An Experiment: The GROWS™ Method

Andy Hunt is a programmer turned consultant, author and publisher. He has authored award-winning and best-selling books, including the seminal The Pragmatic Programmer and eight others, including his latest, Learn to Program with Minecraft Plugins, the popular Pragmatic Thinking and Learning: Refactor Your Wetware and the Jolt-worthy Practices of An Agile Developer.

Andy was one of the 17 founders of the Agile Alliance and authors of the Agile Manfesto; and co-founded the Pragmatic Bookshelf, publishing award-winning and critically acclaimed books for software developers.

2015-08-22

ScrumMaster.dk

- "My Reading List"

Our Story So Far...

Back in February of 2001, just after the dawn of the new century, I was fortunate enough to be invited to hangout with sixteen other concerned citizens of the software development world to talk about what was then known as "light weight processes."

At the time, software development was struggling between unreliable, ad-hoc (often pronounced "odd hack"), development with no process at all, and unreliable, stultifying, over-specified draconian processes that tried the naive approach of "plan, then execute." Developers and corporations realized that neither of these approaches were working.

But there was hope: perhaps a very light-weight process of some sort might work. There were a few interesting ideas coming out about then: eXtreme Programming and Scrum among others were proposing simple and effective ways of working together that didn't succumb to the naive fallacy of executing a predetermined plan, and yet would save us from the wild, undisciplined ravages of random hacking.

Now we all had our own ideas of what constituted *simple*, and surely disagreed on what was more effective, but we all did agree on enough basic points to collectively author The Agile Manifesto as a way to express our ideas and concerns.

On the one hand, the Agile Manifesto succeeded wildly: it provided a jolt of energy; a realistic hope of better ways of doing things. Maybe we, as a community, really could advance the field. Certainly in the 14 years since then, topics such as unit testing, test-driven development, continuous integration and iterative development have become widely known, and even adopted. We actually do talk to customers, sometimes. So we really *have* made progress over the old days.

But despite these advancements, "agile" failed.

The Failure of Agile

The basis of an agile approach is to embrace change: to be aware of changes to the product under development, the needs and wishes of the users, the environment, the competition, the market, the technology; all of these can be volatile fountains of change.

To embrace the flood of changes, agile methods advise us to "inspect and adapt." That is, to figure out what's changed and adapt to it by changing our methods, refactoring our code, collaborating more closely with our customers, and so on.

Agile methods have specific, defined practices that give us the opportunity to identify problems and adapt ourselves. But practices only give us the opportunity; you can lead a horse to water, but you can't force it to change.

And that causes the first part of our collective problem: doing the agile practices only gives you the opportunity to *be* agile, it does not make one agile by itself. The popularity of agile practices does not guarantee the popularity of agility.

This leads us to the current state of affairs where agile practices are popular and more-or-less widely adopted, but practiced without understanding what it means to *be* agile—to reliably figure out what's changed and deliberately adapt to it.

So the word "agile" has become sloganized; meaningless at best, or with a vehement, warlike, jingoist enthusiasm at worst. We have large swaths of people doing "flacid agile," a half-hearted attempt at following a few select software development practices, poorly. We have scads of vocal agile *zealots*—as per the definition that a zealot is one who redoubles their effort after they've forgotten their aim.

We've forgotten that the aim is to adapt. And we are entirely to blame for that, because our popular agile methods themselves have not been agile. The irony is epic with this one.

The Irony of Agility

Our popular agile methods themselves are not agile. That is, they are not generally set up to adapt *the method itself* to change.

Let that sink in for a moment. The core principle of agility is to inspect and adapt as needed in a highly collaborative environment. We are slowly getting the idea that we can adapt and change code over time, but we aren't doing that with our methods of working together. Instead we've gone quite the other way, and tend not to tolerate *any* innovation in our methods.

Now to be fair, I think this is due in part to teams and organizations backsliding into familiar, old-fashioned, toxic territory of heavy process and naive plan-then-execute schemes. To prevent re-adopting these failed ideas, we've accidentally spawned a whole population of zealots who insist on a "canonical" approach to agile.

As a result, what should have been a vibrant, exploding community of individual, finely-tailored agile methods well adapted to local conditions becomes instead a battle for universal conformity: "you're not agile unless you do exactly these 12 practices as defined in THE BOOK."

That is not agile.

Let's be perfectly clear on this point: if you and/or your organization are following the XP practices, or the Scrum practices, then you are not agile. You are not adapting to your team, your code, your market, or the needs of your users. Agility—and success in today's markets—demands a flexible, adaptive approach.

Sounds great on paper. But just saying, "Adapt, damn ye!" while being perhaps cathartic, is not actually useful. You can't just command people to adapt—not even yourself. That's too much of an abstract concept to start off with.

When you are first learning a new skill—a new programming language, or a new technique, or a new development method—you do not yet have the experience, mental models, or ability to handle an abstract concept such as "inspect and adapt." Those abilities are only present in practitioners with much more experience, at higher skill levels. Let's look at that idea in a little more detail.

The Dreyfus Model of Skill Acquisition

The Dreyfus Model is a helpful way of looking at how people acquire and grow their skill sets. It defines five stages that you experience as you gain abilities at some specific skill (for more on this topic, see my article in *PragPub* magazine, *Why Johnny Can't be Agile*, or my book, *Pragmatic Thinking and Learning*).

Here are the stages, with some key features of each:

- Novice - Rules-based, just wants to accomplish the immediate goal
- Advanced Beginner - Needs small, frequent rewards; big picture is confusing
- Competent - Can develop conceptual models, can troubleshoot
- Proficient - Driven to seek larger conceptual model, can self-correct

- Expert - Intuition-based, always seeking better methods

Research suggests that most people, at most individual skills, never get beyond the Advanced Beginner level. At the Advanced Beginner stage, you're comfortable enough to sort of muddle your way through. And for most folks, that's enough. It kind of looks good on the outside, and you can list it on your resume. But that's a hollow victory. In terms of software productivity, you might as well be coding in COBOL with a pencil on a yellow legal pad.

But here's the real catch: the kind of self-reflection and self-correction that an agile approach requires **is not possible** at the lower skill stages. That is an ability you simply do not have access to until you reach the Proficient stage at a skill, which is an advanced stage that the majority of practitioners will not reach.

Showing folks agile practices and asking them to "reflect and adapt as needed" isn't enough. In order to succeed at software development in the 21st century, we need to take a much more directive and pro-active approach.

The Beguiling Trap of Rules

In order for Beginners to be effective, they must to follow simple, context-free rules; rules of the form: "When *this* happens, do *that*." Beginners, by definition, simply don't have the experience to apply any judgment or perform any problem solving in the domain.

And since agile methods conveniently provide some concrete practices to start with, new teams latch on to those, or part of those, and get stuck there. So instead of looking up to the agile principles and the abstract ideas of the agile manifesto, folks get as far as the perceived iron rules of a set of practices, and no further.

Agile methods ask practitioners to *think*, and frankly, that's a hard sell. It is far more comfortable to simply follow what rules are given and claim you're "doing it by the book." It's easy, it's safe from ridicule or recrimination; you won't get fired for it. While we might publicly decry the narrow confines of a set of rules, there is safety and comfort there. But of course, to be agile—or effective—isn't about comfort (see *Uncomfortable with Agile*).

What's worse, if you only pick a handful of rules that you feel like following, and ignore the hard ones, then you've got it made! You are now firmly in deeply dysfunctional territory. You are "doing agile" if anyone asks, and you can focus your energy on enforcing a small set of largely useless rules. Everyone feels better. Until it's time to actually ship something. Until users try and use it. And now you've become a statistic.

Although rules are required for beginners, they also trap you there. According to the Dreyfus research, following a rigid set of rules limits your performance to that of a novice. So teams find themselves unknowingly stuck in a rut: blindly following the concrete rules, not gaining the experience they need to get to the point where they know they need to move beyond the rules.

It's all too common these days to see arguments on Twitter or mailing lists with these rules-bound zealots arguing that "you're not agile" because you aren't following the rules to their satisfaction.

What happened to the idea of inspect and adapt? What happened to the idea of introducing new practices, of evolving our practices to suit the challenges at hand? The canonical agile practices of popular methods have remained essentially unchanged for over a decade. We are stuck in a rut.

And as I'm fond of saying, the only difference between a rut and a grave is the dimensions.

A Better Way

To recap, here's what we've learned so far:

1. People tend to trust the false authority of a tool or method because they don't feel they can trust their own experiences.
2. People are afraid to veer from prescribed practices because they don't know what aspects are safe to modify, and which aren't.
3. People aren't fond of change, but really hate to *be changed*—to have change imposed.
4. People won't necessarily adopt *all* the practices that they need to in order to be successful.

5. People will adopt popular, concrete practices, but won't automatically seek deeper understanding of their applicability or rationale.

Now these are wide generalizations of course. Lots of teams and organizations have gotten over these hurdles and been very successful with their modern software development techniques. But there are many, many more who have not. We need a better way to help them.

Let's start off with our goal in mind:

- To inspect and adapt as needed in a highly collaborative environment.

While being mindful of our constraints:

- The Dreyfus Model skill stages suggest we need rules for beginners, followed by conceptual models and self-reflection later on.
- Psychology suggests that people more readily accept change when they are part of it, when they can see personal gain, and when it comes in small doses.
- Experience with existing methods suggests that we need to be more inclusive, and not aim solely at developers, or solely at team leads/managers, or solely at executives.
- The authority of a name-brand tool or method (no matter how spurious) makes ideas more acceptable

Ok, the last part is the easiest: we need a name: The GROWS™ Method. Why GROWS? Software is not designed and built; that's far too much of a deterministic, linear model that doesn't work here. Growing is a better metaphor, because with growth comes change. Often organic, chaotic, non-deterministic change. That's the real world.

Given that our main goal is to inspect and adapt, we need a method where you can base all your decisions and direction on actual, local evidence: feedback from the real world, under actual conditions. Anything else is just some unfortunate combination of a fantasy and wishful thinking. And while working software is still our ultimate deliverable, it can't be at the expense of a working, functional *team* and overall organization. The software, the team, the users, the sponsors all form **one** system. And we need to make sure the whole system works, for all the participants.

So let's base our "new" method on four foundational ideas:

1. Evidence-based Experiments
2. Dreyfus Skill Model
3. Local Adaptation to Context
4. All-Inclusive

We'll use concrete, well-define practices to help instill better habits in ourselves in our teams. Specifically, we want to get better at deliberately creating an environment where we can get rapid, real-world feedback.

Every team should be able to inspect and adapt based on this feedback, for all activities. But to get them there, we need to use the lessons of the Dreyfus Skill Model to provide support for beginners and latitude for more advanced practitioners. With that framework in place, we can position every team to adapt themselves to local conditions, safely, based on actual evidence. And finally, whatever we attempt here has to work for the *whole system*, not just the developers, not just the managers, not just the testers, not just the users, not just the sponsors. There is no "us" versus "them." There is only us: we're all in one boat and we'll float or sink together.

We'll drive the whole thing by promoting the idea of an *experiment* to a first-class part of the method. We'll use experiments for adoption, to gain experience, to grow the team and to grow the deliverable, shipping software itself.

It's Just an Experiment

It's not strictly true that people don't want change. What folks really want is new and different *results*. Preferably for free, without having to change any significant behavior or ingrained habits.

For any chance of success at all, change needs to come from one's own desire. It's like the old joke where the lightbulb has to *want* to change. That means that people need to be shown a very personal, direct upside for *them*. Abstract notions such as "higher quality code" or "improve our time

to market" just aren't compelling. As a developer, I probably don't care about these issues; any benefit to myself is indirect at best. As a customer support rep, I may have much more of an interest in code quality, because that affects me. As a VP of sales, time to market may be critical to me. I may not give a rat's hat about "code quality." So for me to buy in to some new approach, especially if it requires me to do something differently, then I need to see how it specifically benefits *me*. And then maybe I'll consider trying this new, wacky thing you're proposing.

With all that in mind, the GROWS idea of better adoption goes something like this:

First, everything is paced according to your skill level. Since we're talking about adoption first, then everyone is a novice at GROWS—they have no experience with it yet. As a novice, there are some practices you want to try, and there are practices that GROWS says you need to start with.

In GROWS, you adopt a practice by running an experiment. Each practice comes with its own experiment, which helps you identify the conditions for the experiment, the feedback to look for and how to evaluate it. For novices, the feedback and evaluation are very concrete and unambiguous, with no judgment required. That part comes later.

Experiments are time-boxed, which limits commitment and risk, unlike the more amorphous "change," which is permanent and open-ended. It's very clear all involved that you aren't yet adopting this practice or committing to it. You're just going to give it a try.

Everyone participates in the experiment and in evaluating the outcome, which gives the participants a chance to "put their own egg in," as the saying goes.

(Back story: When Betty Crocker first came out with an instant cake mix, it was a failure. All you had to do was add water to to the mix. Consumers couldn't relate to it; it was too far removed from the usual cooking experience. Betty Crocker changed the formula so you had to add a fresh egg and water, and now consumers felt like cooks again. That version was a success. Moral of the story: level of participation makes a difference.)

While experiments are used to fuel adoption, they are also in daily use to help us find answers.

"Which design should I use?"

"How long will this take?"

"Which framework is better suited for this application?"

These are all fair questions. You answer them by writing software.

The use of experiments in GROWS helps you to understand that you don't have all the answers—and neither do we, and neither does anyone else. Instead, you answer questions with working software. As in real science, no experiment fails: every experiment provides data to better inform the next experiment.

Our motto: don't guess; prove it.

Practices: The Important Stuff

Even though we want to instill good values of modern software development, people can't just adopt values out of the blue. They can adopt practices if you tell them how.

But as we've seen with a lot of agile adoption, knowing "how to do" a practice is not enough. For each practice, you also need to understand:

- Benefits: what is the motivation behind the practice? What will you hope to get out of it, both long term and short term?
- Where do you get feedback from? With an empirical approach, feedback is everything. Where is the meaningful feedback, and what is just a distracting (or even dangerous) metric?
- What does it look like in the real world? What are the warning signs if not done correctly? These are the things an experienced practitioner may know, but are hard to come by otherwise.
- What can you do to improve your skill at this level? How do you get to the next level?
- At the higher skill levels, how do you then teach this skill to others?

These are the extra bits of information that are critical to each practice, but are seldom explained in any detail. For adult learners in particular, motivation is critical. It's not sufficient to say "just do this," anyone trying to learn these practices needs a compelling explanation of the expected benefits.

If we can expand every practice to provide these additional bits of information, that would be great by itself. But it's not enough. As it turns out, volume and order matter.

When trying to get started with a new approach, teams and organizations very often make one of several common (and fatal) mistakes:

1. Changing too much all at once
2. Adopting new practices in the wrong order

The first problem is well known, if often unheeded anyway. In starting any new endeavor, *only change one thing at a time.* That's the approach we take in GROWS. Eventually, you'll end up changing nearly everything, but do not start that way. Change has to come in small doses.

So what do you change first? And next? Is the order important?

Yes, we believe it is. The first order of business should be the mechanical matters of getting software reliably and repeatably built and delivered: basic hygiene and safety are the first priority. After that's addressed, and your team is building software the right way, then it's time to ensure you're building the right thing by working more closely with the users and/or sponsors. Finally, you need to be able to perform all of these feats on a regular, predictable rhythm.

Together, we call these the *Three R's* of building software:

1. Right Way
2. Right Thing
3. to a Rhythm

The *Right Way* encompasses the various technical practices that are required for modern software development, including basic hygienic issues such as version control, continuous integration and deployment, as well as higher-level technical practices regarding good design and architectural habits.

The *Right Thing* gives us the practices designed to help facilitate better communication with the ultimate users and/or sponsors of the the project. Building the right thing is largely a matter of having low-friction, continuous communication with the end users and sponsors, and some skills at knowing which questions to ask. But it also requires traceability of features from inception to delivery, so that the team and executives can easily get visibility into the work being done.

It's difficult for the team make progress in the right areas when management can't effectively communicate their ideas for the team's direction. So we need practices for executives to ensure that their vision is communicated in a way that can be easily understood and consumed. That then drives day-to-day work efficiently, without falling victim to micromanagement.

Finally, in a sea of chaos and uncertainty, we can take some genuine comfort in a repeatable rhythm of multiple code check-ins and integration every hour, timeboxed iterations every two weeks or so, regular releases every few iterations, and so on. Most aspects of software development are not repeatable, but the *Rhythm* of development is.

What's Ahead

That's our idea for The GROWS™ Method in a nutshell. Experiments driven by constant feedback, guided by ascending skill stages, involving executives, teams, and individual practitioners.

The GROWS™ Method is itself an experiment. We're just starting and gathering feedback. If you'd like to participate, try it, comment on it, teach it, or complain about it, please email us at feedback@growsmethod.com, or sign up for our mailing list at http://www.growsmethod.com.

We do not have all the answers. But we'd like to help you find them.