

# **Agile Project Management**



# Prerequisites

- Experience working as or with a manager
- Experience with an agile project management tool

# What you'll learn

- The history of scientific project management.
- Agile estimation and planning.
- Agile roles.
- Agile practices.
- Agile principles.

# Why agile?

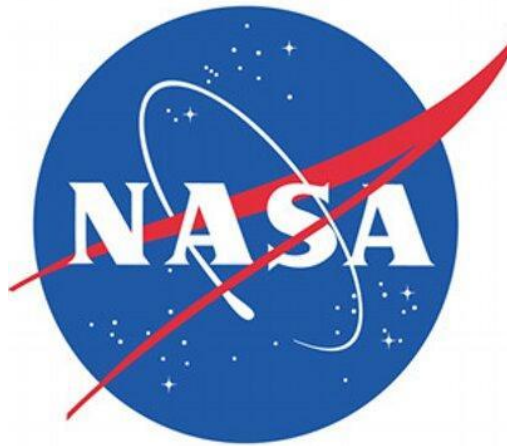
- Because everyone's doing it.
- Because it works well with other things people are also doing.
- Because there is such a thing as "best practices".

Why? This is how professionals make and manage software these days.

Why? Because it works well with a lot of other techniques you'll learn about, like product management, test-driven development, good software engineering practice, and marketing.

Why? Because there's a growing understanding that, while there may not be a silver bullet, there's certainly a list of best practices when it comes to running a business. There's a list of things you should consider doing to maximize your chances of success.

# Before there was agile...



NASA, in its heyday, was the quintessential “mission critical” software. Since lives depended on it, the software development process went so far as to have computer scientists mathematically prove the correctness of an algorithm. That means every path through the code, every possible input, has to be considered, no matter how remote the possibility that it may happen, to prove that no combination of errors could produce a bad result.

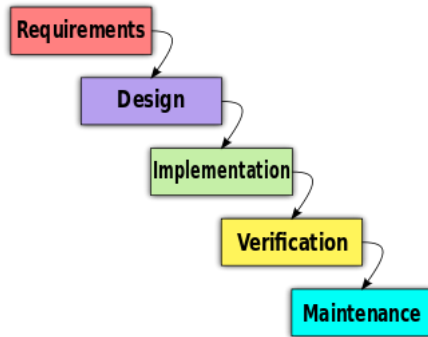
If it was good enough to get us to the moon, it’s good enough for us.

# And for good reason...



The focus on “mission critical” software development process wasn’t misplaced. This device, for example, the Therac-25, was a radiation therapy machine that, in the 1980s, was involved in a series of accidents. It accidentally exposed patients to massive doses of radiation. The cause: a bug in the software. This case study is still taught in engineering schools to this day.

# Waterfall



The resulting software methodology is commonly referred to as the “waterfall model”. The basic premise is that everything in one step of the process should be done completely and perfectly before you can move on to the next step. So you spend a lot of time collecting requirements until you’re done. Then you spend a lot of time designing your system - on paper, without coding - until you’re done. Then you spend a bunch of time coding until you’re done. And so on.

“Waterfall” has since become a derogatory term in the agile community. Any management process you don’t like is “waterfall”. So it’s almost meaningless. But it is a real process. And it’s still in use to this day.



In place of the month-long (even years-long) development cycles of waterfall, we have a new mantra. “Ready, Fire, Aim.” Shoot first, then aim.

When you’re working on a software project that isn’t mission critical, where lives don’t hang in the balance, then it makes more sense to just push out product, even buggy and unfinished product, and make adjustments on the fly.



# R.E.R.O



In other words, release early and release often. Originally coined in a 1997 essay about Linux entitled “The Cathedral and the Bazaar”, the RERO philosophy has been adopted by the likes of Facebook, GitHub, and Etsy.

Facebook releases code twice per day, once for the East Coast team and once for the West Coast team.

<http://techcrunch.com/2012/08/03/facebook-doubles-release-speed-will-roll-new-code-twice-a-day/>

On August 23 2012, GitHub deployed code 175 times in one day.

<https://github.com/blog/1241-deploying-at-github>

Etsy deploys new code, on average, 25 times per day, every day.

<http://www.slideshare.net/beamrider9/continuous-deployment-at-etsy-a-tale-of-two-approaches>

How do we adapt to this fast moving world of non-mission-critical software?

# Scientific Management



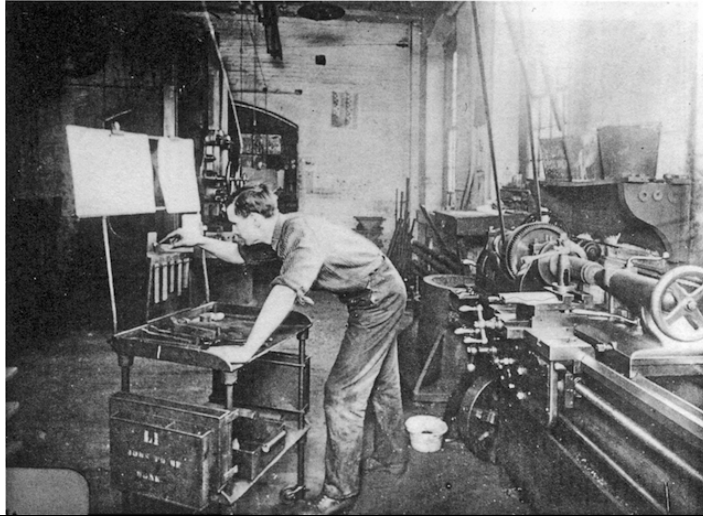
It actually all started in the late 19th and early 20th centuries. In the 1880s and 1890s, Frederick Winslow Taylor, pictured here, began to apply science and engineering to the study of the manufacturing process.

According to a story I first heard from author and agilist Jim Benson, there was once a time when companies actively discouraged innovation. For example, say you were a member of a team of laborers whose job was to shovel coal all day long, emptying train cars coming into town from the mines. One day, you discover a special technique - maybe it's a special way to hold your shovel, or a better way to position your back and shoulders. Whatever it was, you find that you're suddenly capable of shoveling much more coal per day than you used to be. Great news, right?

Actually, no. Terrible news. In the 19th century, when your boss finds out you're suddenly shoveling coal better than everyone else, his first reaction is to punish you. Why haven't you been shoveling coal like this the entire time? You must have been loafing, cheating the company of wages. In fact, your whole team is punished because you've demonstrated that they're all capable of doing more work, but haven't been.

As a result, there's actually a disincentive to efficiency. Everyone is not only encouraged to shovel slowly, they're also encouraged to all shovel all the same slow pace.

# Taylorism

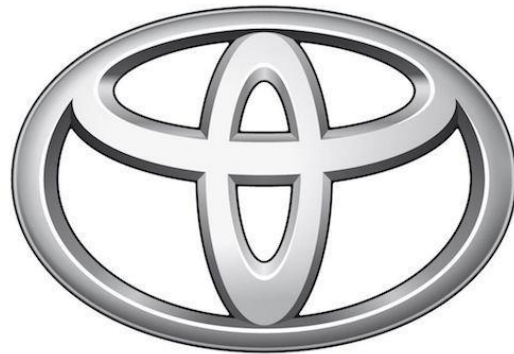


Taylor observed the manufacturing process scientifically. He discovered:

1. That workers tend to work at the slowest rate that goes unpunished. He theorized that workers have self-interest, and won't work faster if it won't result in higher pay. He theorized that if each worker's compensation were tied to his output, productivity would go up.
2. That a careful time and motion study could reveal the best method for performing a task, and that prevailing practices were rarely best practices due to the worker incentive structures in place.
3. That both manual laborers (e.g. coal shovelers) and knowledge workers (e.g. product inspectors) were \*more\* productive when given rest breaks, not less.

Taylor's version of scientific management, called Taylorism, grew popular in the 1910s and 1920s, and gradually developed into modern theories of management. But the major achievement of scientific management was the idea that the process of working is a topic that can be studied scientifically. Like a computer program, it can be refactored and optimized to achieve a particular result.

# Toyota Production System



**TOYOTA**

Let's jump ahead to post-war Japan. Between 1948 and 1975, Toyota Motor Corporation developed the Toyota Production System, a manufacturing management system that served as a precursor to the modern "lean manufacturing" movement (which we won't be talking about).

Ron Harbour, an auto plant efficiency expert, was quoted in a 2007 NYT article as saying "When [Toyota] really went at the U.S. market seriously, in the late 1970s and 1980s, the product they brought out was far superior to what the Big Three were producing".

[http://www.nytimes.com/2007/02/18/magazine/18Toyota.t.html?pagewanted=all&\\_r=0](http://www.nytimes.com/2007/02/18/magazine/18Toyota.t.html?pagewanted=all&_r=0)

# The Toyota Way

Let's go back in time and say you've got a guy who in 1985 bought a Camry, Harbour says. That Camry buyer was surprised to find he never had to get his car fixed at the dealership. "That guy never, ever looked back," he adds. "G.M., Ford, Chrysler — they've basically lost a whole generation of Americans."

*New York Times, "From 0 to 60 to World Domination", 2/18/2007*

Toyota got a reputation for making affordable, quality cars. And small Japanese cars dominated in the 80s. The U.S. wanted to find out why. So they sent academics from management schools, like MIT's Sloan School, and from car country, like the University of Michigan, to study the Toyota Production System in the late 80s and 90s.

Toyota noticed the increased attention and pre-empted it. They publicized their managerial approach and production system in 2001. Before then, in the 90s, they had already begun sharing their techniques with their parts suppliers. They even "donated" the technique to charities. For example, they helped boost efficiency at the Food Bank of New York.

By 2004, University of Michigan professor Dr. Jeffrey Liker had published a book entitled "The Toyota Way".

# Goal: Eliminate Waste

- muri - overburden
- mura - inconsistency
- muda - waste

The goal of TPS is to eliminate waste. Waste is defined very broadly and grouped into three categories.

Muri (overburden) is focused on reducing stress, both on people and processes. Allowing workers to rest would be an example of reducing muri.

Mura (inconsistency) is focused on operating smoothly. For example, you want to make sure the amount of work done each day (e.g. the amount of coal you shovel) is consistent, and doesn't spike up and down day-by-day. Work like the tortoise, not the hare.

Muda (waste) is focused on what you would normally consider to be waste, like:

- overproduction, where you'll have to toss extra product away or store it (leading to overstocks)
- waiting around for something you need to happen
- process inefficiency, like not using "best practices" to do your job
- and bugs: producing low quality output

# Continuous Improvement

改善  
Kaizen

With success linked to the elimination of waste, day-to-day operations become a series of scientific experiments (a process inherited from the “Taylorism”). Try some particular process out for a few days. If it reduces waste, it is a success, and the process continues. If it fails to reduce waste, the process is abandoned.

TPS calls this culture of constant experimentation, innovation, and evolution Kaizen.

# Respect



A culture of continuous improvement requires a mutual sense of trust and respect between team members. When someone decides to try something new, the team is expected to trust that the experiment is well-intentioned. When an innovation is discovered, the individual discoverer is expected to share that breakthrough with the rest of the team.



# TPS Practices

1. long term planning
2. level workload
3. stop and fix
4. optimize routines
5. visual control systems
6. organizational and ecosystem learning
7. reflection

Through 50 years of process experimentation, the TPS process discovered some interesting practices.

1. Base your decisions on the long-term success of the company, even at the expense of short-term financial goals.
2. Provide a consistent and level workload. (Work like the tortoise, not the hare.)
3. Build a culture that allows anyone to stop production to fix problems, and treat that person with respect.
4. Routine tasks are the best place to look for process improvement and employee empowerment.
5. Visualize your process to make problems easy to find.
6. Promote both organizational (internal) and ecosystem (external) learning. For example, teach your partners and suppliers how to work better.
7. Take time, at regular intervals, to reflect on the past and search for lessons.

# Pull System



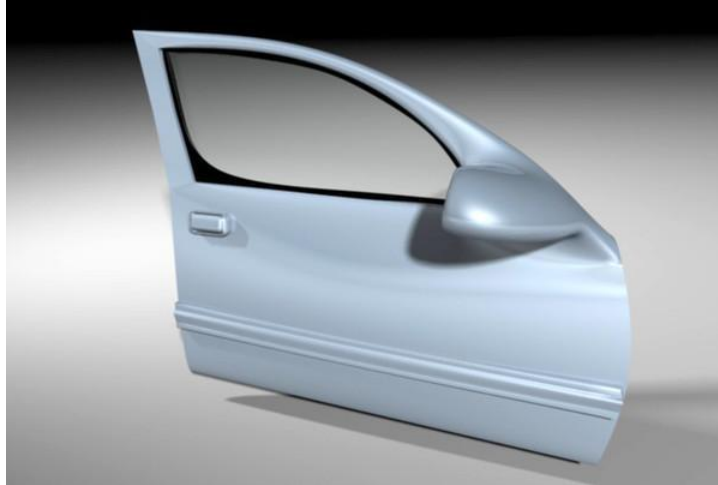
Perhaps the biggest innovation TPS discovered was the “pull system” for inventory control. One of the biggest sources of waste for Toyota was inventory. They would have to store over-produced items, and they would constantly find themselves in a position where they didn’t have enough of something they needed.

There’s a TPS myth that says Toyota founder Kiichiro Toyoda was inspired by American supermarkets. He considered them a model for how he wanted his factory to work. In a supermarket, the store owner stocks the shelves. Then the customers take and purchase what they want. When the shelves look empty, the store restocks the shelves.

In a “push” system, product is delivered pre-emptively, on a regular basis, with the hope that you don’t run out. A “push” system requires planning to overstock so as to prevent the possibility of shortages.

TPS prefers a “pull system” where inventory is restocked “just-in-time” as it’s needed. There’s no need to keep an overstock inventory. As soon as your inventory gets low, you immediately order more so that it can arrive “just-in-time”, before you run out.

# Car Doors




For example, imagine that your job is to attach doors to cars on an assembly line. There's a pile of car doors sitting next to you. You take them, one-at-a-time, and attach them to each car that passes by.

Within that pile of car doors, near the bottom, someone has placed a big red card. That red card is a purchase order. As soon as you reach the red card, you bring it over to your manager, who then orders more car doors. Later, before you run out, more car doors arrive, along with another red card.

In a pull system, there's no need to have an expensive warehouse full of car doors. As soon as you need some, you go buy some.

# Agile Manifesto



## **Manifesto for Agile Software Development**

It was in this historical context, in the early 2000s, around the time the TPS was being publicized, that the agile manifesto was written. At the time, a number of “lightweight methodologies” as they were known at the time, already existed: Extreme Programming, Scrum, and others. The “Agile Alliance”, a group of agile software developers, got together to try to agree on a shared set of principles.

# Agile Principles

Build projects around motivated individuals.  
Give them the environment and support they need, and trust them to get the job done.

The principles behind the agile manifesto should look familiar. This one, for example, is about respect and trust.m

# Agile Principles

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

This principle is about inconsistency (mura).

# Agile Principles

Simplicity--the art of maximizing the amount of work not done--is essential.

This principle is about reducing muda (wasted work).

# Agile Principles

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

And this principle reflects now only the experimental culture of TPS, but also it's self-reflective nature.



# Agile Manifesto

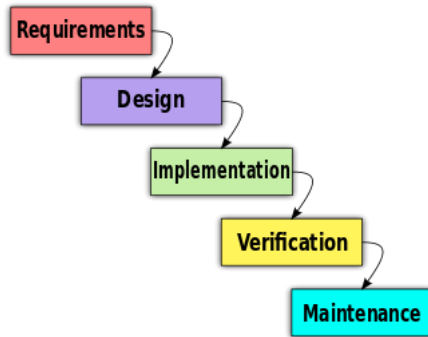
- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

The manifesto itself has only 4 parts. These 4 main values are very revealing. As you gain experience as software developers, you'll begin to appreciate the pain behind these values.

Namely:

- previous processes and tools were painful
- previous documentation requirements were burdensome
- previous contract negotiations were difficult and, in the end, meaningless
- and previous well-thought-out plans had failed miserably

# Developer Initiative



There's a reason developers took the initiative to criticize the predominant methodologies of the time. Checking back in with the waterfall method, you'll notice that writing code doesn't happen until step 3. At that point, everything is mostly theoretical - plans written on paper. It's at step 3, the implementation step, where the first dose of reality is injected into the process. And that's where the developers stood, standing up against wishful thinking and poorly trained managers.

# Developer Motivation



It should come as no surprise that “Dilbert” appears 3 times in the history section of the agile manifesto. Scott Adams’ satirical office humor was uncomfortably close to the daily life of a software developer in the corporate world.

# Management Algorithm



The agile software methodology in practice reads like an algorithm for project management, which makes sense because it was written by software developers. It's a collection of practices that have been tested, in the style of "scientific management" and TPS, through experimentation. Some people like to use all the practices at once. Some people pick and choose which practices work best for them. However you do it, as long as you're practicing continuous improvement and respect, you're still doing agile.

In that sense, even the waterfall methodology could be considered agile if, in your organization, you validate that as the best way to get work done.

# User Story

As a <type of user>,  
I want to <goal>,  
So that <reason>.

Most agile transitions begin with the creation of a User Story. A User Story is usually written in this format:

[slide]

It can sometimes be hard to use this template, but it is recommended.

# Example User Story

As a dog,

I want to be able to order food online,

So that I don't have to rely on people anymore.

Here's an example user story. Note that a User Story specifies the what, but not the how. Sometimes, like in this case, the "how" is more important than the "what". Figuring out the how allows you to clarify the user story.

# Example User Story

As a hungry person,  
  
I want to be able to toast bread,  
  
So that I can enjoy a tasty treat.

Here's a better user story. A user story should be the beginning of a conversation.

Dev: What do you want to happen?

PM: I want the toast to pop up when it's done.

Dev: That's hard to do.

PM: Why? How do existing toasters work?

Dev: They just use a timer. They don't actually know when the toast is done.

PM: Our customers don't want a super toaster. A timer is fine.

Dev: Ok, that's easy.

PM: Cool.

This conversation is a good example of how keeping the end user in mind can drive the product.

# Example User Story

As an: Account Holder

I want to: withdraw cash from at ATM

So that: I can get money when the bank is closed

This is a better user story. It's pretty clear how this should work, but there are still questions.



# Scenarios

1. The account has sufficient funds
2. The account has insufficient funds
3. The card has been disabled

What should happen, for example, if the account has insufficient funds or the ATM card has been disabled? There are three possible scenarios for our user story. How do we express that?

# Scenarios

Given: <persona>

When: <task>

Then: <environment>

This is the format for a scenario, which helps to clarify a particular case of a User Story.

# Scenarios

Given: the account balance is \$100

- And: the card is valid
- And: the machine contains enough money

When: the Account Holder requests \$20

Then: the ATM should dispense \$20

- And: the account balance should be \$80
- And: the card should be returned

Here's an example. Note that it reads very much like a boolean expression, which is on purpose. Since Ruby is so close to English, these kinds of descriptions will literally write the code for you.

Sometimes, despite your best efforts, you forget a scenario. Uncommon scenarios are called "edge cases", and missing them is a big reason why product deadlines slip.

# INVEST

- independent
- negotiable
- valuable
- estimateable
- sized appropriately
- testable

A good user story should be:

- independent: each story should be indep. of each other (this is not always possible, but it's ideal)
- negotiable: you should be able to have a conversation about each user story
- valuable: you should be able to understand why the client needs this user story. You should understand how it aligns with his product vision.
- estimate-able: you should be able to estimate the amount of time you'll need to implement a user story
- sized appropriately: your estimate for the user story should be reasonable in length
- testable: you should be able to confirm that your story meets each of the scenarios (aka acceptance criteria)

# Slices

- user interface
- business logic
- infrastructure

“potentially shippable increment”

Applications are typically composed of 3 pieces: a user interface, some business logic (i.e. the scenarios), and some infrastructure (e.g. a place to store data).

Good user stories should cross all 3 of these slices. That means, upon completing a user story, you should produce a “potentially shippable increment”. That means the resulting code could be a product in-and-of-itself.

# Story Maps

Persona

>> Goals

>>> Features

>>>> User Story

>>>>> Scenario

Multiple stories can be organized into a story map. A story map is like a tree. At the root of the tree is a persona, a description of your ideal customer. A persona has 1 or more goals. Each of those goals involves 1 or more features. Each of those features require 1 or more user stories to implement. And each of those user stories describe 1 or more scenarios.

The story map should be prioritized. The most important goal should be listed first. The most important feature should be listed first. And so on down the list.

# Estimation

Why points?

When estimating user stories, you may be wondering why I've asked you to use story points rather than hours or days. If we're interested in predicting how long it'll take to do something, how do we do that with points? And why?

Good question.

# Estimation Exercise

As your agile facilitator,  
I need you to walk from here to there,  
Because I said so.

Here's a user story. I need 2 volunteers to implement this story (I hope this works...)

[group exercise]

Person 1: Estimate, in seconds, how long you think it'll take to walk from here to here.  
Person 2: Time it.

Did you over- or under-estimate? Over is more likely.

Now let's do it again.

Person 1: Re-estimate, in seconds, how long you think it'll take.  
Person 2: Time it (sidebar).

How did you do this time?

Moral of the story: humans are really bad at estimating in absolute terms. You should have noticed this yourself with your own estimates. Chances are you typically underestimated a task.

But humans are actually very accurate at estimating relative distances.



# Relative Estimation

How big is Neptune?

Let's try an example: Does anyone know the answer to this question?

Seems unlikely. It's 17.15 times the mass of Earth.

# Relative Estimation

What's the size of Neptune relative to Saturn?

What about relative to Uranus?

Does anyone know the answer to this question? Is it smaller or larger? How much larger, do you think?

Neptune is about 5 times smaller than Saturn

Next question:

Neptune is about the same size as Uranus.

Humans are actually much better at relative sizing than absolute sizing. It's much easier to compare a user story to another user story rather than to estimate it absolutely.

# Estimation

complexity

implementation

doubt

A user story estimate is typically made up of 3 parts: complexity, implementation, and doubt. Take into account all three when coming up with your estimate, not just the implementation.

How complex do you think the story is?

How hard do you think it would be to implement it? Something complex could be easy to implement because you've done it before.

How sure are you of the first two?

# Point Scales

Linear: 0,1,2,3

T-Shirt sizes: S, M, L, XL

“modified Fibonacci”:

1,2,3,5,8,13,20,40,200

Up to now we've used a linear point scale. Other options include the t-shirt sizes scale (small, medium, large, xtra-large), and the “modified Fibonacci” sequence. You may be wondering, why Fibonacci? It turns out that this is actually the most popular point scale, for two reasons.

1. It prevents people from getting into a long discussion about whether or not something is exactly twice as hard as something else. The only doubles are 1/2, and 20/40.
2. It also better represents the fact that, as estimates increase, so should doubt and uncertainty. If you think something is really hard, you're likely to be wrong, spacing out the higher values gives you more wiggle room.

You shouldn't spend a lot of time estimating. Just pick a number. In agile, we understand that estimates are just that - estimates. We're not going to hold you to it. But the estimate is your first opportunity to communicate with the team - a low estimate means “no problem”, a larger “we may have a problem”.

# Planning Poker



A common complaint from traditional (non-agile) project managers regarding estimates is “How do you know your developer isn’t cheating you?” How do you know he isn’t padding his estimate to bill more hours for what would otherwise be a simple task?

One solution to that problem is the “planning poker” exercise. I happen to have a planning poker deck here with me right now.

In planning poker, all the member of your development team estimate a task at the same time. Some people will estimate a large number, some a lower one. The point is that the development team have a discussion and come to a consensus about how difficult a story is. The goal is to prevent a single individual from skewing the scale.

However, if you work with a team you can’t trust enough to give you well-intentioned estimates, you have bigger problems than any project management can resolve. I’d also argue you’re not agile because you don’t trust and respect your team, which is an important part of both the TPS and agile.

# Estimation By Analogy

This story is kind of like that story.

Another way to estimate a story is to estimate it relative to some other story. For example, you may say that a new story is similar to an old story, and an old story was 2 points, so this story is probably 2 points.

This technique is also known as “yesterday’s weather”, meaning if you want to know what today’s weather is going to be like, chances are it’ll be similar to yesterday’s weather.

# Velocity



Despite all this talk, I haven't yet answered the main question. How do you convert points into time? At the end of the day, you want to know how long until something is done, not how many story points it costs.

The way to convert points into time is by calculating your velocity.

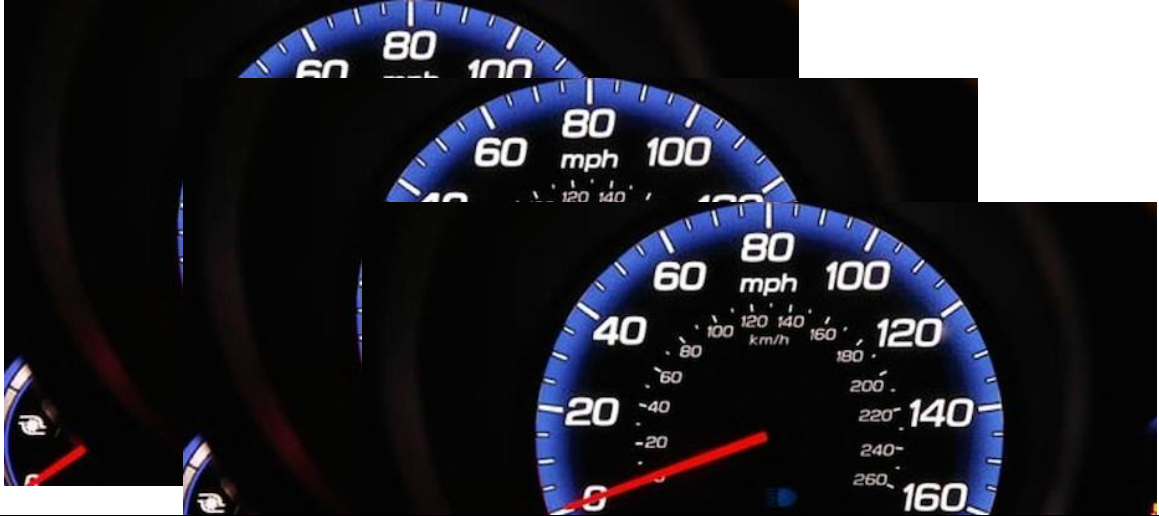
Step 1: Pick some unit of time. This is usually a week, but it can be longer. Shorter is probably not a good idea.

Step 2: Calculate how many points you accomplished in the previous week (or whatever your unit of time is). This number is your velocity. This is another application of the "yesterday's weather" principle.

Step 3: Assume that, going forward, you'll accomplish approx. the same number of points per week.

With the velocity value in hand, you can estimate approx. how long your planned user stories will take to complete. Tools like Pivotal Tracker do these calculations for you, drawing a line in your backlog.

# Rolling Average Velocity



Velocity can vary widely from week-to-week, so one technique you can use to smooth out the bumps is to calculate a rolling average of the previous few week's velocities. That will help you increase your confidence in its predictive power. Some tools do this for you, calculating the velocity as a rolling average of the last 3 weeks.

They say you can't manage what you can't measure. Agile estimation, including story points and velocity, give you the ability to measure your development process. And once you can measure your process, you have a basis upon which to run experiments.

For example, try some so-called "agile practice". Did it increase your velocity? If yes, keep it. If not, try something else. Now you have Kaizen, a process for continuous improvement.



# Agile Roles

Stakeholder

Scrum Master

Team

In a typical agile organization, there are three types of people: the stakeholder, the scrum master, and the team members.

It's important to note that this list doesn't include "project manager". The traditional pm role is split between the scrum master and the stakeholder. This is one of many reasons why traditional PMs are so threatened by agile methodologies.

# Stakeholder



The stakeholder is the business person, the suit. This could be a product manager, a business analyst, an executive, an investor, a client, or a customer. This person is the “single wringable neck” - ultimately responsible to the rest of the company for delivering a project.

# Scrum Master

- agile “project manager”
- process focused
- story points bookkeeper
- scientist
- servant leader

Scrum is a particular flavor of agile. The Scrum Master role is defined by Scrum, but has been adopted generally, and it sometimes called the “Agile Project Manager” role. Unlike a traditional PM, the Scrum Master is not responsible for any of the business-related aspects of the product. The SM should be focused entirely on the process - the how, not the what.

The SM facilitates work, but doesn't dictate work.

The SM coordinates and administers all the other agile practices. For example, the SM schedules and leads team meetings.

The SM keeps track of the velocity and point estimates. The SM always knows the answer to the “one true question”: when will the team be done? When reporting to the rest of the business, that's the only question that matters. Everything else, including cost, can be figured out from there.

The SM is a scientist, constantly running agile experiments to achieve continuous improvement.

The SM isn't a “boss”. Think of the SM as more of an “administrative assistant at-large” or a “servant leader”.

The SM is responsible for removing impediments, anything that's blocking the team from working. For example, “I don't understand this task.” or “I can't start this task until this other task is finished.”

The SM also protects the team from outsiders trying to distract and interrupt the work in progress.

This last point is yet another reason why traditional PMs are threatened by agile. After building up a career and senior position, they ask “What do you mean I’m not the ‘boss’ anymore?” In agile, there is no role for someone whose job is just to “manage” other people’s work. Everyone works *for* the business. The business is the boss. Everyone work *with* people. *For* the business, *with* people, not the other way around. This is why agile organizations typically have flat org charts.

# Team



Everyone else in the business is split into two categories, commonly referred to in agile as the “chickens” and the “pigs”. There’s a story in the Scrum literature, which has been turned into this semi-famous comic strip, describing the difference.

A chicken asks a pig to co-found a restaurant together called “Ham & Eggs”. The pig says no because the chicken would “only be involved”, but he, the pig, would be “committed”. “Ham & Eggs”, get it?

The team are all “pigs”. They are committed. They have skin in the game. Everyone else is a chicken. Part of the SMs job is to protect the pigs from the chickens.

The team are the people who do the work. They’re the ones who write the code or create the designs. They are the ones to whom you can assign a user story. Everyone else doesn’t matter.

# Story Points vs. Hours



Outside of agile estimation and roles, everything else is up for grabs. There's a grab bag of assorted agile practices that you can choose from. Experiment with them all and find out what's best for your organization.

For example, some agile teams prefer to use hours instead of story points. For more traditional developer freelancers, they may already be accustomed to providing clients with time-based estimates. They can continue to do so within an agile framework.

# Sprints vs. Kanban



Another question is the size of your timebox. Some tools assume a week-long timebox (which is called a “Sprint” in Scrum). You evaluate your velocity on a weekly basis. But you can use whatever sized timebox you’d like: 1 day, 1 week, 2 weeks, 1 month. It all depends on your organization.

You can even ignore timeboxes all together. In the Kanban style of agile, you don’t worry as much about velocity. There’s simply a list of tasks, and you work through them in order from most important to least important. When the feature is done, you release it. In Kanban, the answer to the “one true question” is a little different. When asked “When will this feature be done”, you’d respond with “next” or “after this one” rather than with a specific time.

This kind of release cycle is appropriate for projects like Facebook and GitHub. They release features whenever they’re ready. They’re not racing to beat some competitor to the market. They just provide a steady, continuous stream of new features and bug fixes on a daily basis, even multiple times a day. Everyone trusts that the team is working as fast as they can, so there’s need to use deadlines for additional motivation.

# Stand-up Meetings

- team-focused accountability
- *once* daily status checkins

Scrum first introduced the concept of the daily standup meeting. In a flat organization where you're no longer reporting to one main manager, who motivates people to get things done? You can no longer depend on the traditional PM to "manage" other people's work. Teams self-manage. So motivation has to come from elsewhere.

The stand-up meeting is a psychological ploy. Instead of promising a single manager that you'll accomplish something in a certain timeframe, you're promising everyone on the team. All those eyeballs staring back at you should, at least theoretically, be more motivating (or maybe more intimidating) than one person. You're making a commitment to your fellow coders, people you should trust and respect.

The fact that it's called a standup meeting is another psychological ploy. Developers don't like meetings, especially long ones. They allow this meeting, but only under the condition that everyone stand up. The idea is that standing up will force the meeting to move quickly because people are lazy.

The standup meeting also solves another pet peeve with traditional PMs: the random status check. Traditional PMs have a habit of going to every developer, one-by-one, and asking, face-to-face or via email or IM, for a status update. And these discussion can happen at any time during the day, usually when the PM is feeling stressed for whatever reason. But it can be very distracting and annoying to have to stop in the middle of a task to answer a question like that. So, instead, the daily standup is a way to tell the PM: this is the time to ask for a status update. After this meeting is over, you're not allowed to bother me anymore.



# Visual Control Systems



PivotalTracker



asana:

Atlassian  
**JIRA** Agile



For the PM who just can't help himself (or herself), there's the practice of using a visual control system, which comes from the TPS. The idea behind a visual control system is that, at any point in the day, a PM can view some shared, collaboratively edited document that lists the current status of everything in the project.

In the old days there used to be spreadsheets that were emailed back-and-forth. Then we started using Post-Its on boards. Some people switched to Google Drive Spreadsheets. But mostly people use one of the many online services providing digital agile management solutions. And there are quite a few.

# Agile vs. agile

Agile  
prescriptive

agile  
descriptive

The list of agile practices is long. There's something in it for everyone. But the point is that "agile project management" isn't just one thing. You can pick and choose which practices work best for you and your team.

There are those who believe that you're not doing agile if you're not doing certain things. Extreme Programming (XP), for example, is typically more strict than other forms of agile (although some would disagree with that). Those who prefer capital-A Agile tend to believe that certain practices are a prescription for project success. You have to do them all or it won't work.

Most agilists are lowercase-A agile. They describe how certain teams find success using agile principles, and ask that you do what's best for your team. For lowercase-A agilists, "Are you agile?" isn't a yes or no question. There's a gray area.

# Agile Principles

- transparency
- trust
- accountability
- change
- simplicity
- agility
- happiness

In general, all the agile practices share a common set of beliefs. At a basic level, when evaluating an agile practice for your team, you want whatever stats you're collecting for your team to go up. If they do, you should continue that practice.

While that kind of experimentation answers the “what” of the hypotheses, as in “What will this practice do to my team?”, it doesn’t answer the “how”, as in “How is this practice making my team better”. The “how” is better expressed by some basic agile principles. If an agile practice re-enforces one of these principles, then it’s likely to improve your team.

In other words, just like with the TPS system, Toyota’s end goal was to boost the bottom line, but the “Toyota Way” of doing so believed that continuous improvement and respect were \*how\* they would accomplish that goal. Practices that improved either of those things were assumed to, in the end, make the company more successful.

Here’s my list of 7 agile principles. Other people have their own lists. This is a list I came up with after having experienced and talked about agile for a few years.

[slide]

# Transparency

- know what's up
- always be present
- continuously demo
- continuous feedback (no "2-week freakouts")
- continuous (re-)prioritization
- know the answer to "the one true question"

You should know what the heck is going on at all times. I consider this my most important value. You can't manage what you can't measure. So you should be measuring all the time.

How do you achieve transparency? Through constant communication. That means talking to people - collaborating, coordinating, even just socializing - on all of your devices - all the time, all day long (until you sign off for the day).

This is the reason why I prefer to use group chat apps like Skype, HipChat, and Campfire. Always be present, even virtually when you're not in the same room together. You can also use email too (if you have to), but I don't recommend it. It's too slow. Show everyone that you're there.

Team members need to be transparent to the stakeholders. That means, for example, showing the stakeholders what you're working on *\*before\** it's finished. Have a demo available at all times. For example, regularly deploy your code to a test site that anyone can access at any time.

Team members should be transparent to their fellow team members and the scrum master. Show your co-workers what you're working on every day (for example, during the standup meeting, or by keeping your online tool up-to-date in realtime). Prevent duplicated work (e.g. "waste"). Understand who "owns" what piece of code (who's primarily responsible for it). Understand how each of the pieces of code depend on each other.

Stakeholders should be transparent as well. They should be providing continuous feedback. An involved stakeholder is a satisfied stakeholder. A silent stakeholder is playing Russian Roulette. If the project doesn't go as planned, the first question the stakeholder will get is "Why?". An uninvolved stakeholder won't know the answer to that question. Remember, he's the single wringable neck. He has ultimate responsibility for the project.

Typically, an uninvolved stakeholder will experience the "2-week freakout". After not having checked in with the project for 2 weeks (sometimes less, sometimes more), he'll suddenly panic, shut the whole project down, and demand a full accounting. This is bad.

Stakeholders should also be transparent about priorities. The list of user stories should be ordered from most to least important, with a clear understanding that stuff at the end of the list has a higher chance of not getting done on time. It's not priority in the sense of importance. Everything is important. It's priority in the sense of the physical reality of the universe, meaning the team can't work on everything at the same time. You have limited resources. Unexpected things happen. Spend them wisely.

Over time, priorities change. Daily/Weekly re-prioritizations are not uncommon.

Agile is not magic fairy dust you can sprinkle on your team to make them move faster. The stakeholder is a real job with an active role. Agile isn't just for the devs. It's for everyone.

Transparency also applies to the entire agile team: dev team, scrum masters, and stakeholders. The entire team should be transparent to the rest of the company. A big part of that is always having an answer to the "one true question", which can come in many forms: "when will you be done?" or "how soon until I have something I can play with?" or "when will we ship this?" or "when can I tell the client this will be ready?"

# Trust

- truthful transparency
  - trust the visual control system
- treat people like professionals

Transparency doesn't work without trust. Trust that, when we communicate, we're being honest. We need to trust that all the data we're collecting is not misleading or lying. That means trusting all the data in your visual control system.

That also means treating everyone like a professional. Professionalism is not just a buzzword. It's a real thing. Would you argue with your doctor over a diagnosis? Would you argue with your lawyer over legal strategy? Then why does everyone feel so comfortable negotiating how much time it takes to build a website? We're used to treating people in certain professions with a certain amount of respect: respect for what they know, respect for their experience. But professional respect isn't just for doctors and lawyers. Treat your team like professionals as well.

This is hard for some people to do. Business is all about negotiation and competition. We lean on contracts and face-to-face interaction to make us feel good about a deal. The problem is that most software developers aren't business majors. They don't know how to negotiate. Instead, they fail silently. The dev will say a feature will take a week to implement. You'll ask if it can be done in 2 days. He'll just say yes. You'll think you've won, but the dev will just fail, quietly, without letting you know ahead of time. You didn't trust his professional judgment.

# Accountability

- take responsibility for your data
- give the benefit of the doubt:
  - usually people don't fail, processes do
  - usually people don't fail, teams do
  - sometimes people fail
- fail fast, fail early

The flip side of trust is accountability. If you're being treated like a professional, you should act like one. That means being professionally and personally responsible for the information you report and the data collected about you.

Accountability is a motivator. You should consider your professional reputation to be on the line at all times.

Accountability doesn't have to hurt. However, bad reactions to accountability can hurt. Let's say, for example, that something "goes wrong". The team misses a deadline, or they hit a deadline, but there are lots of bugs in the release. What then?

Rule #1: In most cases, when something goes wrong, it's a problem with the process, not the people. Everyone has to fight the urge to blame someone when things go wrong. Usually the problem is that the processes you're using isn't as fool-proof or fail-proof as you had hoped. For example, there shouldn't be unexpected bugs in a release if you're using a good automated testing strategy.

Rule #2: The team rises and falls together. If someone on the team fails, the whole team fails. Everyone suffers when the business fails. So another way to fight the urge to point fingers is to accept blame collectively.

Rule #3: Realistically, sometimes one (or a few) people are at fault. Stuff happens. In that case, if it's you, take responsibility, suffer the consequences, and move on. Act like a professional.

Regardless of who was to blame, failure should never be a surprise. Transparency reveals failure early. That's a good thing, even if it means you lose a client or customer. Failing early and predictably is \*much\* better than failing, not telling anyone, and having the client find out on their own after the fact. Be honest about your failures. Have integrity. Fail fast and transparently. (This, btw, is the flip side of RERO (release early, release often). If you're releasing early and often, you're probably failing a lot as well).



# When you fail

- Pivot
- Fire your customer
- Fire team members
- Stay professional

What do you do when you fail? Not \*if\* you fail, but \*when\* you fail.

Failing can come in many forms. For example, you may decide that a particular product or strategy just isn't going to work out. Instead of completing the project just because you have some momentum, have the courage to cancel it, cut your losses, and pivot. Pivoting, a technique borrowed from the Lean Startup methodology, is the process of changing product direction. We'll learn about Lean Startup later.

Failing can also come in the form of firing a client or customer. Sometimes things just aren't working out. You should cut off bad relationships before they take your business down with them.

Sometimes the consequences of failure mean that a business can no longer support a particular team size. It's always hard to have to let someone go. But if you have to do it, it's usually best to have some data to back you up, rather than a general feeling of who seems to be the most valuable player. You can, if you must, calculate individual stats (e.g. individual velocity, # of times a deadline was missed, etc.). Don't make a habit of it. But, in a world of wrongful termination lawsuits, it's always better to have some math to back your decision.

In the end, regardless of how you fail, always behave like a professional.

# What about liars?



One of the most common questions/criticisms I hear about agile is “What do you do if someone is lying to you?”. There are business owners out there who don’t believe they can trust anyone. They convince themselves that agile doesn’t work with liars, so more traditional project management techniques should be used instead.

No so fast. First, these kinds of business owners need to admit they have a problem. You can’t build a business if you can’t trust anyone. They need to work on that first.

Second, if you do think someone is lying to you, how do you expect to prove it? These same untrusting business owners are also most likely to rely on their “gut feelings” to let them know something is wrong. That kind of management style may make for a good story, but it’s not science. If you expect to be doing “scientific management”, you better have some data to back you up. And agile project management is all about the data.

If you think someone on the team is lying to you, then run an experiment. Assign the same user story (or stories) to multiple people (or, if necessary, multiple teams). Do it in secret if you must. Use the agile metrics to compare how two different people or teams accomplish the same tasks. If someone completely new to the project finishes a task sooner and with higher quality than someone you’ve been working with for a while, you have a problem. That’s real proof that something may be wrong.

# When you succeed



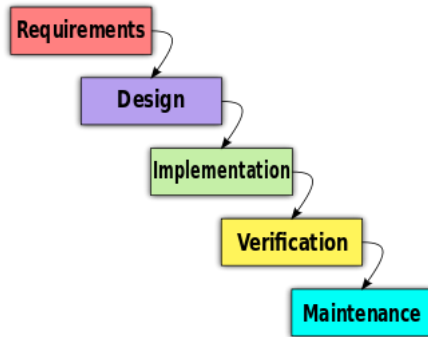
Accountability isn't only about dealing with failure. You should also, perhaps even more importantly, learn to deal with success. When things go well, celebrate. Give credit to the people and processes that do the right thing. Celebrate success. Use positive reinforcement.

The “retrospective” meeting is a great time to talk about what went right and what went wrong. The “retrospective” typically happens at the end of your time box. So if you measure velocity on a weekly basis, you may have a retrospective meeting every Friday. Or you can save it for the end of a particular project. In Kanban-style agile, where there is no timebox, you can schedule it for once every few weeks or once a month.

Regardless of when you schedule it, it's the perfect time to celebrate success. It's also the perfect time to bring up any big issues that may have been bothering you during the sprint. Rather than have a disruptive conversation in the middle of the week, ask everyone to save their feedback for the retrospective. That prevents the project from being derailed by unsatisfied team members.

Pro tip: Retrospectives are a good time for alcohol. You want people to be honest - honest about the good and the bad.

# Change



Perhaps the biggest criticism agile has for the waterfall methodology is its resistance to change. If something comes up later in the chain, it's too late to inform the people further up the chain to make adjustments. They could literally have left the company at that point.

Agile embraces change. It admits that change is inevitable and good. It's not afraid of change.

An agile team is always prepared for change. They don't whine when it happens. They're professionals.

Change, as a principle, perhaps best reflects the dictionary definition of agility as "the ability to move quickly and easily".

# Types of Change

- stakeholder changes
  - competitive landscape
  - new technologies
- team changes
  - resilient code
- role changes

Agile embraces all types of change, not just simple re-prioritizations.

Stakeholders understand that embracing change is good for business. The tech industry moves fast. Business plans should not be dogma, so software projects shouldn't be either.

The agile development team understands that embracing change is good for the code. They build the expectation of change into the product, so it doesn't hurt when it happens (e.g. "defensive programming").

The team as a whole understands that embracing change is good for the company and their careers. That means even the agile roles themselves are soft. You may have to step out of your comfort zone to contribute to the company in ways you're not trained or qualified for. That's ok. Always be learning. Expect to adapt your role. Don't let your title limit you.

# Risk Management

- if you want more, spend more
- iron triangle
  - time
  - money
  - product



Managing change, in general, is a “risk management” task, which has its own “best practices”. We won’t go into detail, but in-a-nutshell:

Changes should have easy-to-calculate consequences. That’s what the visual control system is all about. Basically, if you want more, you have to spend more (for example, if you want more product sooner, you need to increase your team’s velocity). Transparency is your friend here. If you know what’s going on, you’ll be more comfortable adjusting when new facts present themselves.

Typically, the trade-off to consider when evaluating an unfavorable change is expressed by the “iron triangle”. You have time, money, and product. You can only pick two.

If you need more time, you can spend more money or reduce product quality. Reducing quality may mean testing less, but more likely it means implementing fewer features (a smaller product).

If you need more money, you can either spend less time (deliver sooner) and/or reduce the size of the product.

If your product is too small (needs more features), you can spend more money and/or spend more time to compensate.

In the best of times, we work in organizations where increasing the size of our product

costs more time and money. That's a normal day. But when a change forces you to decrease time, decrease money, or increase product, then you need to make a choice.

# Simplicity

- complex projects, not complex tasks
- plan and work in small increments
- document as-you-go
- beware of MS Word
- demo early, demo often
- decomposition

The next agile principle is simplicity. There are big and complex projects. There are not big and complex tasks. The goal is to focus on the small tasks required to reach your goal rather than the full-picture.

Simplicity can also be called incrementalism or iterative development. Like evolution, the idea is to keep things simple, and advance complexity incrementally. We've talked about that in the context of software design, but it also applies to agile project management.

In agile, simplicity is expressed by, for example, not waiting to document an entire project before starting to work on it. Waiting for full-documentation is quintessential waterfall. Instead, just document the next iteration of tasks (say, a week), and just have an outline for what should come after. Then expect things to change.

In practice, that means, for example, that you should be very afraid of the giant Word doc. Big project documentation, especially if it can't be easily updated collaboratively (like, say, a Google Drive Document), is likely to be out-of-date as soon as it's written. No one wants to keep big documents up-to-date. They have actual work to do.

This lack of complete documentation can freak some people out. You can start working on a project before you even know how it will end.

Instead, focus on continuously delivering a demo site. Documents can be misunderstood. Demos cannot.



If you're working iteratively, then your demos should be simple. You should only have a few new features to demo after each iteration. And the demo should get better and better after every pitch.

Sometimes even an individual task can seem too big and scary. When that happens, you have to break it apart (a process known as decomposition). If a big task can't be decomposed, send it back. It probably means there's not enough detail.

# Pacing

- Reduce mura: inconsistency.
- Aim for a constant velocity.

Simplicity can also be reflected in project pacing. As Toyota discovered, an agile organization should aim to reduce mura (inconsistency). That includes the pace of work itself. You should aim to maintain a constant team velocity.

A constant velocity is a way of keeping your schedule “simple”. The number of hours your team works should be consistent and maintainable. You shouldn’t be rushing all the time. Burnout is bad.

Being disciplined about pacing should force you to avoid biting off more than you can chew. Don’t try to stuff too much work into a single iteration. Accept when something can’t be done. When pressed, instead of telling the stakeholder “no” or “we can’t”, say that a particular schedule is “not mathematically possible”, and use your agile metrics to back you up.

Pacing applies to the individual as well as the team. So much so, in fact, that some agile organizations will show preference for the developer who has a consistent velocity over a dev who doesn’t, even if the erratic developer is more productive on average. A consistent velocity makes your velocity-based predictions more accurate. Being able to say you’re going to do something at a particular time, and then doing it on time, can be more valuable to the business than having a guy who can pull crazy hours and make miracles happen at the last minute.

# Agility

- Be agile about your agile
- Be transparent about your process
- Experiment with your process
- Continuous improvement
- Scrum Master as scientist

The 6th agile principle is agility. This isn't change, per se. The point of this principle is that you should be agile about your agile. This is the principle that distinguishes descriptive (lowercase-a agile) vs. prescriptive (capital-A Agile). Don't have a lot of rules. Don't treat agile practices as "sacred cows".

The scrum master should treat the agile process itself the same way the agile team treats the project. Be transparent about how well the process is working. And be prepared to change it.

For example, say your team does daily in-person standup meetings every morning. Then one day you decide to start hiring remote workers. What do you do? Try to do agile without them? Use a webcam? Change the standup meeting to a chat or email checkin? Maybe you need to change the meeting time because of time zone differences.

The point is that there's tons of room for innovation in agile. IBM, for example, experimented with holding virtual team meetings inside Second Life. Many teams experiment with non-traditional work hours to make life easier for workers in other timezones or workers with kids - even Miami traffic.

The agility principle best represents kaizen (continuous improvement). Each new practice is an opportunity to run a new experiment. If your numbers go up, the experiment worked.

Agile experiments can be very silly. Some people test if putting plants in the office will affect velocity. Some people test natural lighting. This kind of experimentation can be a full-time job for an agile project manager. Be a scientist. Learn how to use the scientific method.

The point is that there are still things left to be discovered in the agile landscape. There are many opportunities for cross-disciplinary influences. Maybe there's something from Economics or Politics or even Computer Science that you can use to improve your team. It's still wide open.

# Happiness

- psychology is a big part of agile
- work is stressful
- agile “ceremonies” should not be
- the work environment should not be
- if you’re not having fun, you’re not being agile

The 7th and final principle is happiness. Just like Ruby focuses on developer happiness, agile project management focuses on the happiness of the entire team.

By emphasizing happiness, I mean to emphasize the importance of psychology in agile. At the end of the day, the process is about people. It’s about motivation, satisfaction, and moral. Basically, the assumption is that the less stress a team feels, the more work they do, and the better work they do.

Admittedly, transparency and accountability are stressful. Change can also be stressful, especially if you’re not ready for it.

But your daily work routine shouldn’t be stressful.

The agile “ceremonies” - standup meetings, retrospectives, story estimation - shouldn’t be stressful.

The work environment (meeting location and times, conference rooms, your office space) shouldn’t be stressful.

Basically, if you’re not having fun, you’re doing something wrong. And you’ll probably fail. And you’ll probably blame agile for failing.

If you’re not having fun, you’re not doing agile correctly.

# 7 Agile Principles

- transparency
- trust
- accountability
- change
- simplicity
- agility
- happiness

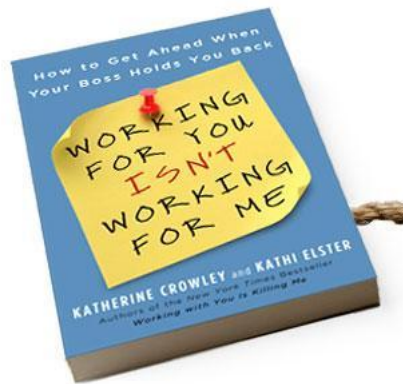
So here are all of my agile principles. When some high-paid consultant recommends some agile practice, or when you read some blog about how some team had a huge breakthrough after implementing some agile practice, you should ask yourself:

- Does this practice make our work contributions more transparent and measurable?
- Does it demonstrate or promote a sense of trust and respect in myself or my co-workers?
- Does it improve my ability to hold people or processes more accountable? (For example, does it help me better judge the effectiveness of something?)
- Does it help me better respond to change?
- Does it help me iterate faster or better?
- If I'm not comfortable with the idea, can I test it?

And finally,

- Does the practice make my team happier or less stressed out?

# Agile isn't for everyone



Agile isn't right for everyone. Sometimes agile principles conflict with a company's culture or the way a team wants to work. Agile enforces a particular company culture, and that culture isn't for everyone. Some companies don't like transparency. Some companies thrive with big, top-down hierarchies. Some industries still use waterfall methods. And that's fine. That's what makes them happy.

And you don't have to work for those companies, if you don't want to.