On the Need of Minimalist Polyglots

Adrian Kosmaczewski

2008-05-12

Many companies, at some point of their history, ask themselves a simple question: what programming language should I use? The answer to this question is tricky, and has big, big consequences, for every single line of code of your future products will be written, read and suffered by it. This single choice defines the level of salaries you will have to pay, the skills of programmers you will have to deal with, the relative length and performance of your systems, the availability of tools (or lack thereof), the kind of support you will get (or not), the number of operating systems your code will work in, etc.

Given the fact that Web Development equals Software Development, this discussion will be of interest to those building the smallest websites, as well as old desktop-intensive apps. It will not be a "Tell me what programming language you use, and I will tell you who you are" type of article, though it may look like one, because that is something you have to figure out all by yourself.

If you take a look at the list of programming languages, any business person would have an instant headache. There are lots of them. With the strangest names. You cannot possibly guess which one to pick from such a list, obviously. So, how do companies choose the languages they use? There are some straightforward methods that I have seen so far, in no particular order:

- Looking at what other companies use (typically Google, Microsoft, Apple or Sun, but it could be 37signals too).
- Following the advice of the CIO, the Lead Architect or some other politically-powered person, which might or might not have read this article ;)
- Looking at what the current pool of programmers in the company know how to use. Rinse, wash, repeat.
- Following hype.
- Because there is a market plenty of available, cheap programmers that I could use for this project.
- Taking into account the characteristics of the languages themselves (static vs. dynamic, etc).
- Following the company's history of past projects (successful or not).
- Following what your the client suggests (or mandates).

I think that it is a very bad idea to take any of the above methods in isolation, without considering other factors. Doing so is a path to self-destruction in the medium to long term, even if you succeed in the short term.

Just as a small background for people not into programming: you can safely (roughly) group programming languages in a table like this:

	Static	Dynamic
Strongly typed	Java, Objective-C, Pascal, ...	Python, Ruby, ~~JavaScript~~, Objective-C, Lisp, ...
Weakly typed	C++, C, ...	JavaScript, VBScript, ...

In a static language all variable type references are bound at compile time; in a dynamic language, this is done at runtime (which allows you to assign a string or an int to the same variable). In a strongly-typed language, either the compiler or the runtime enforces the operations you can and cannot do on an object (depending on its type, as you may have guessed). In a weakly-typed one, there is no such restriction, and you can perform implicit conversions from one type to the other. And then you have functional ones, but that is another problem, because there are many programming paradigms out there. And then there is the hybrid ones, which I love as Steve Yegge does:

But also there's, like, the Boo language, the io language, there's the Scala language, you know, I mean there's Nice, and Pizza, have you guys heard about these ones? I mean there's a bunch of good languages out there, right? Some of them are really good dynamically typed languages. Some of them are, you know, strongly [statically] typed. And some are hybrids, which I personally really like.

He did not include Objective-C as "hybrid", but I think it is. Anyway, so much for the theory, here is the main point of this article:

First of all, I consider programming languages (I know a few of them, and I have my personal picks) just as tools to get things done™. Nothing else. I think of them as hammers or Black & Decker screwdrivers or saws or nail guns.

Second, I believe that specialization is for insects. Getting stuck in a single programming language because of any of the reasons I have enumerated above is just stupid.

So what I want to say is: You need polyglot programmers in your team, like you need a team of people knowledgeable in many human languages in every company doing business at global scale. I would say even more, you need not only people fluent in western languages (like English, French, Spanish and Italian, which is my combination) but also in other languages, with different paradigms behind, like Arab, Hebrew, Hindi or Chinese.

What does that mean in programming terms? You want to have programmers in your team being able to use different languages in different ways; you want to have programmers that learn new languages every so often, just for the sake of it. And most importantly, you want to get rid of programmers that not only get stuck on a single language, but that, even worse, dismiss any other way to do things. Having people that takes a negative look on the learning side of things can bring your whole company to a dead-end. This industry is plenty of integrists, and you do not want that in your company.

For example, JavaScript can be used as a procedural, object-oriented or functional programming language. Does your JavaScripters know how to write functional JavaScript? If not, that is too bad, because they will not be able to fully understand what is going on behind the scenes in Prototype or jQuery then. Of course this will not block them from using those libraries, but they might not understand how to apply some interesting patterns in their own code.

Ask more about your programmers: do they know how to do complex C++ template metaprogramming? Are they aware of some performance problems brought by garbage collectors? Do they follow the latest evolutions of the next version of JavaScript? (even if they do not like them) Which blogs do they follow? Do they know why some people hate PHP? Do they know what a continuation server is? Which programming books have they read lately?

And finally, what is even more important than having chosen a good programming language? Your methodology. Ask yourself (or your team) about these points:

- Do you write unit tests? You might not have testers, which is a bad idea anyway, but that does not block you from testing code yourself (Steve Yegge):

And I would say it's a pain in the butt, but I mean… it's a pain in the butt because… a static type-systems researcher will tell you that unit tests are a poor man's type system. The compiler ought to be able to predict these errors and tell you the errors, way in advance of you ever running the program.

- Do you use defensive programming techniques?
- Do you have a wiki, a project website or at least good documentation in your code? If not, how do your new hires learn about your product? No, reading the code is NOT a good answer.
- Is your chosen programming language portable? You might think that your system will never have to run on Linux or Mac, but why stuck yourself on purpose?

The common factor of all the above items is that you have to manage complexity. If you write code, you are creating complex stuff, and one of the best ways to manage complexity is to create small systems; as Steve Yegge said:

Small systems are not only easier to optimize, they're possible to optimize. And I mean globally optimize.

And this is why you do not only need polyglot, but also minimalist programmers in your team. Small is beautiful. Paul Graham (of Y Combinator fame) knows that dynamic languages yield small systems:

The right tools can help us avoid this danger. A good programming language should, like oil paint, make it easy to change your mind. Dynamic typing is a win here because you don't have to commit to specific data representations up front. But the key to flexibility, I think, is to make the language very abstract. The easiest program to change is one that's very short.

And how can you get small programs? By choosing the right programming language. Which brings us to the beginning of this post! So, here go some tips for all of you looking for the right programming language:

- Prefer strongly-typed, dynamic languages; there are a lot of reasons for that, particularly those exposed by Steve Yegge in his excellent presentation at Stanford:

Yeah, sure, it catches a few trivial errors, but what happens is, when you go from Java to JavaScript or Python, you switch into a different mode of programming, where you look a lot more carefully at your code. And I would argue that a compiler can actually get you into a mode where you just submit this batch job to your compiler, and it comes back and says "Oh, no, you forgot a semicolon", and you're like, "Yeah, yeah, yeah." And you're not even really thinking about it anymore.

Which, unfortunately, means you're not thinking very carefully about the algorithms either. I would argue that you actually craft better code as a dynamic language programmer in part because you're forced to. But it winds up being a good thing.

- Some points about his talk:
- Many of the caveats of dynamic languages are not (so) true anymore (like the lack of decent IDEs or the performance problems);
- There is a lot of research going on nowadays on the performance of programming languages, and this means that code written in these languages will benefit from many improvements in the near future, for free;
- And yes, there is the hype factor I have mentioned above; the cool kids are using them:

Which makes them exactly the kind of programmers companies should want to hire. Hence what, for lack of a better name, I'll call the Python paradox: if a company chooses to write its software in a comparatively esoteric language, they'll be able to hire better programmers, because they'll attract only those who cared enough to learn it. And for programmers the paradox is even more pronounced: the language to learn, if you want to get a good job, is a language that people don't learn merely to get a job.

- Teach yourselves; setup internal workshops to have all your team learn how to do cool stuff with those languages you have read about in DDJ.
- Teach the community around you, and do not be scared of competition; have your team write papers, articles on business or technical journals, publish code as open source projects, and show that you can go beyond.
- Keep an open mind: continuation servers, Comet or multicore processors are the future. Is your team prepared?

Software is a social process. Once you get this in your mind, and get your team working proactively, collaboratively and teaching each other, the choice of a programming language comes in second place.