Popular fallacies

Sheldon Linker
sheldon.linker@faculty.umgc.edu

Abstract: *I do consulting. I see a lot of want ads and a lot of projects in trouble. I recently had to go through a big list of want ads, and I can see why a lot of the projects are in trouble. Below, I explain these management fallacies, and at the end of the list, I tie it into this week's subject matter.*

Introduction

The following are a list of common fallacies, and why they are fallacies.

1. Agile is better than other development methods

False. Agile almost guarantees that some portions of the product will be re-done. It also guarantees that the designers, programmers, and testers will have to work for the entire project. Waterfall doesn't go back and revisit things, and doesn't require the whole team to work for the whole project. But, waterfall requires a very stable and well-known requirement set.

2. Solution-oriented architecture (SOA) is better than other architectures

False. If several companies are cooperating, and the system must be spread out, then SOA is the necessary way to go. For instance, let's say that a vendor uses a shipping company and a credit card company. The order comes in to the vendor's web-site, which then places a lien on the credit card using the credit card web site. The vendor site then goes to the warehousing/shipping site and orders the product shipped. If that site reports no problems, then the vendor site goes back to the credit card site and converts the lien to a charge, and then reports back to the customer that the transaction succeeded. That's SOA, and it works well. But, let's say that this is all going on within one system. It's much more efficient to have a monolithic shared data system. Consider the case of two people facing each other. There is a table on one side with 60 sheets of paper. When I blow the whistle, the first guy, with his right hand, is going to hand a piece of paper to the other guy, who's going to origami-fold it into something. When this second guy is almost done, the first guy will pick up a piece of paper with one hand. When the second guy is done folding, the first guy will take the paper from the second and at the same time, hand him the new paper. Then, the first guy places the folded origami on the second table. That's SOA. One transports, and the other folds. If transporting takes 1 second, and folding takes 1 minute, then the entire operation is going to take 1+60·1:00+1 = 1:00:02. But, if we don't use SOA, each guy has to pick up and drop off his own paper. That's going to take (1+1:00+1)·60÷2 = 31:00.

3. Efficiency of programming is no longer important

Maybe. Let's say that an efficient program runs 10% faster than a slower one. If 2% of the world's power goes to the cloud, and we increase the power required to the cloud by 10% through inefficient programming, then we're upping the world's energy usage by 2%·10% = 0.2%. Depending on what you believe, that wastes clients' time, global climate, fossil resources, and/or money.

4. Object-oriented programming is better than pure procedural programming

Maybe. But I have never seen a proof of this. I have, however, seen that pure procedural programs tend to be smaller than object oriented programs, by translating one to the other and then measuring size differences. Try to beat this APL example in any object-oriented language:

```
∇HELLOW̲ORLD
'HELLO WORLD!'∇
```

5. The inversion-of-control paradigm leads to better programs

How? Cheaper to create? Prove it. Faster to run? Prove it. Better results? Prove it. Easier or just as easy to automatically check? Prove it.

6. We should modernize our code

False. It's cheaper to teach the whole team Cobol or Fortran II (or some other out-of-fashion language) than it is to rewrite and retest the entire application every 10 years. I can get a modern z/OS system (present or in the cloud), and run S/360 code on it. On that, I can run a 1401 emulator. On that, I can run a 709 emulator, and run assembly code from the '50s. And it will run faster that way than it did in the '50s.

7. Everything should be moved to the cloud

False. There are speed and security issues. If your data never leaves the building, you don't have to worry about external security. If latency is important, then it may be important that the data not travel to and from the cloud.

8. If a single PC can't handle what you need, then you must have your system grow horizontally, such as using Map/Reduce for processing or "big data" concepts

False. You can also grow vertically, using "big iron" systems with zero extra programming effort. For processors, look at what IBM has. For databases, look at "federated" database systems, where big data happens for you, especially (in order) IBM, Oracle, Teradata, and Microsoft.

9. If I use a "rich stack", I'll get things done faster and cheaper

Maybe. Let's take a rich stack example, HTML, SASS, CSS, Angular, Spring, JQuery, JavaScript, JSP, JSP Tag Libraries, Java, Beans, Hibernate, Erwin, and SQL. Sure, SASS may make your CSS a bit easier. Angular and JQuery may make things a bit easier on the JavaScript side. Spring will add inversion of control (see above). JSP Tag Libraries will help to keep your JSP looking less like Java, but that's style over substance. JSP will keep your Java looking more like HTML. Hibernate and Beans will add layers of abstraction between Java and SQL but that's back to style over substance, at a speed cost. Erwin puts a GUI on data base design. So, that stack is going to run slower than a minimal "vanilla" stack, but when it comes to hiring a replacement, you've now got 14 things you need to see on that one person's resume. But, if you used a simple stack of HTML, CSS, JavaScript, Java, and SQL, that's just over a third as many things you'd need to look for in a resume, and 3 clear debugging paths.

10. Documentation should be in the form of wikis

Depends. If the documentation is encyclopedic in nature, in that people are going to use it by looking up a particular thing that they already know the name of, sure, wikis are about the best thing going. But, if someone is trying to learn something, then they'll never be able to traverse the wiki, and certainly won't be able to traverse it in any order that makes sense. If you're going to explain the system, rather than a minuscule component of it, have a manual or a primer with a "page 1" and a "the end". As requirements go, there also has to be a document where someone can say "this is it". When there are changes, those changes must be trackable, with change bars or a change list or the like.

Conclusion

Before adopting the latest trend, check it out. Will it help you? Will it hurt you? Unless the two answers are "Yes" and "No", respectively, don't implement it. If the answers are "Yes" and "Maybe", "Maybe" and "No", or "Maybe" and "Maybe", then it's likely worth finding out.