

# Agile Isn't

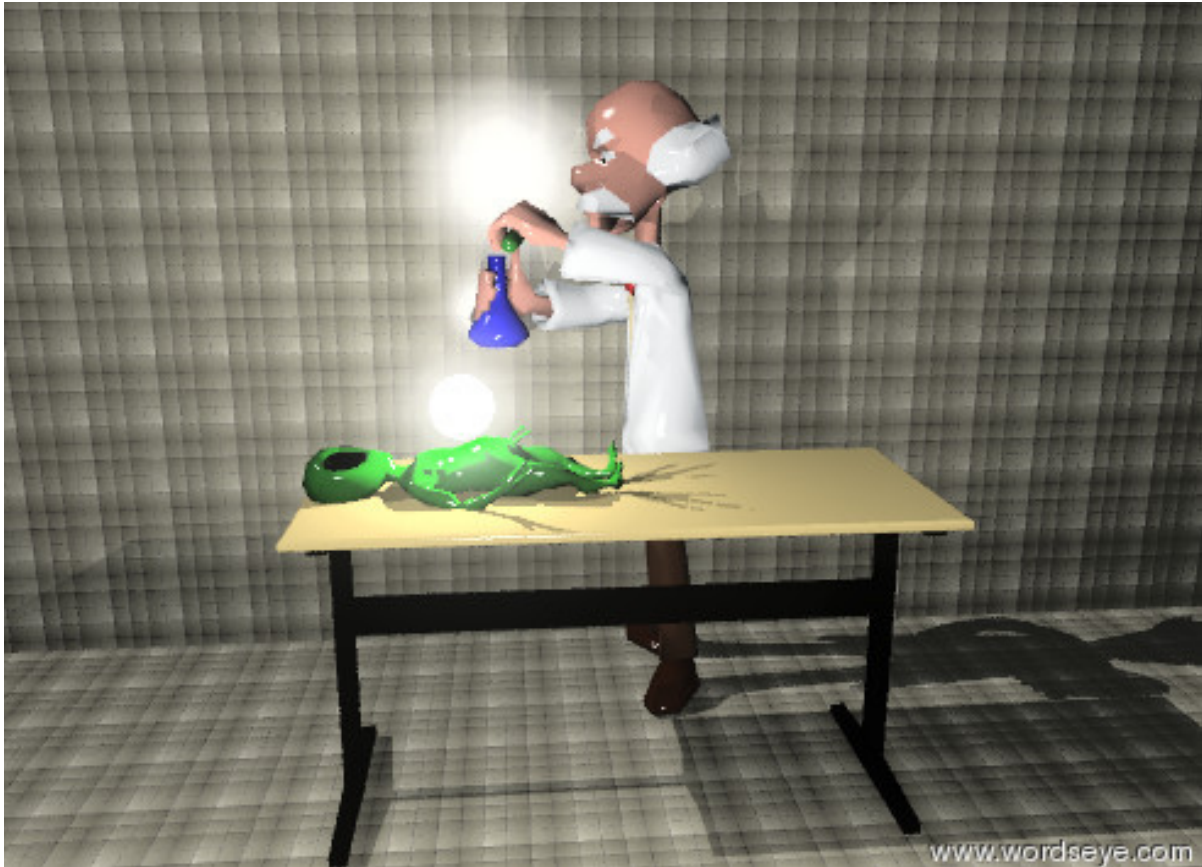
by Philip Chu

## Table of contents

1 Publication Information.....	2
2 What's Wrong with Agile.....	2
3 Learn from Open Source.....	5
4 Keep It Simple.....	7

## 1. Publication Information

Copyright ©2006-2008 by Philip Chu All rights reserved.



"The patient died, but the operation was a success."

## 2. What's Wrong with Agile

A director of engineering once characterized me as "in-the-closet" when it came to process. I do think there are important principles and procedures to follow in software development, but when I hear people brag about having a process, I shudder. Agile development is no exception.

I liked agile development...before it was called Agile. Now everyone says they're doing Agile, job postings list openings for "Agile/XP programmers", and Agile consultants and authors are everywhere.

## 2.1. It's a Religion

Because Agile has so many fervent believers, it's the worst thing to happen to software development since Microsoft. When optimizing programs, engineers are supposed to know that little improvements here and there don't matter if you don't fix the bottlenecks. But given clueless management and a project without direction, somehow Agile is supposed to save the day by saying, don't worry about it. That's not a process for creating quality products, it's process as a coping mechanism.

Normally, processes are pushed by management consultants on gullible management. But Agile is a religion for the masses - rank-and-file programmers think it's their salvation. They recite scripture (design patterns), participate in rituals (daily standup-meetings, unit tests, pair programming), Bible study (reading groups), and community worship (discussion lists and wikis). As a reward, they are assured by the high priests (consultants) they are Good Programmers.

Agile is bottom up. You don't have to know the Big Picture. In fact, you're better off not knowing. Just go month-to-month, week-to-week, write little snippets of code with unit tests, and keep asking "is this what you want?" Forget design. (Seriously, if you had a design, what was it?) Give up on long-range planning. It's the type of fatalistic approach that appeals to those who have been burned and those who don't know any better. You will be rewarded in the afterlife. (or with stock options)

## 2.2. It's Manufacturing

Agile also appeals to management by encouraging the misconception (or wishful thinking) that software development is a form of manufacturing.

- One large software project I worked on had an architectural identity problem - if you looked at any piece of the code, it was hard to discern the intent. Not because the code was poorly written. On the contrary, all of the thirty programmers on the project were highly qualified. The lack of design continuity arose from the practice of assigning the tasks pinned up on the corkboard to any available body. I was assigned several user interface bugs because the regular UI programmer was busy on other tasks. This had the pleasing effect of marking all of those bugs as "in-progress" or "fixed" in short order. But since the regular UI programmer had to inspect and approve my fixes before they went in, and he ended up rewriting them later, it really cost more time and effort in the long run.

There's a saying in video game design, "Find the fun". Squeezing software development into a manufacturing mold could be termed "Find the fun, then remove it". Software development

is not manufacturing, unless your idea of manufacturing is designing a new engine for every car that comes off an assembly line. Software is custom engineering and still resembles a craft more than anything else.

### 2.3. It's Process for Process Sake

Naturally, manufacturing companies would most like treat software development as manufacturing. They love processes and "best practices" - TQM, ISO 9000, Six Sigma. They also have the worst software development.

- I worked for a factory automation company that regularly flew our Director of Engineering out to meet their Software Best Practices committee. Yet we had a codebase that was not buildable for days at a time, misguided design, dysfunctional and hostile interaction among engineers, product requirements appearing immediately after product delivery, and components that were untested to the point of being obviously uninstallable (if anyone bothered to just look at it). And that was our cutting-edge Java application - typical start of the art in manufacturing was Visual Basic.

I often treat "process" as a dirty word, but process should be a good thing. Learning is a process. Self-improvement is a process. Doing things better every time around is a process. Process isn't something you trumpet - good people (in both the moral and competence senses) don't talk about how good they are, they just are. In fact, people who like to talk about how good they are usually aren't.

- One of the more entertaining workplaces I've seen included a programmer who evangelized Agile and a manager who pushed ISO 9000. The programmer covered his cubicle wall with scrum notecards and kept complaining we didn't have proper sprints yet didn't maintain a reproducible build process and ignored all feedback on the product from management. The manager wrote a process manual that didn't say much except that they had a process and gave his employees pop quizzes on the manual. Yet he didn't bat an eye when distributing release CD's filled with random builds of the product placed in random directories. Those guys made a perfect couple. In the meantime, there was plenty of work for service engineers.

Indeed, one of the tenets of Agile states "people before process". And defenders of Agile will immediately respond to any of these criticisms of negative anecdotes that "You're not doing it right." (This would be more credible if claims of Agile success were examined as skeptically) But theory doesn't matter if you have to suffer the practice. When people talk about process, what they really mean is a formula that allows them to avoid thinking about principles.

## 2.4. Striving for Mediocrity

The public schools system is great at educating the middle. If you're slower than the middle, you get pushed a bit or repeat a grade. If you're ahead of the curve, you just get bored.

Popular agile practices are also geared toward the average (or is it the median?) Scrum meetings by definition require everyone to move together (ostensibly like a rugby team moving up the field together - or like a soccer team of five-year-olds moving en-masse around the field while their parents yell encouragements). Pair programming is great for those who need to have their hand held or code monitored. Proving that one plus one is sometimes less than two.

- I worked for one company that boasted of its agile practices - in particular, scrum. Overall, it was a quality project with high production values. But I found the process more a hindrance than a help. The daily meetings just meant there was no planning - instead of working out task dependencies early, developers just waited until the morning scrum to say, "oh, by the way, I'm waiting on this..." The pair programming required for code checkin either meant someone was totally bored while I explained the code or I ended up quickly changing the code to match someone else's idea of good programming style just so I could check it in.

The real reason it was a quality project was the fact that they recruited talented people who were able to work with each other outside of the process enough to get the job done during the crunch that the whole process was supposed to avoid. All the process accomplished was to give the project managers the illusion that they were in control of things.

Coding is not inherently a team sport. Doubles tennis can be fun to play and entertaining to watch, but the game is meant for individuals. The Cohen brothers can write scripts together and finish each other's sentences while directing, but that is a rarity - most good work of that nature is performed by individuals. Your best programmers are the ones that can get the project done. But corporations feel more secure with all their engineers contributing mediocre work than with a handful of engineers contributing superior work and the rest killing time.

## 3. Learn from Open Source

Agile arguments tend to be anecdotal (like those of yours truly): "We used it on Project X and it worked." But if you want case studies, why not look at open source projects? One could argue that open source development is the most agile development of all.

### **3.1. Transparency**

More than any other type of project, open source projects are transparent. You can evaluate for yourself the quality of the code (just look at it), the popularity of the product (e.g. SourceForge ranking, or in the cases of projects like Apache or Firefox, it's well known), and the level of development (check the history of change, who changed what and when).

### **3.2. Communication**

With transparency and the tools to support it, communication follows. A typical open source project has active discussion lists (email and/or forum), a wiki for collaborative documentation, occasional chat sessions for real-time meetings, and occasional face-to-face get-togethers at conferences. Code changes are automatically announced on commit mail lists, giving everyone an opportunity to inspect the diffs.

### **3.3. Testing**

Also with transparency comes testing. Anyone who's willing to download and install is a tester. Again, tools are important. Users are more likely to submit quality bug reports (as opposed to just sending whining email) if the project web site sports a convenient bug entry page and searchable database. And users are more likely down to download the software, even free software, if the project has automated builds and tests.

### **3.4. Meritocracy**

Contrary perhaps to public perception, open source projects are not uncontrolled pools of code. (It's not Wikipedia!) Successful open source projects are led by programmers with a firm direction. The set of developers allowed to commit changes is typically limited to a few trusted people. Some projects have elaborate procedures for admitting new committers. Developers have to prove themselves before getting the keys to the vault. Bad code submissions are ignored - not incorporated into the codebase just so they can be fixed later.

### **3.5. No Interference**

With good tools for communication and good people doing the work, there's no need for constant supervision by professional managers. No need for frequent meetings. I have yet to see a notable open-source project with strict style guidelines. The development drives the process, not the other way around.

### **3.6. Competition**

What motivates open source developers to maintain effective practices? Competition. Even more so than with commercial software. Release buggy software, wait too long between releases, take the product in the wrong direction, anyone is free to take the code and spin off a new project with a new group of people (or even some of the same ones). Open source projects compete in a free and open market for users and developers. The agile movement, on the other hand, started in large corporate projects that can live or die based on reasons that have nothing to do with quality.

#### **4. Keep It Simple**

Good tools. Good people. Transparency. It's not that complicated.