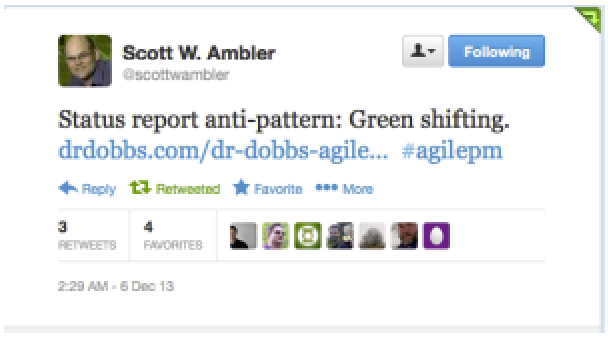
The “Green Shifting” Anti-Pattern

I saw this tweet from Scott Ambler earlier today:


Scott W. Ambler
@scottwambler

Status report anti-pattern: Green shifting. drdobbs.com/dr-dobbs-agile... #agilepm

The link in the tweet leads to the July 24, 2006 edition of Dr. Dobb’s Agile Journal, in which Scott writes:

> In astronomy there is the concept of red-shifting and blue-shifting: red shifting occurs when a body is moving away from you (the wavelength of the light increases) and blue-shifting occurs when a body is moving towards you (the wavelength of the light decreases). This is useful input for determining the speed and direction of something. In software development we have green shifting which occurs when people rework the information contained in status reports to make them more politically palatable to their manager.
>
> A few years ago I was involved with a seriously challenged project. In my weekly status report to the project manager I would very bluntly list all of the problems we were experiencing. Strangely, even though myself and several others were clearly documenting the problems, all of which were out of our control, nothing ever seemed to happen. After a few weeks of this I ran into the CIO and she congratulated me on how well the project was going. I was surprised, and told her that the project was in serious trouble and summarized the critical issues for her. It was news to her.
>
> After a bit of investigation we discovered that although myself and team members had been reporting the serious political challenges that the project team faced, when our project manager summarized our reports he decided to put a better spin on it and reported that the team found the project challenging. His manager in turn reworked it in his report that the team enjoyed the challenges it faced. This was the report which the CIO received.
>
> The problem is that the project status “green shifted” as it rose through the various levels of management. The people on the ground were very clearly reporting a red project status, our project manager wanted to play it safe so reported yellow status, and his manager was more political yet and reported green status.

This “green shift” isn’t just a problem in the software world. **Any** time somebody has to report upward to a more powerful person in the organization, there’s going to be some tendency to try and make any bad news less unpalatable. Filtered through multiple reporting levels, it’s all too easy for the bad news to be completely lost by the time the top person gets the report.

There are, fortunately, a number of ways for leaders to overcome this problem. Scott describes one simple one: the agile practice of a short daily standup meeting, where each team member provides a quick verbal status report. Anyone else interested in the project status is welcome to attend the meeting as an observer.

I saw a couple of other techniques while I was serving on a nuclear-powered submarine in the US Navy. Unsurprisingly enough, there’s a very clear hierarchy on a warship:

- the junior enlisted report to their division’s Chief Petty Officer
- the Chief reports to the division officer (usually an ensign or very junior lieutenant)
- the division officer reports to the department head (a senior lieutenant)
- the department head reports to the Executive Officer
- and the Executive Officer reports to the Captain.

Obviously, there’s a lot of room for the sort of “green-shifting” Scott mentioned to occur. One way the Navy counteracts this is with what is effectively a second, parallel hierarchy within the command. The division Chief Petty Officers report directly to their division officers, but they also communicate amongst themselves, and to the Chief of the Boat (the senior enlisted man on the submarine, typically a Master Chief Petty Officer). The Chief of the Boat has the direct ear of the Executive Officer and the Captain; this gives the skipper a second communication channel to cross-check the reports bubbling up from the junior officers. In addition, the Captain would spend part of every day at sea walking throughout the ship, seeing exactly what the conditions were without any filtering (deliberate or not) from the people reporting to him. That latter technique has crossed over into consulting lore with the label “Management by Wandering Around.”

The above techniques are clearly not specific to software development. One technique that **is** very software-specific is the agile practice of running short iterations, each of which delivers working code. Frequent releases are pretty binary; either you release working software every X days or you don’t. Suppose the person who’s signing the checks on a one-year project insists on seeing working software every two weeks. If the project hasn’t delivered anything new for six weeks, the person signing the checks **knows** something’s wrong, regardless of any green-shifting that might be occurring in the status reports.

Alas, all of these techniques depend on the leader devoting time and energy to supplementing status reports with hard data. If that leader chooses to restrict himself to status reports, he runs the serious risk of green-shifting hiding bad news for months or even years. The consequences can be calamitous; just ask anybody who’s been involved in a multi-year project that gets cancelled after burning through millions of dollars and thousands of hours of people’s lives.

A couple of days ago, Robert “Uncle Bob” Martin asked Where is the Foreman?:

> The foreman on a construction site is the guy who is responsible for making sure all the workers do things right. He’s the guy with the tape-measure that goes around making sure all the walls are placed properly. He’s the guy who examines all the struts, joists, and beams to make sure they are installed correctly, and don’t have any significant defects. He’s the guy who counts the screws in the flooring to make sure it won’t squeak when you walk on it. He’s

> the guy — the guy who takes responsibility — the guy who makes sure everything is done *right*.
>
> Where is the foreman on our software projects?

Many moons ago, way before I started making a living writing software, I spent several years in the US Navy as an enlisted man specializing in operating submarine nuclear reactor plants. Freshly out of the training pipeline, I reported aboard my first submarine, which happened to have just started an overhaul at the Pearl Harbor Naval Shipyard. The shipyard had confidently scheduled eleven months for the work, even though the standard for that particular type of overhaul was closer to eighteen months. By the time I reported aboard, the boat was already in drydock and the heavy construction work was well underway. The vast majority of the work was being done, not by the ship’s crew, but by civilian shipyard workers, supervised by shipyard management – including, of course, plenty of foremen. We of the ship’s crew **were** heavily involved in the quality-assurance role (since we’d be the ones living with the results). In particular, whenever a reactor plant piping system (seawater, fresh water, reactor coolant, etc.) was modified, the resulting work had to be hydrostatically tested and we had to sign off on the test results. These tests generally involved:

1. isolating the modified piping from the rest of the system
2. connecting a test pump and one or more high-accuracy pressure gauges to the sytemm
3. filling the piping with water
4. pumping enough additional water into the piping to raise the pressure well **above** the maximum pressure normally expected in that system
5. shutting off the test pump
6. and watching the pressure gauges

To pass the test, the system had to hold test pressure for a specific period of time. So, the very first time I was involved in one of these hydrostatic tests, we had all the valves lined up properly, the system was filled with water, and one of the shipyard workers started the pump … and the pressure didn’t go up.

Ooops.

They ran the pump some more, and the pressure still didn’t go up.

Ooops.

It turns out that one of the welded pipe joints hadn’t actually been welded fully. The presence of those above-mentioned shipyard foremen didn’t prevent one particular welder messing up. Neither did any of those foremen spot the problem prior to hydrostatic testing.

Eventually, the leak was found, and fixed, and successfully tested. Eventually, in fact, we got **all** the systems working again. Eventually, we were able to take our newly-overhauled submarine out to sea and start performing the sorts of missions we were being paid to accomplish.

“Eventually”, however, ended up being **twenty-six months** after entering the yards. Not the eleven months originally scheduled. Not the eighteen months that the Navy considered “normal” for that sort of overhaul. **Twenty-six** months, with far too many of those months involving twelve-hour shifts seven days a week trying to “catch up” to an unrealistic schedule slipping into the abyss. (Not that that would sound familiar to anyone in the software development world. </sarcasm>)

And all that happened **despite** having all those foremen.

So would having foremen in software development solve our problems, as Uncle Bob implies?

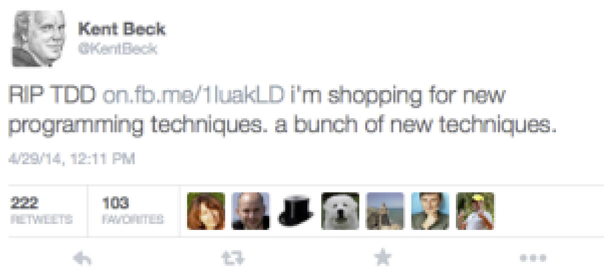
> If you want to get a project done, done right, and done on time, you need a foreman. And that foreman has to be so technically astute that he can check the work of all the workers. He has to have the authority to reject any work he considers sub-standard. And he also has to have the power to say “No” to the unreasonable demands of the customers and managers.

My answer: nope.

It’d probably be very helpful; don’t get me wrong. In fact, I’d love knowing that a second set of eyes would look at my work, providing a backstop in case I missed something, or giving me glowing feedback about the stuff I got right. I’d love having people on my team with both the skills and the bandwidth to do that sort of review quickly enough that the review process didn’t bottleneck the project as a whole. I’d love knowing that the organization I’m working for values high-quality work enough to assign that sort of person to my projects, not to grind out code, but to keep the standards high.

But having foremen is **not** sufficient to guarantee project success. I found that out back in the yards, many moons ago.

It started, as so many things do, with [a tweet](#) (this one from Kent Beck):



Kent Beck
@KentBeck

RIP TDD on.fb.me/1luakLD i'm shopping for new programming techniques. a bunch of new techniques.

4/29/14, 12:11 PM

222 RETWEETS 103 FAVORITES

[The link in that tweet points to an entry on Kent’s Facebook page](#), which in turn references a blog entry by someone named David Heinemeier Hannson, titled [TDD is dead. Long live testing.](#):

> Test-first fundamentalism is like abstinence-only sex ed: An unrealistic, ineffective morality campaign for self-loathing and shaming.

Kent’s response:

> I’m sad, not because I rescued [TDD] from the scrapheap of history in the first place, but because now I need to hire new techniques to help me solve many of my problems during programming

Kent then lists a number of things that he gets from Test Driven Development. I paraphrase many of those items here because they’re many of the things that *I* get from practicing TDD:

- **Avoiding over-engineering**: By not implementing any new features until I have a need for them (as evidenced by a newly written failing unit test), I keep myself from coding things that I currently *think* I’ll need in the future, but that end up never being used.
- **Rapid API feedback**: Creating a new Application Programming Interface (API) one test at a time helps me keep the API usable. After all, if I can’t code a test that exercises a method, how should I expect anyone else to use that method in their code?
- **Rapid detection of logic errors**: I’m human. I make mistakes. I even make mistakes while writing code. With TDD, the automated unit tests catch those mistakes quickly and cheaply.
- **Documentation**: While I don’t automatically get a beautifully written, lavishly illustrated users manual from applying TDD, I *do* get a nice suite of automated unit tests, each of which shows the folks who will be working with this code in the future exactly how I intended the implemented code to be used.
- **Avoiding feeling overwhelmed**: Every once in a while I get stuck trying to figure out how to implement something. However, assuming the tool support exists, I can pretty much always figure out how to write a test to prove that something is or is not implemented. Writing that test lets me make some progress in the face of what may feel overwhelming, and often gives me that missing bit of insight needed to implement the feature.

One thing Kent did *not* mention, but that I’ve personally found hugely valuable, is the way I can use TDD (and the resulting suite of automated unit tests) to uncover problems in the specified *requirements*. Suppose one is writing a program to assign accounting codes to medical claims depending on such factors as the type of provider, type of medical service provided, the patient’s insurance type, the date of the claim, etc. Suppose further that one is given the rule “all alcoholism-related claims are to be assigned code 90116”. One writes a test to see that an alcoholism-related claim is assigned code 90116; then one runs the test suite and finds that the test fails (because no one has implemented that logic yet). One implements the logic and reruns the test suite. The new test passes, but an older test stating that all claims for members of ASO plans are to be assigned code 31000 fails. One reviews the requirements and discovers that, in fact, they state that:

1. all claims must be assigned to one and only one code
2. all ASO claims must be assigned to 31000
3. all alcoholism claims must be assigned to 90116.

Clearly, there’s something wrong in the requirements, although at this point one cannot say for certainty what that something is. But, with TDD, I’ve discovered these sort of problems within moments of trying to implement something. With a test-last approach, I won’t find these sorts of problems out until after I’ve already implemented the inconsistent requirement. If I’m lucky, I’ll find them out before the code gets handed off to someone else for testing; if not, I’ll probably not find them until the fixes go live and screw up the general ledger accounts.

TL;DR? I’m still doing TDD. Because reasons.