wildcard ("_") to say "match anything else."

Pretty nifty, eh?

To be fair, it's not such a huge win in this example. But as your data structures and class structures get more complex, and more deeply nested, this pattern-matching facility becomes *reeeeally* convenient.

Also, the pattern-matching syntax can be used in all sorts of places, not just in a match-statement. For instance, if you look at the three OCaml solutions (or the surprisingly compact Haskell solution) to the July Amazon Developers Journal "Farmer puzzle" challenge, you'll see pattern matching all over the place. In the Haskell one, the function for checking if a state is "bad" (i.e. someone gets eaten) uses pattern matching on the argument to the function:

```
-- check bad state:  chicken alone with dog or bag
bad [[_,fs],[_,ds],[_,cs],[_,gs]] = (cs /= fs) && (cs == ds || cs == gs)
```

Here, the bad function gets a list as its argument. But instead of naming the list, it specifies a pattern that it expects the list to match: a list of 2-item sublists. It ignores the first element of each sublist, and assigns the second element to fs (farmer side), ds (dog side), etc. The variables then become available for use in the body of the function, after the "=" sign, where we check if if the chicken's alone with the dog or the grain.

Generally, pattern matching allows you to specify something that looks like a literal declaration, but instead of building the data structure, it matches it against an existing one. You just can't *do* that in most languages, but looking at the farmer-puzzle solutions, it's clear than when you *can* do it, you use it all the time.

Meanwhile, back on Planet Earth...

OK, so how does this help us with our day-to-day C++/Java/Perl coding? After all, we're not about to use OCaml or Haskell for anything at Amazon; they're not popular enough, so we'd have a heck of a time hiring people, finding documentation, integrating with other languages and 3rd-party systems, etc. So what good is knowing about this feature?

Well, I'd argue that you've just learned yourself a new Design Pattern -- one that's usually just called pattern matching, but it's a Design Pattern nonetheless. And now, whenever you start learning a new language, you'll probably wonder whether they have support for this feature, and if so, how.

Some other languages do have it - for instance, Lisp has a function called destructuring-bind that does pretty much the same thing. And a new JVM scripting language called Groovy is offering something kinda similar called a "path expression". If you already know about pattern-matching syntax, picking up path expression minilanguages like Groovy's GPath becomes quite a bit easier.

For some abstractions -- maybe not this one, but for some -- you can find ways to encapsulate them as new classes or library routines in your favorite language. That's actually what the Design Patterns book was all about: expressing idioms in C++ that are often natively supported in other languages.

Perl's Switch.pm is a good example: it provides functionality similar to Ruby's built-in case expressions, with pattern matching on scalar values. This isn't anywhere near as powerful as data-structure pattern matching, but it captures some of the spirit of it.

Isn't learning new abstractions spiffy? I think so.

And *that*, folks, is why I study and write about programming languages. It's NOT to tell everyone to use OCaml at Amazon (Mike). Psh!

Handling that Pesky Cost Issue

Our "little" tech detour was a lot longer than I expected it to be. Oh well. My main point today is supposed to be about choosing languages with an eye towards optimizing cost, so let's talk about that a bit.

For the record, I agree 100% with the many folks who argue (sometimes coming awfully close to spitting on me in their passion about it) that using too many languages, or obscure languages, has a high cost. Yep. It sure does. Thank you for not spitting.

When discussing cost optimization, I tend to classify programming languages four broad categories:

1. **Production Languages** - languages that you use for building robust, scalable production systems. It's a pretty short list: C, C++, Java, and maybe C-sharp or Objective-C (but only maybe). They have to be popular, they should be statically typed, they need to perform well, they need great tools and documentation, etc. Not many options in this category.
2. **Scripting Languages** - languages you use for doing things that C, C++ and Java totally suck at, like string processing, OS automation, data munging, backfills, that kind of thing. Again, you want to use popular ones, so you're stuck with Perl, Ruby, Python, or the built-in Unix-y ones like shell-script and awk (although if you're gonna do that, you almost always want to use Perl/Python/Ruby because they have more expressive power and more libraries.)

   Yeah, I know, people who love scripting languages want to believe they're great for building giant scalable systems. I wish they were. They're certainly awesome for prototyping, and they seem to work well for user-interface creation (e.g. with Mason). But I'm a bit skeptical about building a large production system in a weakly-typed language, for lots of reasons: refactoring tool support, general code readability, performance, blah blah blah. I hope someday someone proves me wrong here, but for now, I *think* you want to use dynamically-typed languages only in certain niches.

   If you feel I'm wrong, please don't mail me with a long rant; publish your thoughts as a blog somewhere!
3. **Studyable Languages** - languages like Lisp, Scheme, OCaml, Haskell, Eiffel, Erlang, etc., that have cool features that you'd love to use, if only these languages were actually popular. But I wouldn't use any of them in production. (Lisp is a special case that I'm still trying to figure out, so let's leave it as a "maybe/sometimes" for now.)
4. **Languages you're stuck with** - e.g., JavaScript for client-side browser automation, Tcl for automating TotalView, Emacs-Lisp for Emacs, or any other language that you have to use because you depend on a product that requires it. There's no help for you here - you just have to know them if you want to be maximally productive with that product, and sometimes you don't have a choice.

So where does that leave us?

I think when you're deciding what language to use for "real" work, you want to choose as follows:

1. *Only consider popular languages.* Veeeeery important. Critical, even.
2. Pick the right language category for your problem. For systems up to a certain size, you probably want a good dynamic/scripting language, and beyond that point, use C++, Java or maybe C#.
3. Once you've decided the category, *always* use the highest-level language in the list. Use the one that's the most modern, expressive, and the least prone to errors when placed in the hands of busy/stressed programmers. If there's a tie, pick the one that's easiest to learn.

You shouldn't worry about language performance - that's a premature optimization. Remember that we're writing server-side software, and it doesn't need to run on our customers' desktops. Programmer time is very expensive, so pick a language that makes the best use of programmer time.

Your profiler will take care of the rest.

You shouldn't worry about whether you or your team know the language or not. That's being short-sighted. You may see more progress initially, but over time, you're losing out on the gains they'd get from having switched to a more productive language environment.

I realize all this flies in the face of the conventional wisdom that you should put performance first. Our industry is still largely slave to performance (and to running on the client desktop). C and C++ are still the only options for most companies, and performance is more important for them than "human" factors like code maintainability, build times, development speed, and so on.

But for Amazon and other web-based companies, the landscape has changed *significantly*. C++ and Perl may have been the only viable options in the late 90's, but that's simply not true anymore. And I think that unless we re-examine our options, and try to optimize for human cost instead of hardware cost, we're actually spending *more* money to do the same amount of work. A lot more.

I know there are plenty of smart people at Amazon who, at this point in the article, are ready to lynch me, but hey, this is just a magazine. They're welcome to write up their opinions too!

The Punch Line

I've spent a very long time trying to find the best solution to this multi-dimensional cost-optimization problem: about 2 years, in fact, part-time at home, studying and learning something like 40 programming languages. I care about this problem a *lot*, both for my own productivity and for Amazon's, since I plan to be here for a good long while.

I'm fairly convinced that for the forseeable future (i.e. the next 5 years or so), the right languages -- for me anyway -- are Java and Ruby. Java for big stuff, Ruby for small-to-medium stuff.

I can hear the cries of anger erupting already. My inbox overfloweth, and all that. Remember, I said *for me*. You can use whatever you like. My only recommendation is that you don't decide by fiat. Give it some thought; don't assume that the language you're most comfortable with today is automatically the right one for your next project.

And don't forget the "Languages You're Stuck With" category. I'd probably rather customize Emacs using Ruby than Emacs-Lisp, but elisp is my only choice for now. If you use VIM or Eclipse, you have more options. But you'll always need to know a few extra languages in order to get the most out of your tools. You can never get away with just two languages, not if you want to be efficient.

The punch line, then, is this: I'll use Java and Ruby when I can, and other languages when I have to. And I'll keep an eye on all of them, since you never know when some new one's gonna be totally... groovy.

Recommended Reading

Here are some nice resources if you found my rambling this month at all interesting:

- Paul Graham wrote a good article about the importance of language popularity. Actually, Paul writes lots of good articles. Check 'em out. He's a good writer, and he's rapidly turning into one of thought leaders of our industry.
- Here's a short and intriguing anonymous Slashdot article about language popularity: You Work in a Fashion Industry.
- A great book, hands-down, that talks about the value of learning languages, and about becoming a better software engineer in general, is The Pragmatic Programmer, by Dave Thomas and Andrew Hunt. Some folks in Dev Services are using it as book-club material right now, and liking it a lot. Good book!

Enjoy!

(Published Feb 2005)

Back to Stevey's Drunken Blog Rants™