Teams

FIVE MISTAKES

FOCUSING ON VELOCITY
If you just do the same things, only faster, then you're missing the point.

FOLLOWING THE EASY RULES
If it doesn't hurt, then you're probably not changing enough.

CHOOSING A CRITICAL PROJECT
If failure is not an option, then learning can never happen.

HOLDING PEOPLE HOSTAGE
It has to *at least* be "my way or the highway", not just "my way".

IGNORING TEAMWORK SKILLS
Lingering communication problems will limit your improvement.

Avoid common pitfalls when you begin your agile journey

Five Mistakes New Agile Teams Make

The road to hell is paved with good intentions, and I imagine there is a special section of hell reserved for those who have tried to "go agile".

Naturally, there are some common, large-scale reasons an agile transition can fail. One particularly funny pair of obstacles goes together well: either there is a lack of executive or management support, or not enough buy-in on the ground for a top-down agile rollout. One particularly vexing obstacle comes when business decision-making, and not producing features, is the primary reason that customers are not satisfied. I could go on, but these large-scale problems have one punishing property in common: they feel overwhelmingly intractable. Rather than exert energy pretending to know how to solve these larger organization problems, I'd prefer to focus on relatively simple mistakes that agile pilot teams seem to make. The good news is that we absolutely *can* avoid these mistakes, as long as we know what they are and can see the warning signs. These mistakes are the topic of the day.

Remember, it's just numbers

Focusing on velocity

I can remember my first agile pilot team in 2001. One of my most striking memories of that team was the sheer amount of energy we spent staring at our velocity measurements, trying to make them look better. Looking back, it reminds me of the kid who spends more time figuring out how to avoid doing his homework than it would take to simply do it. Still, it absolutely seemed like the right thing to do at the time, mostly because of distorted priorities.

Remember that this was 2001. Then, the ideas in agile software development were largely discounted out of hand. Here were three people who believed in extreme programming, trying to help it catch on at a company, so not only would we be able to practice it there, but, we hoped, we would be able to persuade other companies to do the same. We wanted to create a local market for XP or agile software development skill, so we would always be swimming in opportunities to do XP. Our mistake was letting that goal interfere with a more important objective: delivering the system we were charged to deliver.

Before you wake up one morning and realize you've wasted months chasing numbers, be aware of some warning signs that your obsession with velocity is getting in the way of your project.

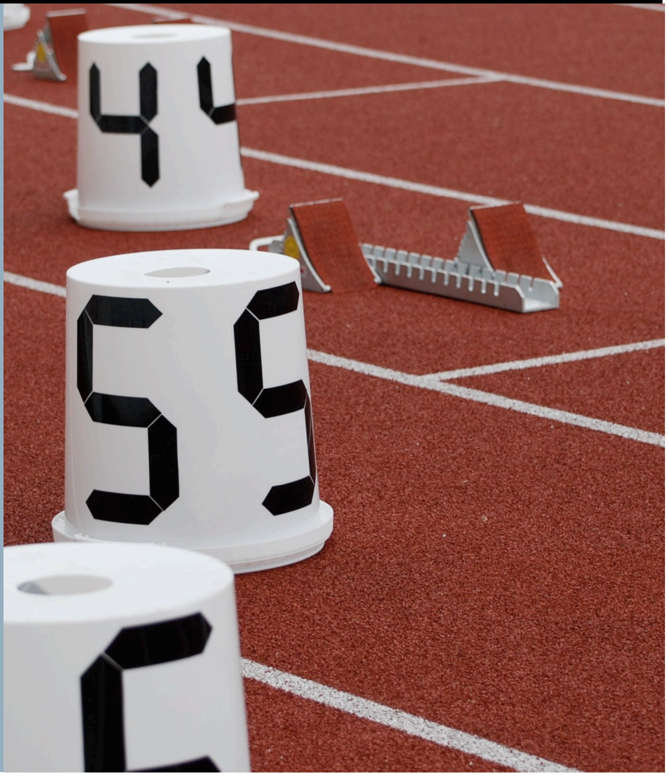
"We're not getting any faster..." The more you hear this, the more trouble you're in. There is a common misconception that going agile means we'll go faster, as though perhaps we'll become better typists or faster readers. One of the principles of agile software development is the idea that we should avoid waste—"maximizing the amount of work not done". One of the ways agile hopes to achieve this is to deliver early those features with the most positive impact on the business, then stop after we have delivered enough. If your velocity curve trends upward from iteration to iteration, that's often a favorable result, but I don't believe it should ever be the goal.

TRY THIS AT WORK

START OFF ON THE RIGHT FOOT

Nothing we do can ever guarantee success, but there are a few key ingredients that I believe contribute to successful agile teams. Do as many of these as you can.

- create a brand new team to become the agile pilot team, rather than selecting an existing team to "go agile"
- give the new agile team thorough training in teamwork and communication along with agile development
- build a team that encompasses all the skills they need to deliver: business or domain knowledge, design and coding, testing, publishing features to production
- let the team decide when it needs to change personnel, and help them make those changes happen



Your reaction to a "slow iteration". How do people react when you under-deliver for an iteration? I understand being disappointed, but do heads roll? Do stakeholders tell you to "go faster"? Do you set a new velocity target that helps you get your average velocity back to its previous level? I've seen them all happen, and they tend to be destructive, not constructive. A team needs to deliver on its commitments regularly to build trust within itself and with the rest of the organization, and merely telling them to go faster (as if they were intentionally holding back) does not seem to help. The most sensible approach I've seen to date consists of examining why the team under-delivered, then lowering its commitment for the next iteration. It's important to do both.

If we analyze why we under-delivered without changing our commitments, then we pretend that we can intentionally catch up later. On the contrary, such teams tend to see velocity spiral downward as they desperately try to catch up, cut more corners, then suddenly—and quite spectacularly—implode. It's not pretty.

On the other hand, if we lower our commitment without analyzing our drop in velocity, we send the message to our stakeholders that they serve at the pleasure of the development team. Not only is this pure fantasy, but it doesn't help: it's the express lane to a canceled project, or worse.

So when you have an off iteration, I recommend you be firm in telling the stakeholders that you need to commit to less, but that you also show those stakeholders how serious you are about improving, by sharing your root-cause analysis and outlining how you plan to avoid those mistakes in the future.

There are more warning signs to look out for, but I have to stop somewhere, so here is a short list to help you out: long discussions after work about how to make velocity "look better", proposals to multiply the estimation scale so the story point values look bigger, and a lack of debate in planning sessions about story estimates.

There is no agile checklist

Following only the easy rules

One of my favorite continuing misconceptions about agile is that we simply stop writing documentation and start writing code on day one. Now I imagine that you already know why that's just not true, but that doesn't stop otherwise well-meaning people from following only the easy rules—or practices—in the course of going agile. It's only natural, and it's a sign that the team sees agile as a set of rules, rather than an approach to delivering software.

Some organizations, to simplify things, characterize going agile as a goal in itself, rather than as part of a strategy to make particular organizational improvements. Now this attempt at simplifying things might be condescension to some and a welcome shield to others, but the results are the same: without understanding the goal of going agile, people will tend to see agile as a checklist they need to complete as part of their work. While this is already a problem, we make it worse when we let the team direct itself, because then the team gets to select which items going on the agile checklist. The result is fairly evident: the team will tend to only put those things on the list it feels it can handle, and won't undertake those practices that face their key problems head-on. The most common case of this I have seen is **eternal arguments about the meaning of done**.

I have worked with a number of teams that start out with little or no control over delivering features to their customers. This happens most often in large organizations with a centralized delivery mechanism, sometimes called "the build team" or "the release team", or both. In this environment, programmers especially tend to see delivering features to real customers as someone else's responsibility. Consequently, they find it easy to adopt test-driven development and to automate regression tests on their greenfield pilot project. The team learns how to write better stories and estimate more accurately, velocity looks good, internal demos to customer proxies go over well and everything appears to run smoothly until the first public release date comes into focus, then it all falls apart.

As the release team becomes more involved, we find out that the team hasn't set up a reliable release mechanism. It has been cutting corners, doing just enough to build and run its internal demos each iteration. As word surfaces that the team needs a few more weeks to be able to release to the public, there is an awful lot of discussion—usually some shouting—about "done". Why aren't the features done? We thought you said you were done! Why did you tell us they were done?! The immediate answer is that the team chose not to agree that "done" means that we can push a button to deliver the new features to real customers, but is the team entirely to blame? I don't believe so. I've never seen a team willfully refuse to use a smooth-running software delivery mechanism.

I have seen a number of teams fall into this trap because they were unable to coordinate their work effectively with existing delivery mechanisms. There are a variety of causes, but the root cause appears to be that no-one wants to change the existing delivery mechanisms, even though they were likely developed by people who never imagined there would be a need to be able to produce a full production release, repeatably, in around 15 minutes, on demand. To go agile, however, you need a feature delivery mechanism that works that smoothly, to enable the frequency and depth of integration that allows you to confidently state that a feature is really, truly done. With an objective measure of "done", we can measure velocity meaningfully and use that information to guide the team effectively. Without it, and in spite of adopting a majority of the remaining practices, we realize only a fraction of the benefit of going agile.

There are other signs that your team is only following the easy rules. A reduction in defects without improvements in velocity, or customers rejecting features regularly during iteration demos, or walking out of retrospectives without a clear understanding of what the team will do differently in the upcoming iteration. If you notice your team doing these things, ask them whether they believe the practices are helping them reach their goals. It's possible they're just going through the motions.

Failure has to be an option

Choosing a critical project as your pilot

It makes sense, on the surface. You've decided to try going agile because your organization needs to be perform better. You need to really shake things up by doing something very different from your current practice, which is why you chose to go agile. To maximize the benefit, then, you want to go agile on a critical project, because that's the most important project to improve—or, more likely, the one you can least afford to fail. Sadly, when organizations choose critical projects as their pilots, they create an environment in which failure is not an option. **If you can't fail, then you can't learn, and going agile is all about learning**.

One of the key underlying principles of agile software development is continual improvement, and we can't improve without learning new techniques or adopting new mindsets. This means that we have to practice, and we can't practice if we're not allowed to fail. There are a number of different signs that a team is afraid to fail. You might hear any of these phrases pop up regularly in conversation.

"We deserve partial credit for that story."

"I can't estimate that story/I'm not starting until I see all the acceptance tests."

"Our part of the story is done, so let's count it as done."

On the surface, these indicate a problem with story writing, or estimating, but I believe that in most cases, they point to the larger problem of being desperately afraid to fail. If failure were an option, then we wouldn't be so concerned about these issues: we would accept the possibility that a story might go wrong and that we could recover from it, rather than trying to create a perfect world in which stories simply can't go wrong. Remember that going agile *means* living with uncertainty by being prepared to adapt to situations as they change, and if you can't fail, then you can't go agile.

Sometimes, we just have to walk away

Holding people hostage

I have to admit that I don't understand how workgroups form in most of the organizations I work with. More accurately, I don't understand how the way most organizations form workgroups can possibly be effective, because it appears to me to be haphazard, or purely politically motivated. I can't tell the difference, but I do feel the results: I am often asked to help a workgroup "go agile" even though the people involved are unwilling, even incapable, of working as a team.

Now I've chosen the words "group" and "team" here quite intentionally. The key difference is that a team works together towards a common goal, whereas a group works near each other on similar tasks. I routinely see organizations try to take a seemingly random workgroup and have it "go agile", forgetting that going agile requires *teams* and not just groups of people. The good news is that the group usually realizes early on that the goals of agile software development compel them to begin to work as a team. The bad news is that the average random group of people won't work as a team, no matter how much we try to make it happen. Time and again, I see would-be agile "teams" run roughshod over one person in the group in their misguided attempt to form into a team. Sometimes you just need to let that person go.

I recognize that your organization is probably neither prepared nor well-suited to accommodating your need to move a person off the team, but that merely explains why it's a common mistake. If letting a person move from team to team is a sensitive issue in your organization, then you need to make people comfortable with the idea before starting your first pilot, because significant change usually causes at least one person to feel sufficiently out of place to need to move. I would go so far as to say that if that doesn't happen, then you might not be changing enough.

It's all about people

Ignoring teamwork skills

I have yet to work with an organization that understands that going agile likely means emphasizing teamwork in a way they've never emphasized before. Commonly such organizations have workgroups, and not teams, except in those fortunate cases where true teams spontaneously formed. Until recently, I thought that teams formed magically, a mysterious process called "jelling". This meant that my agile transitions tended to succeed or fail to the extent that I happened to work with people who did or didn't already work well as a team. When I realized this, it so struck me that I immediately began writing an open letter to apologize for my entire coaching career. I had committed perhaps the greatest mistake of all: I forgot that I was coaching *people* and not feature-delivering machines.

I try not to get too down on myself for this, though, because I simply didn't know at the time that it was possible to understand how groups of people become teams. You might be in a similar position, and if you are, then in spite of your most sincere effort, your group might fail miserably in its attempt to go agile, because it simply cannot work as a team. That's why it's so important to include developing teamwork skills in your plan to go agile.

If you know how to help groups jell into teams, then you're well on your way: teach your agile pilot group everything you know, starting from day one. They might resist it, but they'll be more successful and happier in the long run if you stick with it. If you don't know what to do, consult Patrick Lencioni's excellent work *The Five Dysfunctions of a Team*, which describes a model of five obstacles to effective teamwork as well as how to begin to overcome them. Buy copies for your entire team and devote some time to working through the book together. There is no single book outside the overtly agile canon that I recommend more highly than this one. Even if you're the only one to read it, you can nudge the group in the right direction when you notice them avoiding conflict or showing a lack of trust.

I've seen teams make more mistakes than these five, and by no means are they the most common mistakes or the worst mistakes. They are mistakes that stick out in my head or one reason or another, and frankly, I had to stop somewhere to avoid this becoming a book. Looking back at this list of mistakes new agile teams make, I believe that if you heed their warnings and learn the lessons they try to teach, that that will help you handle most of the other obstacles that will pop up as you learn to go agile.

Best of luck, and have fun.

J. B. (Joe) Rainsberger helps software organizations better satisfy their customers and the businesses they support. Expert at delivering successful software, he writes, teaches and speaks about why delivering better software is important, but not enough. He helps clients improve their bottom line by coaching teams as well as leading change programs. He helps software organizations off the treadmill of over-commitment and under-delivery, addressing all aspects of software delivery including understanding the business, gelling the team and even writing great code. Learn more about how Joe will inspire your software organization at **jbrains.ca**, in his IEEE Software magazine column "Not Just Coding", at conferences worldwide, or by writing him directly at **get.started@jbrains.ca.**

RESOURCES

Patrick Lencioni, *The Five Dysfunctions of a Team*

Kent Beck and Cynthia Andres, *Extreme Programming Explained: Embrace Change (2nd edition)*

Ron Jeffries, Chet Hendrickson and Ann Anderson, *Extreme Programming Installed*

Alistair Cockburn, *Agile Software Development: The Cooperative Game (2nd edition)*

James Shore and Shane Warden, *The Art of Agile Development*

Venkat Subramaniam and Andy Hunt, *Practices of an Agile Developer: Working in the Real World*

Esther Derby and Diana Larsen, *Agile Retrospectives: Making Good Teams Great*

Mary and Tom Poppendieck, *Implementing Lean Software Development: From Concept to Cash*