Scrum, Complexity, and Process Improvement

Everyone likes Scrum. What's not to like? Scrum increases productivity, improves return on investment, delivers useful functionality every month, and helps everyone enjoy working. Yet, everyone wants to tinker with it, to improve it, to increase its accuracy, to make it more amenable to his or her culture.

I used to help people modify Scrum to make it more compatible with their perceptions. Lately, I've come to realize that this is a mistake. For sustained improvements with Scrum, to stop yourself from making changes that undercut the core of Scrum, you have to understand Scrum at its deepest, theoretical level. And this is very hard. But only then will you understand why these improvements will destroy the very success you desire. How did this paradox come to be?

The scientific approach is the basis of much of our current understanding of our world. Based on observations, we make a hypothesis. We test the hypothesis on a statistically significant set of examples. If sustained, the hypothesis can be used in every day life to build technology and to predictably guide our interaction with the world. This is called the deterministic approach and is exemplified by Newtonian physics.

Our approach to science has changed over the last century. Something called quantum mechanics has started to show us that there is more afoot than the deterministic approach would lead us to believe. You see, there is a lot of glossing over the rough edges, accepting "out of boundary" experimental results, and approximation to use the results of the deterministic approach. We've accepted this imprecision because the results of applying these hypotheses are acceptable, more or less. Imprecision like the ability to predict the interaction of planetary bodies using Newtonian physics. We can predict fine as long as we restrict the predictions to two planetary bodies, but predictions fall apart when we try to predict the interactions of three or more bodies. The imprecision, the approximation, shows up then through unpredictable variations between further observation and predictions.

A body of thought called complexity theory is useful in thinking about this problem. It says that complex things act differently than simple things. Simple things operate predictably according to our laws as measured by the available precision of our measuring instruments. Complex things sometimes act predictably, but just as often will act wildly unpredictable, sometimes fluctuating in a manner that cannot be modeled mathematically, sometimes not, sometimes settling into a mathematically predictable pattern after a time, sometimes not. Worse yet, the more precise our measuring instrument, the more we are finding that even things we thought were simple really are complex; we had just never been able to detect this complexity and we had found ways of working around any apparent unpredictability with approximations.

So, what does this have to do with Scrum. Scrum is an empirical process control rooted in industrial process control theory, and applicable to complex problems that otherwise are not amenable to solution. I have applied Scrum to hundreds of projects over the last decade, most of them software development projects. I have and continue to assert that Scrum with its basis in empirical process control theory, is the correct theoretical approach to software development. Why? Over this time not a single Scrum project has failed, at a time when industry-wide failure rates were over 60% and of the other 40% that succeeded, 70% of the functionality delivered wasn't useful when delivered.

Scrum deals with complexity, not simplicity. Scrum calls for frequent inspection and subsequent adaptation during a project. Scrum practices queue up things for inspection and adaptation frequently enough and with enough precision that software projects and other projects that deal with complex problems are able to thread their way through the unpredictable nature of complexity to deliver something of value. Scrum constantly sticks the results of people attempting to turn imprecise, changing requirements and truculent, treacherous technology in your face and asks you to figure out if its resolution is proceeding acceptably. If not, Scrum asks you to devise an adaptation on the spot that redirects efforts to maximally improve the likelihood of an acceptable outcome.

But we are trained in the scientific method. We pride ourselves in simplifying things, in reducing them to a state where they can run unattended or at least with less attention and more precision. Scrum and empiricism are the art of the possible, but people want the art of the predictable – even though complex things aren't predictable by their very nature. This sticks in the craw of people that want deterministic management techniques and the scientific method to reduce the practice of software development into a predictable practice and discipline.

Estimating is the nexus of this failure to understand. In Scrum, estimates of work are only a starting point, a way of getting our minds around a complex problem. Teams start to work on the problem, dealing with the complexity, the unexpected and the unpredictable as they proceed. At the end of iteration, a thirty calendar day time-box called a Sprint, the team demonstrates what it has been able to do. Based on the team's success and progress, everyone then figures out what is the most valuable thing to do next. This continues Sprint by Sprint until the problem has been resolved and the complexity rendered into a system that is satisfactory.

To someone of the scientific method bent, however, this is inadequate. They want to improve the accuracy of these estimates so they will be able to predict better what will be ready at the end of each Sprint. One approach that they've used in the past is comparing estimated work to actual work required. Their belief is that if these are adequately studied, a hypothesis for the variance can be derived, the hypothesis can be tested on more real work, and then the entire process can be improved by applying the hypothesis in practice. This belief exposes the vast chasm between solving problems through a deterministic or the quantum approach to the world. Quantum mechanics and one of its children, Scrum, indicate the fallacy of this approach. The problem is complex and not amenable to reductionism. The degree of complexity and unpredictability inherent in the problem negates attempts to statistically manage the results to increase predictability. Attempts to tinker with the process to increase predictability only lessen its effectiveness.

I tell people that Scrum is really hard work, and they think that they know what I mean. I tell them that the software development problem is so complex that the only solution is to constantly pay attention through inspection and to have to over and over again derive the best possible adaptation to anything that the inspection reveals to be unexpected and out of tolerance. This cannot be shirked or avoided when dealing with complex problems. But, over and over, people look for ways to simplify Scrum so that they can pay less attention, so that the process will require less of their attention and intelligence, so that they can go on to solving other problems. That would be nice but for two things: if you don't pay adequate attention to complex problems, attempts to solve them will fail, and there are no deterministic solutions that will allow anyone to reduce this attention. Worse yet, as Scrum proves itself over and over in solving complex problems, people will increase the complexity of the problems that are being addressed.

Is Scrum easy? In presentation, yet; in practice, no! Scrum not only requires all of our attention and intelligence, but it runs contrary to our ingrained, deeply held beliefs about how to solve problems. Our lives are so full of problems and complexity that it is a normal reaction to want to reduce them. Unfortunately, the consequence of either inattention or reductionism in the software development process is failure.