-from https://michaelochurch.wordpress.com/2015/06/06/why-agile-and-especially-scrum-are-terrible/

JUNE 6, 2015 BY MICHAELOCHURCH

Why “Agile” and especially Scrum are terrible

Agility is a good thing, no doubt, and the Agile Manifesto isn’t unreasonable. Compared to a straw-man practice called “Waterfall”, Agile is notably superior. Yet, so much of Agile as-practiced is deeply harmful, and I don’t really think that the Agile/Waterfall dichotomy is useful in the first place.

There’s a variety of Agile, called Scrum, that I’ve seen actually kill a company. By “kill”, I don’t mean “the culture wasn’t as good afterward”. Rather, I mean that its stock dropped by almost 90 percent in less than two years.

What is Agile?

Agile grew up in web consulting, where it had a certain amount of value: when dealing with finicky clients who don’t know what they want, one typically has to choose between one of two options. The first is to manage the client: get expectations set, charge appropriately for rework, and maintain a relationship of equality rather than submission. The second is to accept client misbehavior (as, say, many graphic designers must) and orient one’s work flow around client-side dysfunction.

Programmers tend not to be great at managing clients. We’re very literal people. We like systems that have precisely defined behaviors. This makes working with non-technical people harder for us than it needs to be, because we tend to take every request literally rather than trying to figure out what they actually want. Corporate Agile, removed from the consulting environment, goes further and assumes that the engineers aren’t smart enough to figure out what their internal “customers” want. This means that the work gets atomized into “user stories” and “iterations” that often strip a sense of accomplishment from the work, as well as any hope of setting a long-term vision for where things are going.

In general, people tend to create two types of jobs, whether inside a company or as clients when off-loading work. At the high end, they hire for expertise that they don’t have. At the low end, they dump all the stuff they don’t want to do. It’s probably obvious that one class of consultant gets respect and the other doesn’t. Mismanaged consulting firms often end up becoming garbage disposals for the low end work. Scrum seems to be tailored toward the body shops, where client relationships are so mismanaged that the engineers have to be watched on a daily basis, because they’ve become a dumping ground for career-incoherent work that no one wants to do (and that probably isn’t very important, hence the low rate and respect).

Violent transparency

There are a lot of workplace trends that are making the programming career extremely unattractive, especially to the sorts of creative, intelligent people that it’s going to need in order to fill the next generation.

Open-plan offices are the most egregious example. They aren’t productive. It’s hard to concentrate in them. They’re anti-intellectual, insofar as people become afraid to be caught reading books (or just thinking) on the job. When you force people to play a side game of appearing productive, in addition to their job duties, they become less productive. Open-plan

offices aren't even about productivity in the first place. It's about corporate image. VC-backed startups that needed to manage up into investors used these plans to make their workspaces look busy. (To put it bluntly, an open-plan programmer is more valued as office furniture than for the code she writes.) For a variety of cultural reasons, that made the open-plan outfit seem "cool" and "youthy" and now it's infesting the corporate world in general.

It's well known that creative people lose their creativity if asked to explain themselves while they are working. It's the same with software. Programmers often have to work in an environment of one-sided transparency. These Agile systems, so often misapplied, demand that they provide humiliating visibility into their time and work, despite a lack of reciprocity. Instead of working on actual, long-term projects that a person could get excited about, they're relegated to working on atomized, feature-level "user stories" and often disallowed to work on improvements that can't be related to short-term, immediate business needs (often delivered from on-high). This misguided but common variant of Agile eliminates the concept of ownership and treats programmers as interchangeable, commoditized components.

Scrum is the worst, with its silliness around two-week "iterations". It induces needless anxiety about microfluctuations in one's own productivity. There's absolutely no evidence that any of this snake oil actually makes things get done quicker or better in the long run. It just makes people nervous. There are many in business who think that this is a good thing because they'll "work faster". I've been in software for ten years, as a manager and a worker bee. It isn't true.

Specific flaws of "Agile" and Scrum

1. Business-driven engineering.

Agile is often sold in comparison to "Waterfall". What is Waterfall?

Waterfall is legitimately terrible. It's a "work rolls downhill" model in which each tier of the organizational hierarchy picks off what it consider to be "the fun stuff", and passes the rest down the line. Projects are defined by executives, design is done by architects, personal deadlines are set by middle managers, implementation is performed by the top-tier grunts (programmers) and then operations and testing are handed off to the lower tiers of grunts. It's extremely dysfunctional. When people have a sense that all of the important decisions have been made, they're not motivated to do their best work.

Waterfall replicates the social model of a dysfunctional organization with a defined hierarchy. Agile, quite often, replicates the social model of a dysfunctional organization without a well-defined hierarchy.

I have mixed opinions about job titles like "Senior" and "Principal" and the like. They matter, and I probably wouldn't take a job below the Principal/Director-equivalent level, but I hate that they matter. If you look at Scrum, it's designed to disentitle the senior, most capable engineers who have to adhere to processes (work only on backlog items, spend 5-10 hours per week in status meetings) designed for people who just started writing code last month.

Like a failed communist state that equalizes by spreading poverty, Scrum in its purest form puts all of engineering at the same low level: not a clearly spelled-out one, but clearly below all the business people who are given full authority to decide what gets worked on.

Agile, as often implemented, increases the feedback frequency while giving engineers no real power. That's a losing bargain, because it means that they're more likely to be jerked around or punished when things take longer than they "seem" they should take. It creates a lot of anxiety and haste, but there's little of what actually matters: building excellent products.

Silicon Valley has gotten a lot wrong, especially in the past five years, but one of the things that it got right is the concept of the engineer-driven company. It's not always the best for engineers to drive the entire company, but when engineers run engineering and set priorities, everyone wins: engineers are happier with the work they're assigned (or, better yet, self-assigning) and the business is getting a much higher quality of engineering.

If your firm is destined to be business-driven, that's fine. Don't hire full-time engineers, though, if you want talent. You can get the best people as consultants (starting in the $200-per-hour range, and going up steeply from there) but not as full-timers, in an business-driven company. Good engineers want to work in engineer-driven firms where they will be calling shots regarding what gets worked on, without having to justify themselves to "scrum masters" and "product owners" and layers of non-technical management.

Ultimately, Agile (as practiced) and Waterfall both are forms of business-driven engineering, and that's why neither is any good at producing quality software or happy employees.

2. Terminal juniority

Software engineering, unfortunately, has developed a culture of terminal juniority, lending support to the (extremely misguided) conception of programming as a "young man's game", even though almost none of the best engineers are young, and quite a few of the best are not men.

The problem with Agile's two-week iterations (or "sprints") and user stories is that there is no exit strategy. There's no "We won't have to do this once we achieve [X]" clause. It's designed to be there forever: the business-driven engineering and status meetings will never go away. Architecture and R&D and product development aren't part of the programmer's job, because those things don't fit into atomized "user stories" or two-week sprints.

As a result, the sorts of projects that programmers want to take on, once they master the basics of the craft, are often ignored, because it's annoying to justify them in terms of short-term business value. Technical excellence matters, but it's painful to try to sell people on this fact if they want, badly, not to be convinced of it.

I once worked at a company where a product manager said that the difference between a senior engineer and a junior engineer was the ability to provide accurate estimates. Um, no. That's insulting, actually. I hate estimates, because they generate politics and don't actually make the work get done faster or better (in fact, it's usually the opposite).

The worst thing about estimates is that they push a company in the direction of doing work that's estimable. This causes programmers to favor the low-yield, easy stuff that the business doesn't actually need (even if bumbling middle managers think otherwise) but that is "safe". Anything that's actually worth doing has a non-zero chance of failure and too many unknown unknowns for estimates to be useful. Agile's focus on short-term business value ends up pushing people away from the kinds of work that the company actually needs, in favor of each programmer's real-time reputation management.

What this culture says to me is that programming is a childish thing to be put away by age 35. I don't agree with that mentality. In fact, I think it's harmful; I'm in my early 30s and I feel like I'm just starting to be good at programming. Chasing out our elders, just because they're seasoned enough to know that this "Agile"/Scrum process has nothing to do with computer science and that it adds no value, is a horrible idea.

3. It's stupidly, dangerously short-term.

Agile is designed for and by consulting firms that are marginal. That is, it's for firms that don't have the credibility that would enable them to negotiate with clients as equals, and that are facing tight deadlines while each client project is an existential risk. It's for "scrappy" underdogs. Now, here's the problem: Scrum is often deployed in large companies and funded startups, but people join those companies (leaving financial upside on the table, for the employer to collect) because they don't want to be underdogs. They want safety. That's why they're in those jobs, rather than starting their own companies. No one wants to play from behind unless there's considerable personal upside in doing so. "Agile" in a corporate job means pain and risk without reward.

When each client project represents existential or severe reputational risk, Agile might be the way to go, because a focus on short-term iterations is useful when the company is under threat and there might not be a long term. Aggressive project management makes sense in an emergency. It doesn't make sense as a permanent arrangement; at least, not for high-talent programmers who have less stressful and more enjoyable options.

Under Agile, technical debt piles up and is not addressed because the business people calling the shots will not see a problem until it's far too late or, at least, too expensive to fix it. Moreover, individual engineers are rewarded or punished solely based on the optics of the current two-week "sprint", meaning that no one looks out five "sprints" ahead. Agile is just one mindless, near-sighted "sprint" after another: no progress, no improvement, just ticket after ticket after ticket.

People who are content to shuffle mindless through their careers can work that way, but all of the really good engineers hate it when they go to work on a Monday morning and don't know what they'll be working on that day.

4. It has no regard for career coherency.

Atomized user stories aren't good for engineers' careers. By age 30, you're expected to be able to show that you can work at the whole-project level, and that you're at least ready to go beyond such a level into infrastructure, architecture, research, or leadership.

In an emergency, whether it's a consultancy striving to appease an important client or a corporate "war room", career coherency can wait. Few people will refuse to do a couple weeks of unpleasant or otherwise career-incoherent work if it's genuinely important to the company. The work's importance creates the career benefit. If you can say "I was in the War Room and had 20 minutes per day with the CEO", that excuses having done grunt work.

When there's no emergency, however, programmers expect their career growth to be taken seriously and will leave if it's not. Saying, "I was on a Scrum team" says, "Kick me". It's one thing to do grunt work because the CEO recognized that an unpleasant task would add millions of dollars of value (and that he'd reward it accordingly) but doing grunt work just because you were assigned "user stories" and Jira tickets shows that you were seen as a loser.

5. Its purpose is to identify low performers, but it has an unacceptably high false positive rate.

Scrum is sold as a process for "removing impediments", which is a nice way of saying "spotting slackers". The problem with it is that it creates more underperformers than it roots out. It's a surveillance state that requires individual engineers to provide fine-grained visibility into their work and rate of productivity.

There's a fallacy in civil liberties debates: the "nothing to hide" argument. Some people like to argue that invasions of privacy– by, say, law enforcement– are non-issues because they're not themselves criminals. It misses a few key things. For one thing, laws change. Ask anyone who was Jewish in Central Europe in the 1930s. Even for respectable people, a surveillance state is an anxiety state.

The fact of being observed changes the way people work. In creative fields, it's for the worse. It's almost impossible to work creatively if one has to justify the work on a day-by-day basis.

Another topic coming to mind here is status sensitivity. Programmers love to make-believe that they've transcended a few million years of primate evolution related to social status, but the fact is: social status matters, and you're not "political" if you acknowledge the fact. Older people, women, racial minorities, and people with disabilities tend to be status sensitive in the workplace, because it's a matter of survival for them. Constant surveillance into one's work indicates a lack of trust and low social status, and the most status-sensitive people (even if they're the best workers) are the first ones to decline when surveillance ramps up. If they feel like they aren't trusted (and what else is communicated by a culture that expects every item of work to be justified?) then they lose motivation quickly.

Agile and especially Scrum exploit the nothing-to-hide fallacy. Unless you're a "low performer" (witch hunt, anyone?) you shouldn't mind a daily status meeting, right? The only people who would object to justifying their work in terms of short-term business value are the "slackers" who want to steal from the company, correct? Well, no. Obviously not.

The violent transparency culture is designed for the most status-insensitive people: young, usually white or Asian, privileged, never-disabled males who haven't been tested, challenged, or burned yet at work. It's for people who think that HR and management are a waste of time and that people should just "suck it up" when demeaned or insulted.

Often, it's the best employees who fall the hardest when Agile/Scrum is introduced, because R&D is effectively eliminated, and the obsession with short-term "iterations" or sprints means that there's no room to try something that might actually fail.

The truth about underperformers is that you don't need "Agile" to find out who they are. People know who they are. The reason some teams get loaded down with disengaged, incompetent, or toxic people is that no one does anything about them. That's a people-level management problem, not a workflow-level process problem.

6. The Whisky Goggles Effect.

There seems to be some evidence that aggressive management can nudge the marginally incompetent into being marginally employable. I call this the Whisky Goggles Effect: it turns the 3s and 4s into 5s, but it makes you so sloppy that the 7s and 9s want nothing to do with you. Unable to get their creative juices flowing under a system where everything has to be justified in terms of short-term business value, the best programmers leave.

From the point of view of a manager unaware of how software works, this might seem like an acceptable trade: a few "prima donna" 7+ leave under the Brave New Scrum, while the 3s and 4s become just-acceptable 5s. The problem is that the difference between a "7" programmer and a "5" programmer is substantially larger than that between a "5" and a "3". If you lose your best people and your leaders (who may not be in leadership roles on the org-chart) then the slight upgrade of the incompetents, for whom Scrum is designed, does no good.

Scrum and Agile play into what I call the Status Profit Bias. Essentially, many people in business judge their success or failure not in objective terms, but based on the status differential achieved. Let's say that the market value of a "3" level programmer is $50,000 per year and, for a "5" programmer, it's $80,000. (In reality, programmer salaries are all over the map: I know 3's making over $200,000 and I know 7's under $70,000, but let's ignore that.) Convincing a "5" programmer to take a "3"-level salary (in exchange for startup equity!) is marked, psychologically, not as $30,000 in profit (which is tiny) but as a 2-point profit.

Agile/Scrum and the age discrimination culture in general are about getting the most impressive status profits, rather than actual economic profits that would be attained, usually, by hiring the people who'll deliver the most value (even if you go 50% above market rate, because top programmers are worth 10-30 times their market rate).

The people who are least informed about what social status they "should" have are the young. You'll find a 22-year-old 6 who thinks that he's a 3 and who will submit to Scrum, but the 50-year-old 9 is likely to know that she's a 9 and might begrudgingly take 8.5-level conditions but she's not about to drop to a 3, or even a 6. Seeking status profits is, of course, useless. These points don't mean anything. You win in business by bringing in more money than you pay out in costs, not by exerting margins in meaningless status points. There may be a whole industry in bringing in 5-level engineers and treating (and paying) them like 4's, but under current market conditions, it's far more profitable to hire an 8 and treat him like an 8.

7. It's dishonestly sold.

To cover this point, I'm going to focus on Scrum. Some people argue that Scrum is "extremist Agile" and others say that it's the least Agile variant of Agile. (This, perhaps, underlies one of the problems here; we can hardly agree on what is and is not "Agile".)

Something like Scrum has its place: a very limited, temporary one. The dishonest salesmanship is in the indication that it works everywhere, and as a permanent arrangement.

<u>What Scrum is good for</u>

Before the Agile fad made it a permanent process, "Scrum" was a term sometimes used for what corporations might also call a "Code Red" or a "War Room emergency". If there was an unexpected, rapidly-evolving problem, you'd call together your best people and form a cross-cutting team.

This team would have no clear manager, so you'd elect a leader (like a "Scrum Master") and give him authority. You'd want him not to be an official "people manager" (since he needs to be as impartial as possible). Since the crisis is short-term, individuals' career goals can be put on hold. It's considered a "sprint" because people are expected to work as fast as they can to solve the problem, and because they'll be allowed to go back to their regular work once it's over. Also, in this kind of emergency, you need to make it clear that no one is being evaluated individually. The focus is on the mission and the mission only.

When you impose process and demand one-sided transparency of all the workers, people feel watched. They get the sense that they're being stalked and will be labelled "low performers" as soon as their week-by-week, individual fluctuations go the wrong way. So they become resentful. That's dysfunctional. On the other hand, when you call together a group of known high-performers to handle a specific and time-limited crisis, they don't mind giving frequent status updates so long as they know that it's the exigency of the situation, rather than their own low social status, that is the reason behind it.

There are two scenarios that should come to mind. The first is a corporate "war room", and if specific individuals (excluding high executives) are spending more than about 4 weeks per year in war-room mode, than something is wrong with the company, because emergencies ought to be rare. The second is that of a consultancy struggling to establish itself, or one that is bad at managing clients and establishing an independent reputation, and must therefore operate in a state of permanent emergency.

Two issues

Scrum, in this way, acknowledges the idea that emergency powers must sometimes be given to "take-charge" leaders who'll do whatever they consider necessary to get a job done, leaving debate to happen later. That's fine. It's how things should be, sometimes.

A time-honored problem with emergency powers is that they often don't go away. In many circumstances, those empowered by an emergency see fit to prolong it. This is, most certainly, a problem with management. Dysfunction and emergency require more managerial effort than a well-run company in peace time. The result is that there are many managers, especially in technology, who seek out opportunities for emergencies and emergency powers.

It is also more impressive for an aspiring demagogue (a "scrum master"?) to be a visible "dragonslayer" than to avoid attracting dragons to the village in the first place. The problem with Scrum's aggressive insistence on business-driven engineering is that it makes a virtue ("customer collaboration") out of attracting, and slaying, dragons (known as "requirements") when it might be more prudent not to lure them out of their caves in the first place.

"Agile" and Scrum glorify emergency. That's the first problem with them. They're a reinvention of what the video game industry calls "crunch time". It's not sustainable to work that way. An aggressive focus on individual performance, a lack of concern for people's career objectives in work allocation, and a mandate that people work only on the stated top priority, all of these provide a lot of value in a short-term emergency, but become toxic and demoralizing in the long run. People will tolerate short-term losses of autonomy, but only if there's a clear point ahead when they'll get their autonomy back.

The second issue is that these practices are sold dishonestly. There's a whole cottage industry that has grown up around teaching companies to be "Agile" in their software development. The problem is that most of the core ideas aren't new. The terminology is fresh, but the concepts are mostly outdated, failed "scientific management" (which was far from scientific, and didn't work). Again, these processes sometimes work toward hitting well-defined targets in emergency circumstances, but permanent Scrum is a disaster.

If the downsides and upsides of "Agile" and Scrum were addressed, then I wouldn't have such a strong problem with the concepts. If a company has a team of only junior developers and needs to deliver features fast, it should consider using these techniques for a short time, with the plan to remove them as its people grow and timeframes become more forgiving. However, if Scrum were sold for what it is– a set of emergency measures that can't be used to permanently improve productivity– then there would be far fewer buyers for it, and the "Agile" consulting industry wouldn't exist. Only through a dishonest representation of these techniques (glorified "crunch time", packaged as permanent fixes) are they made salable to the corporate world ("the enterprise") at large.

Looking forward

It's time for this culture of terminal juniority, low autonomy, and aggressive management to die. These aren't just bad ideas. They're more dangerous than that, because there's a generation of

software engineers who are absorbing them without knowing any better. There are far too many young programmers being doomed to mediocrity by the idea that business-driven engineering and “user stories” are how things have always been done. This ought to be prevented; the future integrity of our industry may rely on it. “Agile”, at least as bastardized in every implementation that I’ve seen, is a bucket of nonsense that has nothing to do with programming and certainly nothing to do with computer science. Business-driven engineering, in general, is a dead end. It ought to be tossed back into the muck from which it came.
Big Data is dead. Collecting a lot of data is literally useless, if the data is not properly utilized. The key is the systematic exploration of the data with a right set of questions. For instance, is the data uniform or irregular? Is there a significant amount of variation in the data set? Is it buried in a mass of other irrelevant information? Can it be easily extracted and transformed? Is it possible to load the data at a reasonable speed? Can it be thoroughly analyzed? Can powerful insights be garnered? Otherwise, Big Data alone in an old style is really obsolete.