My Blog Essays

Agile's Dark Side

Being a self-proclaimed Agile Advocate I seem to find myself in discussions regard the bad points about agile. Books, articles, and talks on the subject of agile always paint the rosy happy story about using agile. I'm no fool, and I realize that things aren't quite as happy as some people make it out to be. No one said that agile was a silver bullet. The reason that I'm an advocate for it is because I believe it is simply a better way to write software. Let's get down to the meat of things. What is the "Dark" side of agile?

1)Increased Visibility can cause organization crisis. Agile, if nothing else, exposes all of the problems an organization has. If no one wants to take responsibility for these problems, then a crisis can (does)arise.

2)Lose of titles. What does it mean to be a "Senior QA Tester" on an agile team? Since everyone is responsible for getting the product done, titles/roles get blurred. Some people have definite problems with this. Especially when it comes time to do the yearly performance review.

3)People have to work together. Which, going back to item one, creates those pesky people issues. Normally you can avoid this on a traditional project by segregating opposed personalities. It isn't easy to do in an agile project.

4)Projects fail fast. This is a negative because in a traditional project, money from the budget usually keeps coming in because everything is "moving smoothly". But in an agile project more visibility may produce angry bean counters if the project goes directly into a dive. Again, there is no where to hide your dirty laundry on an agile type project.

5)Sustainable pace. We are used to slacking off at the beginning of a project and death marching the end. So most traditional projects see spikes in productivity. Agile projects push to establish a sustainable pace among the team. This means the developers are always busy and don't get a "break" from work. People are turned off by that (odd I know).

6)Feeling of micro-management. Agile promotes increased visibility. How do you get increased visibility? Your workers update their statuses in a daily manner. This can cause a definite feeling of micro management amongst the team.

7)Less up front design. Architects are perhaps the most difficult types of people to convince to use agile. This makes sense because these people are the ones who try and minimize risk from a technical aspect. Having a methodology that says, just do the minimum that is needed, to them is risky and goes against everything they have been trained on. I think it is important to remember that agile doesn't mean no design, just no unneeded design.

8)Flawed implementation of agile. If you rush, don't plan, are unsupported, and don't get team commitment you are likely to leave the organization with a mess on it's hands. The organization is likely to stay away from anything agile after getting such a bad taste in it's mouth.

9)Physical reorganization costs. A team is likely to want to remove the cubes and create a team room. Or at the very least reorganize their cubes in order to be closer to each other. There is a cost associated with doing this. Same is true for when the team demands newer hardware to do development. I'm all for new hardware, just realize that again, this is a cost that isn't obvious to an adoption of agile.

10)Loss of privacy. Well, no one said you had privacy at work anyway. But when you implement agile, this loss becomes a bit more apparent. Team rooms, pair programming, these things all attribute to some loss of privacy. Have to make a personal call? Before you may have sat in your cube. Now with agile you might have to go outside the office to have it be personal. Want to check your personal email? That is harder when your workstation is in the middle of the team room.

Change is hard, take time and do things the right way and make sure everyone understands what is happening. Remember that these problems listed above are not show-stoppers. With good leadership and good people they can all be moved out of the way. But I don't want you to go into this with your eyes closed. Know that there is a large possibility that you will see some of these dark sides. Be prepared and don't give up.

Benefits of Pair Programming

This post is in response to a blog entry at beyondng.com entitled "How much pair programming is cost justified".

Well, if we assume the perfect development environment with all top-notch developers, then yes, I agree Pair Programming probably has a low ROI.

For the rest of the world though, the following benefits to me show a ROI in short and long term. The benefits are:

1) Two pairs of eyes on the code. Yes, I understand that a pair of terrible coders will still produce terrible code. This also isn't an excuse to not do code reviews with the team. To me team code reviews should be coarse grain dealing more with design/functionality than syntax details. This is where the extra eyes on the code comes in handy.

2) Increases involvement. That is, as a single programmer, there are times where you are either way off task or just plain stumped. By having your pair with you it is more difficult to go surf to digg or slashdot. Thus your personal daily involvement with doing work increases.

3) Rapid learning. Pairing should allow a more open communication between people of different skill levels. Because of the increased involvement each person is more acceptable to learning new things. When the mind is focused and engaged learning is much quicker.

4) Higher actual time vs ideal time. A single coders actual coding time per day is not eight hours. It is more along the lines of 2-4 per day. This is due to the numerous other tasks that we handle along with whatever meetings we attend. By pairing up, each of the coders are committing to a chunk of time to work. I find it harder to interrupt two people working hard on a problem than a single person. Thus the actual time doing productive work goes up when pairing.

I'll finish up with stating that pairing is NOT for everyone. It isn't that they don't see the benefits; it is that they just don't like doing it. To me this is fine. In no way should it ever be forced upon a team. I would much rather a full team try it out and buy into it than have one person shove this technique down the team's throat. However....I also think that everyone should give it at least a couple of weeks when trying it out. One day or a couple hours isn't enough for a test run.

Career Paths for Coders

Something has been bothering me lately. You could say that it is another reason I'm Curmudgeon. Once a coder hits a certain point, say senior level, in his career he will plateau. There are only a few ways to continue moving up on the career ladder.

1. Start your own business

This isn't for the non-risk takers out there. Basically it is assumed that after five to ten years of experience in the field, you can start your own business. It can either be consulting out to other companies or maybe you have a great idea you've always wanted to make. Either way, you can continue on your career path by starting your own business.

Be warned though that you will need to know how to run a business and thus may still need that MBA. Which leads us to our next option.

2. Become a manager

I think that this is the most common path that senior level coders take. Either by going back to school and getting an MBA, or by simply being "promoted". Companies seem to think that the low level code is beneath the senior coders so we need to manage instead. There is a small problem here. That problem lies in the fact that great coders don't always make great managers. Heck, I doubt that they even make good managers. Yet companies still force us into this role with little to no training and then wonder why a great coder failed as a manager. Go figure.

3. Become a teacher

The third option for continuing with your career is to go back to the academic world and teach. You will still be able to do research on "cool" things, but your pay will most definitely drop. If you aren't coding for the money, then this may be an attractive option. Personally I find mentoring a very rewarding experience and given the opportunity would love to teach some classes. Another way to teach is by writing books and going on tour with a conference such as No Fluff Just Stuff. This might not be as fulfilling to you as teaching a class full time, but I would wager the money may be better.

These three options are all that I see out there for us to advance. Of course, we can always stay at our plateau and be happy coding. To me doing System Architecture or full time design work isn't at a higher level than senior developer. It is at the same vertical level, just a different horizontal level within the profession.

I know that our profession as a whole is very young, but are these really the only options that I have? If I want to be a professional coder for 30+ years what more can I look forward to? Maybe we need to rethink the current standard of what it means to be a "senior" developer. Many people that I interview seem to have really just one year of experience repeated five times.

This is what I propose. In order to have a year of experience it must be in doing something you have never done before. My definition of experience is "things learned from failure". Thus you need to do something new and fail in order to really have experience with it. So working ten years with the same company might only net you three years of "experience". This would push the salaries up for people who really are senior level, and perhaps make companies rethink forcing great coders into that management position.

What do you think? Do we have any other options besides these three?

Five Reasons I am a Curmudgeon Coder

These are my top five reasons for being the curmudgeon old coder that I am.

1. Unnecessary Meetings i.e. Sink Holes of Evil
   Enough said. And how to fix them? Try this

2. Coding is becoming simpler
   Yup, this is the "Back in my day.." reason, but with a twist. We are abstracting coding up higher and higher. Closing connections? Blah, don't worry, the framework will handle it. Memory allocation? Its okay, the garbage collection will handle it. Now don't get me wrong. I do appreciate the fact that because the abstractions are moving higher, I am left with more time to solve business problems. However, therein lies my problem. See, if I don't have to be an expert with bits and bytes and shifting, you know, that low level stuff; then that new grad from college doesn't need to worry about it either. And guess what? He's going to be cheaper than I am so who is going to get the position? I think we are abstracting ourselves out of a job. If ANYONE can program, then EVERYONE will.

   The second problem with going higher and higher with abstractions is that, well, sometimes they leak. If your abstraction leaks and something breaks, will you be prepared to fix the problem? For example, Hibernate is an abstraction for mapping a database table row to an object. Is there anyone out there who has NOT pulled their hair out because of a leaky Hibernate abstraction? Its okay to admit it, really it is. Millions of people admit it everyday on IRC, message boards and newsgroups. The abstraction leaked and now you are stuck with a problem that is over your head. So yeah, coding is becoming easier and we are creating abstraction zombies as the next generation of coders. Bleh.

3. XML
   Repeat after me, XML is not a programming language. Say it again a couple times. Feels good doesn't it? I have to admit that I was an XML groupie for a bit. XML was supposed to save us all. Every bit of data in the world was supposed to be wrapped in XML goodness and be shareable between businesses. It was a great happy plan. And then something terrible happened. Developers got a hold of XML. We ruined it, at least for me we did. The thing that ruined it...ANT. ANT and I have a love/hate relationship. I do really like ANT but sometimes, a build just needs to be scripted. You know, things like conditionals and loops? XML is NOT a very willing friend when it comes to scripting. ANT and I, we want a good relationship, really we do. But XML just keeps getting in the way. Oh, and while I'm discussing XML, why oh why does EVERY SINGLE PROPERTIES FILE NEED TO BE IN XML?!?!?! Sorry, I feel better now. Onto number four.

4. The coders aren't in charge anymore
   Since when did management become experts in how to create software? It used to be that the business would have a problem and want us to make software to solve their

problem. We would work with them and create the appropriate solution. Everyone was happy, we got to do things our way, and business got software to make their jobs easier. Somewhere along the line management felt they needed to step in. "You need to understand, business people can't understand you, and you don't know business". This was their excuse...I think it is BS. So now we have business people who no longer trust us to make what they want. We have management who dictates the who/why/when/where/how of us building the software. And we have been reduced to...well, code monkeys?

To those of you who are working in an environment where this is not the case. I applaud you and would just like to say that I am jealous of you. The best I can seem to manage are some guerrilla tactics followed by a campaign of "shock and awe". This is also the appropriate to mention that I believe Agile methodologies will become our answer to building trust again with the business. But until it becomes more mainstream we are going to continue to get beat up at the workplace.

5. Fixing the effect of a defect, not the cause of a defect

This last reason is becoming more of a trend and I think it relates up to reason number three. Our army of abstraction experts are developing a knack for applying band-aids to defects. Let's use Hibernate again for our example. Session closed errors are a big thorn in developers sides. What is the correct way to fix these problems? Well, there isn't a "one size fits all" fix. Instead, you might actually have to dig into the code. Message boards and newsgroups are a great source of information, but the debugger is your friend. Learn it, use it, become intimate with it. Please please please do not cut and paste solution X from a message board into your code base. Please? With sugar on top? Use the debugger, find the CAUSE of the problem and fix it.

How to Succeed at Outsourcing

Outsourcing is becoming a very popular trend at large and medium sized companies. Who can really blame them though? The thought of having the same work done for less cost is very attractive. Less cost means more profit, which means bigger dividends for the shareholders. At least...that is the idea.

I've been on three different projects with four different outsourcing companies. On each project at the end MANAGEMENT deemed the project a success while EMPLOYEES saw it as a failure. Management's main criteria for success was the line item balance sheet. If that showed a bigger revenue than loss, success! The employees viewed success in a different way. They looked at the quality of the product, because ultimately, THEY were responsible for it.

And thus this leads to the body of this entry, how to make it so both management and the employees are happy with outsourcing a coding project.

The first step in success is to tighten up every single feedback loop your company has. If you are having weekly meetings make them bi-weekly. If management only meets once a month, make it bi-monthly. If the feedback loops in the project become smaller and more efficient, any problems will show up sooner than later. This should in effect make the communication across all boundaries of the project become stronger. And in my opinion, communication is the number one thing to make or break an outsourced project. Another very important feedback loop to tighten up are the code reviews.

Which leads us to step number two. Make it clear between the on-site team and the offshore team where the final authority in code quality. Code reviews are a necessity for a successful outsourcing projects. The employees need to be involved and see the code that is going into the project which they will have to maintain. This is where the code review comes into play. It should be made clear that the employees have the final word on accepting code into the source control system, and that the code will be reviewed.

Ah yes, step three is have excellent source control management. Development is going to be happening 24 hours of the day. So when one team sleeps, the other codes, and visa versa. If your source code control system (you do have one right?) is not managed properly, it will cause some very unexpected problems. Make sure all your branching, versioning, merging procedures are in place and well documented.

Step four is...documentation. Remember step one when I talked about communication? One part of communication is the written part, or documentation. Everything needs to be documented BEFORE you start a project. Not during, not after, before. You are hiring a whole team, department, company to do your work. If you don't provide precise direction as to how you want things to work, then it won't be as you want it. The unfortunate truth is that an outsourced project needs MORE documentation than an in-house one. Teams that are overshore tend not to call you at 4am to ask simple questions and instead make a decision for themselves. Thus, document every single thing.

Step five, commitment to steps 1 - 4. Everyone in the company needs to be onboard with doing the outsourcing thing. If only one developer is doing code reviews, that person will not catch everything. Management must recognize the time needed to do documentation and be willing to let the employees create the necessary docs.

Step six find the right outsourcing company. There are many many outsourcing companies in this world. Your company needs to do the due diligence and select the right one. In the end, the people will make the difference. For example, one outsourcing company was being used as a jumping point for junior developers. They would hire a "developer" who had taken a single class in Java. Then put that person to work on our project. The on-site team would then basically have to train this person how to code by continually rejecting that person's work. Admittedly, the person would in the end become better, but then they would leave for a hiring paying company. Thus, starting the cycle all over again. If a bit of scrutiny was done before hand by my company, it would have saved us quite a lot of time.

Step seven your on-site team must be able to talk directly with the outsource team. Outsourcing companies seem to want to protect their developers. This will lead to questions being filtered through one person and a stone wall erected between the developer teams. Remember step one? The teams need to be able to talk and discuss issues that may arise. Tunnelling all communication through one person creates an obvious bottleneck. Make sure that this is agreed upon before starting and that the channels are fully open. Publishing an e-mail list is one excellent way to acheive this. If the outsourcing company refuses this..refer to step six.

I'll stop with these seven steps. With any change comes pain, and outsourcing for the first time is a huge change. I hope that with these seven guidelines that everyone in your company can honestly say that it was worth outsourcing that project.

How to Interview Java Coders Soft Skills

In the first part of this I discussed the technical interview. After the technical interview, we like to examine the candidate's "soft" skills.

Some of you might argue that since the candidate passed the technical part, there are no more questions needed. I disagree with this. In our profession we are not sitting in a room alone with no human contact. Daily we interact with our team, management, and the stakeholders. If our people skills are not up to par with our technical skills, then the performance as a whole will suffer.

One important distinction between the technical and the soft skill interview is that the interviewee drives the technical while the candidate drives the soft skill. So you should find yourself doing much more listening in the soft skill interview than in the technical. This is the time that you want the candidate to be talking, and hopefully making sense.

The question I start with for every soft skill interview is "Tell me about your current project and role in that project". Now sit back, relax, and let the candidate talk. Some things to look for:

1) Does the candidate feel comfortable talking to you?

2) Do you get a sense of excitement or ownership from the candidate?

3) Imagine the candidate talking about your project, is this how you would want it portrayed?

Once the candidate is finished describing the project the interview can go in many directions. Here are some ideas of where to go next. As you do more soft interviews, you will start to get a better feel of what to ask next.

* What was your biggest success during your career?

* Describe the hardest bug you have tracked down and how you fixed it.

* If you could change something on your current team, what would it be and why?

* Tell me about your ideal work environment

After the candidate is finished there are two things left to do. First is give the candidate an opportunity to ask questions to you. Try to answer these as truthfully as you can, lying to the candidate will come back at you if you hire the person.

The last thing to do is to give the candidate a sales pitch about your company. Developers talk amongst themselves. So take this time and sell your company to the person. Maybe you won't hire the candidate, but your chances to have your name mentioned has now increased.

How-to Interview Java Coders

Interviewing to me is definitely a trained skill that you have to practice with. Your goal is to extract as much information from the candidate so that you can make a useful recommendation. If you get too little information then your recommendation will contain more of a risk.

Each candidate that is interviewed is a different person and thus each interview is slightly different. For example: A candidate directly out of college will be asked different questions then a person with 10 years in the industry.

Regardless of the person, if they are wanting work as a professional programmer then they need to know a few of the basics. For the most part, it comes down to communication. When you interact with other programmers you have a basic vocabulary that is used. If the candidate doesn't have that vocabulary, then communication between the candidate and the other developers will be hindered.

I also am a strong believer that if you are going to work in the industry, you should at least have a basic understand of the underlying workings of a computer. Thus my questions are more general, yet still technical.

So this is my personal interviewing technique. One thing to note is that I try very hard to not ask trivia questions. These questions are things like "What does transient mean?". Guess what, I can look it up if I need it. I'm much more interested in if the candidate has a good solid base to grow with.

The first thing I hit on is what is a bit and a byte. From there I ask the primative types in Java and their sizes. The sizes question shouldn't be difficult if the candidate understands basic memory storage. Simply start with byte and work your way up, increasing the power of two each time.

Directly following this is a discussion about Data Structures. Remember your old friends (Map,Set,List,Array,Tree,Graph,Stack,Queue)? Hopefully you can name a couple of these, explain what makes each of them special, and have an example of when to use them. Oh, and a good side discussion regarding the equals() and hashcode() methods is good here as well.

Next is basic OO terminology. This is that vocabulary I discussed above. Polymorphism, encapsulation, interface, inheritance, overloading, overriding, pass by referrence, object, class, abstract class. If you don't know these then how can you a)communicate with your co-workers and b)create object-oriented software?

After OO I look at the resume and determine if they have database experience and if they are comfortable answering database questions. If yes then I would discuss primary keys, relationships, common database problems, and joins.

To round off the technical portion of the interview I discuss software designing. Things like coupling, cohesion, design patterns, tiered architecture. Usually I also go through a modelling question as well to see what types of behavior and attributes the candidate can come up with. The modelling question has been an item of hot debate with the team and we have yet to find a question that we are happy with.

My final technical interviewing requirement is asking the candidate to code. What? Code at an interview for a job that requires coding? That's crazy! The sad fact is that I have never been asked to code at any interview I've gone through. Would you hire a knife throwing juggler without seeing him or her juggle? NO! So why do we allow people to code without seeing them, well, code?

It doesn't take long to come up with some sort of coding exercise that can be written on the board or a piece of paper. Yikes! No IDE for help? Well, yeah, see the idea is that the exercise shouldn't really need an IDE. It should be simple enough to finish in 5 min, yet hard enough to get some sort of information from it. Currently the "Write a function to reverse a string" question has been working well.

If the candidate passes the technical portion of the interview, then it is onto Step 2, the soft skills interview.

The Heliocentric Software Revolution

As early as 4th Century BC the group known as the Pythagoreans thought that there was a fire in the middle of the sky and the Earth rotated around it. However, the Pythagoreans were considered crazy at the time, though Aristotle did give it some thought, and then dismissed it. In 1540 Nicolaus Copernicus again wrote down this idea. Galileo strengthened this with his observations and then Johannes Kepler came up with the math. So it took about 5000 years of observation, fact gathering, and mathematics but the MODEL that the Earth was the center of the universe did change. This was the Heliocentric Revolution.

If a model which is engrained in science and religion can change after 5000 years, then WHY can't a date on a project plan change? This is the central question for the Software Heliocentric Revolution (SHR).

A software plan starts with estimates and guesses. Not facts. These numbers and tasks are then written to a piece of paper or some computer program. The project gets started, real facts start trickling in. Bits and pieces of information gradually start forming a real model, one that is based off of facts. However, the project plan, the model, is not updated. It is as if all the facts are ignored and the Earth is still the center of the universe.

There are three basic items on the project plan; the number of resources, the tasks, and the estimate on how long a task will take. Given these three items a project manager is to create the model. The PM shapes this model, adds decorations here, color there, until it is a work of art. Perhaps this is why it is difficult to change? Coming up with this model is not an easy task. After putting that much effort into something, it is hard to admit that it isn't "right".

Maybe this is the solution? Stop making your project plans so darn pretty! Put less work into them because they are going to be thrown away and revised weekly if not daily. At the end of the project when the model is indeed correct, then add your decorations. Make it look just as good as the product. You will want it to look good when you hang it on your wall as another successful project.

So part one of the SHR is to turn your project plan agile. If the plan doesn't match the facts, then quickly and accurately revise that plan. The goal should be to keep the plan up-to-date at least weekly, if not daily. You can't have part one without part two so here it is.

As a developer I don't think I have ever been on a team that has wasted time. Indeed, most of the time we are happily working overtime on projects because we like it. Yet, it is common mantra to state "We are falling behind". What? What exactly is it that we are falling behind of? If the project plan is wrong, then how can we fall behind it? Isn't really what is happening is that the project plan is falling behind? So part two calls for a change in the mantra. No longer will we say things such as "I'm a week behind". Instead we will say "The plan is a week off" or "My estimates were a week off, please revise the plan".

Models are based upon facts. The developers need to get those facts to the PM so the model can always be right. Dates must move if the facts deem it. If the date just absolutely can't move then you have to get rid of some features. We can't continue to work in our current fashion. Viva la revolution!