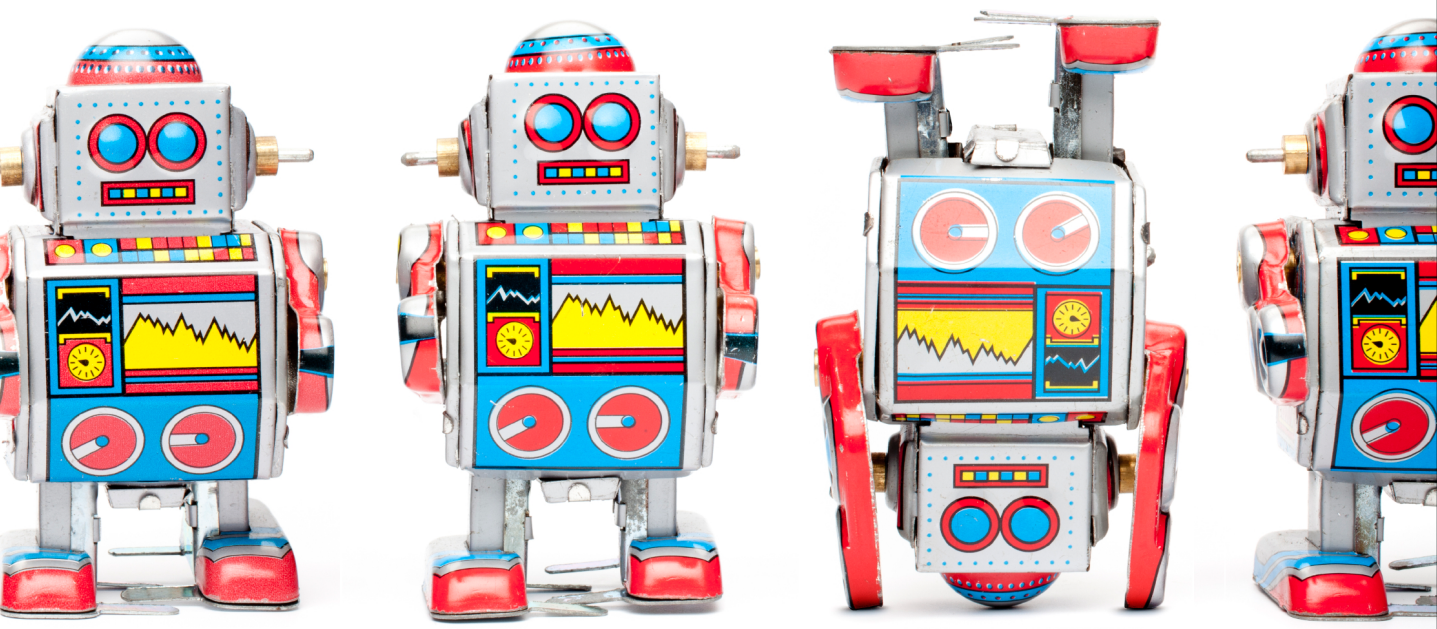
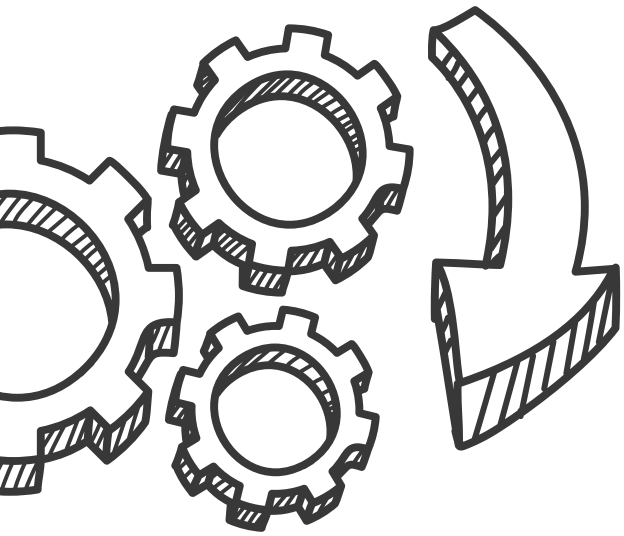
Leading *agile* teams



by Doug Rose



Project Management Press
14525 SW Millikan Way #32865
Beaverton, Oregon 97005-2343

Paperback Edition Printed in the United States of America
Paperback ISBN-13: 978-0-9864356-0-7
Kindle ISBN-13: 978-0-9864356-1-4
ePub ISBN-13: 978-0-9864356-2-1

Chapter 3

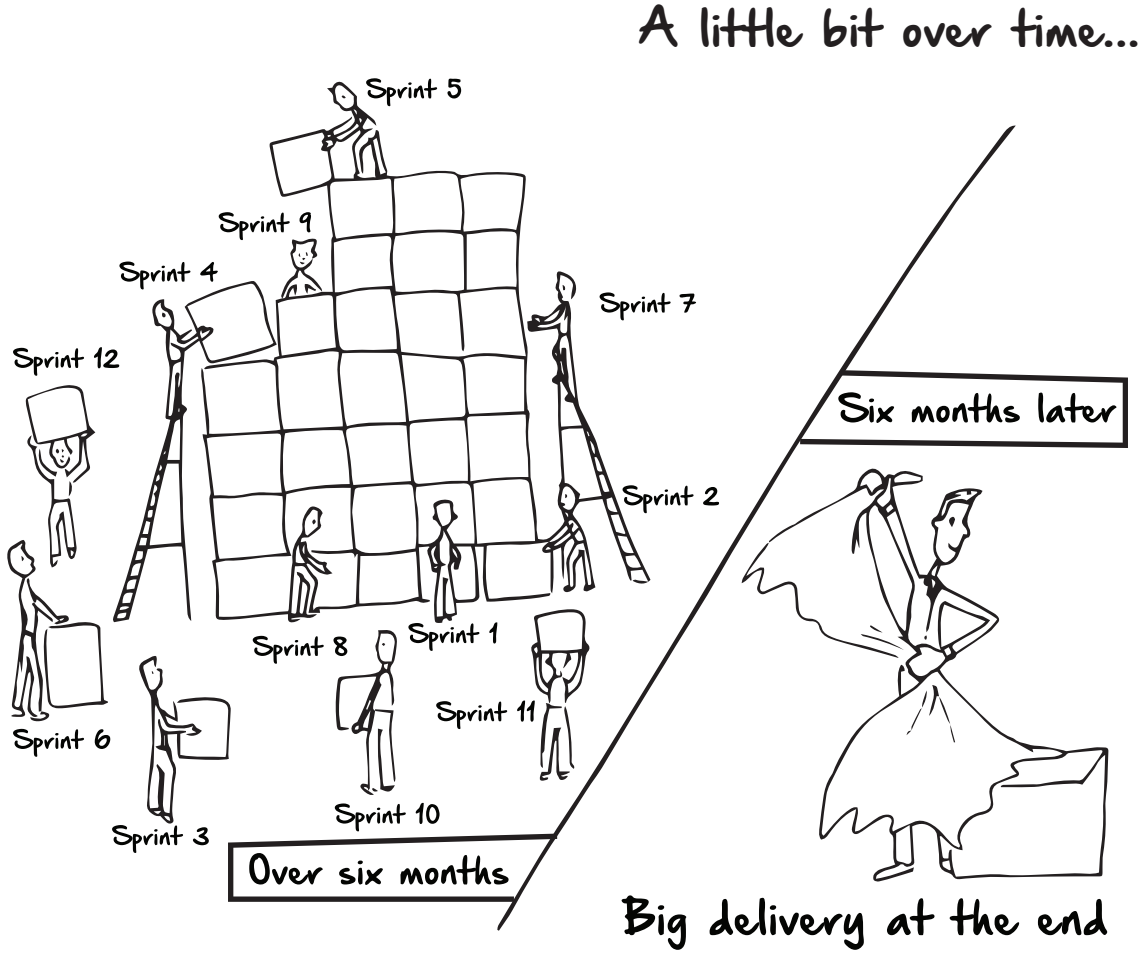
Planning with Agile User Stories

Planning Incremental Delivery

> Small deeds done are better than great deeds planned"
> – Peter Marshall

The agile framework uses short durations of work called iterations. In scrum, these are commonly called sprints. A sprint is a short completed deliverable which can be improved over time. This is much different from a traditional project. In a waterfall project, there is one final product released at the end.

This might sound like a subtle difference, but they're actually two different ways of working. One way is creating something bit-by-bit and improving it over time. The other is incomplete work finished in phases and delivered at the end.

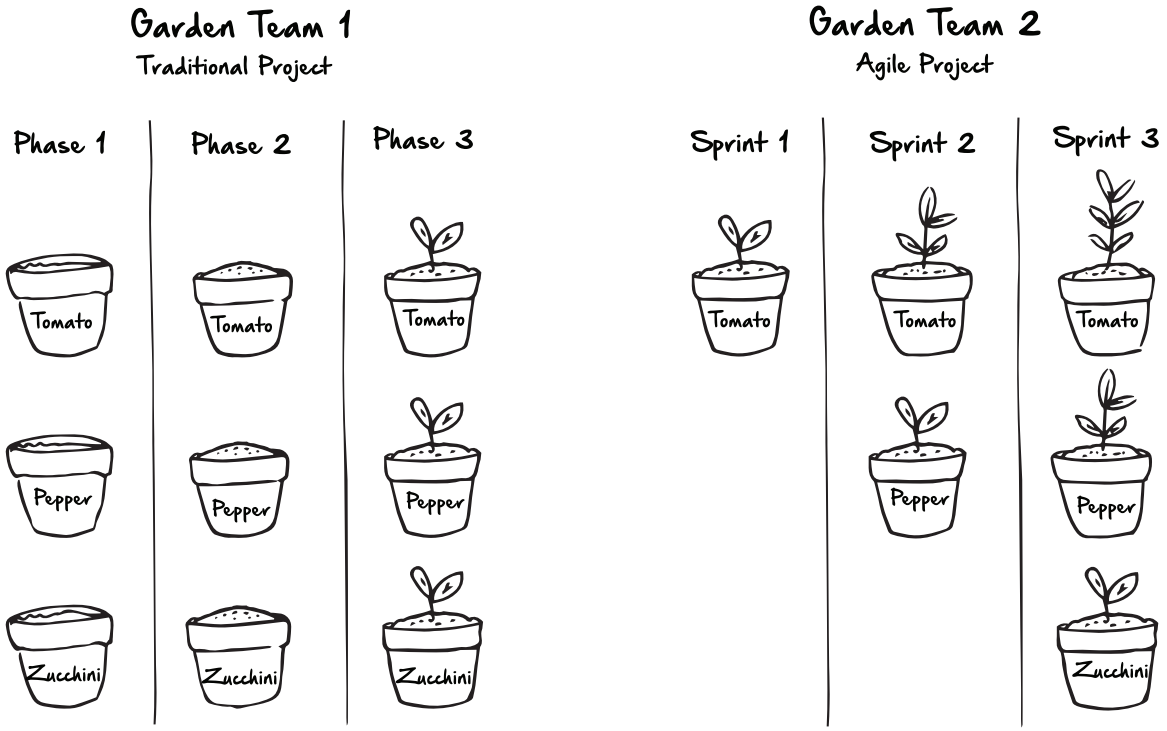


To see the difference, imagine you hired a team to build a vegetable garden. You want to have three planters, each with vegetables. The first planter would have tomatoes. The second planter would have peppers. The final planter would have zucchini.

Each planter needs to have soil, a special fertilizer and seeds. You get proposals from two different gardening teams to help you with your project.

The **first team**, following the waterfall approach, recommends that you finish the work in three phases. In the first phase, all three planters will be filled with soil. In the second phase, all three planters would be filled with the correct fertilizer. In the final phase, all three planters get seeded with each vegetable.

The **second team**, following the agile approach, recommends that you finish your garden bit-by-bit over three sprints. This team will start with whichever vegetable is most valuable to you.Tomatoes are your favorite vegetables, so you ask them to start with this planter. For sprint one, they add soil, fertilizer and tomato seeds to one planter. For sprint two, they add soil, fertilizer and pepper seeds to the second planter. For sprint three, they add soil, fertilizer and zucchini seeds to the final planter.



Think about it!

If you think about it, the agile team would have a lot of advantages over the first. The first advantage is that you get a completed planter every two weeks. Your favorite plant will be delivered first. It will start growing after the first two weeks. If you went with the waterfall team, you would have to wait for six weeks before anything started growing.

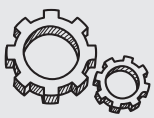
You'd also have a lot more flexibility with the agile team. Let's say after two weeks your tomato plants really take off. So you decide to cancel your pepper and zucchini plants. Instead, you ask the second team to plant tomatoes in all three. There is a good chance that you won't have much rework. At worst you just have to redo one whole planter.

With the waterfall team, you'd had to wait until the project was complete. Then you might notice that the tomato seeds are really growing. But by that time, it would be too late. You would have to redo the entire project if you wanted three planters of tomatoes.

What would happen if you ran out of money before the project was complete?
If you went with the agile team, you won't have a full garden but you'd at least have one planter. It would also be your favorite vegetable. If you went with the waterfall team, you'd have three planters filled with soil and maybe some fertilizer. Not really anything of value. You'd have just a few half-filled planters without any seeds.

The more uncertainty there is with the project, the easier it is to deliver in sprints. You wouldn't be able to deliver the project with the waterfall team if the customer was fickle about their plants.

What if?



Now what if you didn't want to hire any garden teams?
Let's say you're a do-it-yourselfer and wanted to create your own garden. How would you approach the work? Would you completely finish one planter at a time bit-by-bit? Or would you do the more phased approach? First, you get them all filled up with soil. Then go back and do the fertilizer. Then, finally, you'll finish them all at once.

Chances are you would do the waterfall approach. You would most likely deliver the project in phases like the first team. That's just how most people think about work. They try to finish one big thing first then go onto the next big thing. Finally wrapping it all up at the end.

One of the biggest challenges with agile teams is breaking away from this phased mindset. Many agile teams struggle with this idea of delivering the project bit-by-bit over time. Often what they wind up doing is just renaming phases to sprints. Then they'll come up with a big unfinished deliverable at the end.

The product owner should always be driving the team to incrementally produce valuable deliverables. They may be knocking the team off track if they start running sprints like traditional phases. It's not that difficult to spot when this is happening because the deliverable will start to sound like an IOU.

It's very common for an organization that has a lot of experience with traditional project management to struggle with this difference. Incremental delivery is not just a way to divide up work. It's an alternative way to think about work. Keep your eye out for these renamed phases. They will almost certainly pop up if you work for an organization with a lot of traditional project management.

Imagine in the garden project if the product owner had just changed the phases into sprints. They might create a sprint deliverable like, "three pots filled with soil." Or even something worse like, "three pots ready to be seeded." The scrumMaster should recognize that these are not real sprints. The stakeholders doesn't have a finished deliverable at the end. A finished deliverable would be one pot filled with soil and fertilizer with seeds ready to go.

Planning Starts as Estimates

Projects are unique by definition. There will always be something in your project that is new. Maybe the team has never worked together. Maybe the project is working with new hardware or software. Either way, there will be a degree of uncertainty.

Usually the greater the uncertainty, the more trouble you'll have with traditional project planning. Traditional projects depend on a lot of up-front planning. If you have a lot of uncertainty, then you'll have a lot of guesses in your plan. If your guesses are wrong, it will create a lot of instability. Traditional projects work much smoother if your predictions are accurate.

Think about a typical project. The longer the project runs, the more you know about how to finish. That's usually because you're learning a lot from actually doing the work and you don't have to rely as much on predictions. Unfortunately, the more you work, the more expensive it is to make changes.

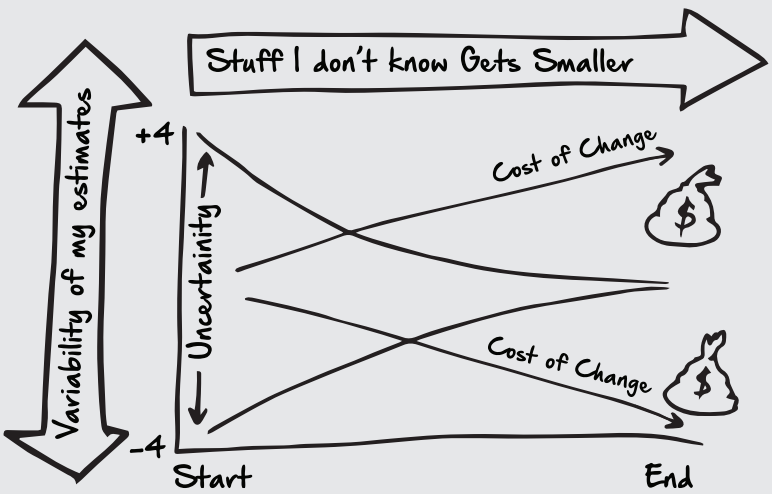
Think about it!

Think about a mobile application project. Let's say you wanted to create an application that can identify birds just by taking their picture. You call the application the Mobile Bird Finder.

There are several things you won't know about the application. For example, you won't know the number of bird pictures you'll need to make a match. You can make a pretty good guess, but you'll never really know until you build the application. Also, you won't know how long it will take for the application to match a picture to a bird. Again, you can guess, but you won't know until the application is near complete.

Now let's say that halfway through the application it becomes clear that the number of pictures you need to make a match is far too large. The only way to get the application to work is by splitting it into several regions. Now you're calling the application, Mobile Bird Finder – North American Edition."

Because the application is halfway finished, the cost of the change is now much greater. You'll have to rename all the files. The designers will probably have to redesign the application. There'll be a lot of rework. The cost would have been much less if the project had originally started with the application split by region.



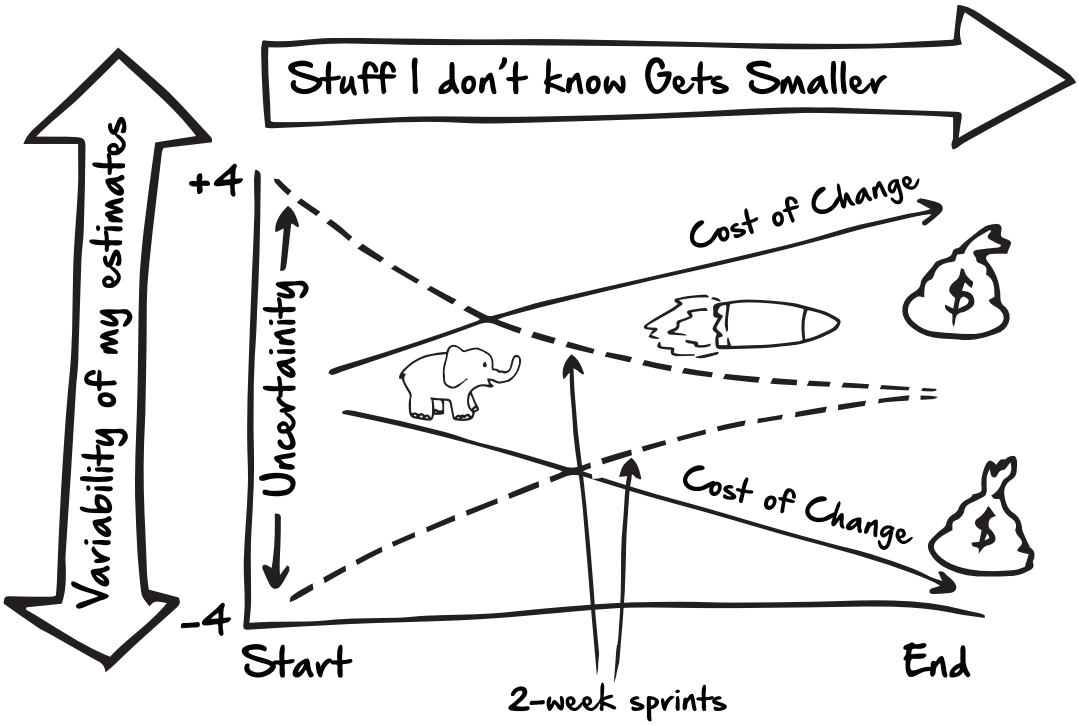
THE CONE OF UNCERTAINTY

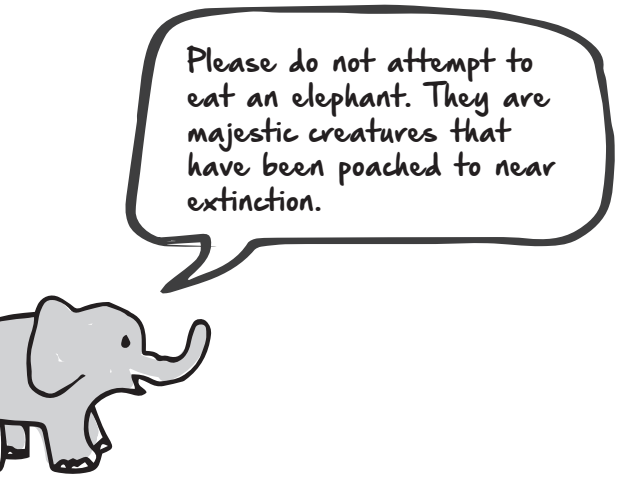
This is called the project's **cone of uncertainty**. It's when you're forced to make all the predictions at the time when you have the least information. If you look at this chart, you can see that projects are the most uncertain before they start.

This was true with our mobile bird-finder application. You didn't know how many pictures you'd need until after the work began.

As the project moves forward through time, you can see that it is more expensive to make changes. These are the two arrows pointing to the money to show these costs increasing over time. The **cone of uncertainty** narrows, but the cost of making changes increases.

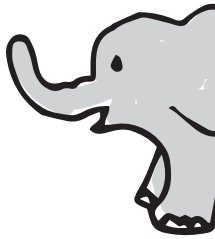
The cone of uncertainty is one of the big challenges with trying to design everything up front. This is the reason why project managers working on traditional projects will often complain about bad requirements. They're usually not happy with the project's predictions. Often the plan doesn't even identify the problems.





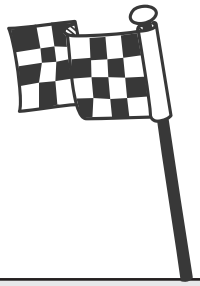
The agile approach to the cone of uncertainty is quite different. In many ways, the framework sees too much up-front planning as a waste of time. It's too difficult to make worthwhile predictions for the whole project and there's no reason to try.

For agile, it's like the joke about eating an elephant. The only way to eat an elephant is by taking one bite at a time. You shouldn't try to swallow everything up front in one big plan.Instead, agile breaks it down bit by bit over time. These are the iterations of the project, or the sprints.



The team will also start with the highest-value items first. It's not just about eating bites of elephant - it's also about eating the highest-value items first. If you're eating an elephant, you'll want to start with the tastiest parts. I've never eaten an elephant, but I'm guessing that would be the trunk.

The team should work like a heat-seeking missile. From the very beginning, the team will be working on the tastiest parts. Then they'll move through the project until they get to the tail.



Each sprint should have a stakeholder checkup at the finish line. The product owner will present the deliverable to the stakeholders to make sure they're building the right thing. In a sense, each finishing line is the team asking, "Am I building the right thing?" If the answer is yes, then they move on to the next sprint. The team will use this rhythm to complete the project

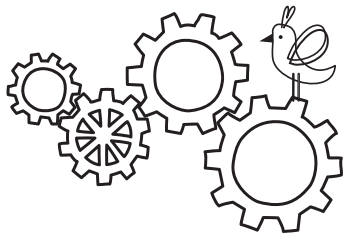
Field Notes

For our bird-finder application, the team would've created a small bit of the larger project. Perhaps for the first sprint they would have just created an application that took a picture of a bird. And then during the next sprint they would have an application that took a picture of a bird and matched it against two or three photographs.

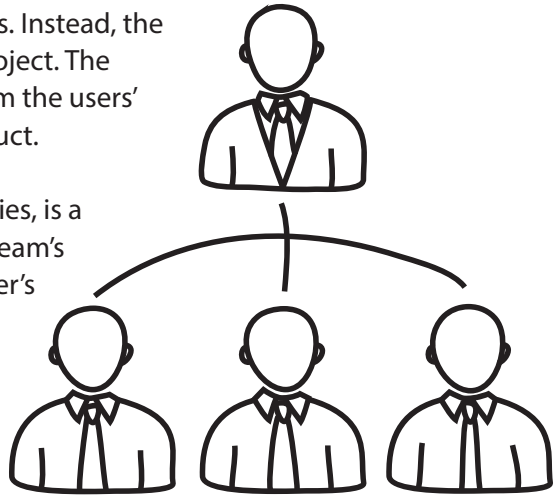
The project would continue that way, bit by bit, until the stakeholders agreed that all of the functionality was complete.

A traditional project will always be working on the whole elephant. The only way to divide up the work is by the amount of time and effort. The team will be working on the items they predict will take up the most time. Project managers often call this the long pole in the tent. It's the biggest piece of the elephant from head-to-tail.

In our bird application, it is likely that the team would be gathering up thousands of photographs to test out the matching feature. The more time the team works on gathering the photographs, the more difficult it will be to later make changes to the project. If nothing changes, that will be fine. If there is a change, then it usually means there will be a lot of rework.



Starting with User Roles



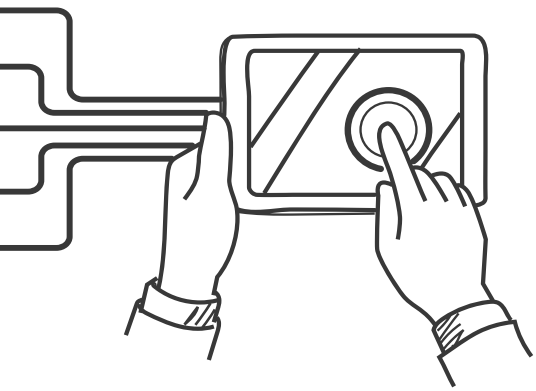
Agile projects don't use traditional project requirements. Instead, the product owner maintains a set of user stories for the project. The user story is a conversation vehicle. It's a short story from the users' perspectives about what they find valuable in the product.

The product owner, as the maintainer of these user stories, is a very important person on an agile project. They're the team's North Star. They're the light that represents the customer's value. They may know a little bit about what it takes to deliver the project, but they won't create the kind of detailed technical specifications you need for a traditional project. Instead, the product owner will always speak the customer's language.

This customer's language is much different from what you'd see in a traditional project. It focuses on the users' experiences and not on what goes into creating those experiences.

Let's go back to our ridesharing application. In a traditional project, you would create a plan to finish the website. In that plan, you might have milestones. Maybe after three months, the project would have a milestone to complete the server installation. You might call this milestone infrastructure complete.

Now you'd go back to your customer and say, "the infrastructure will be complete after three months." The customer who's sponsoring the website is probably not technical, so the customer would need to learn what infrastructure means. Then the customer would have to figure out how that helps with their website. At best, it would be a large learning curve if the customer wanted to prioritize the work. You usually wouldn't have a customer that says, "Well, why don't we use virtual machines?"



Let's take that same website and use our agile product owner. The product owner will also have limited technical knowledge. How can they prioritize when the infrastructure should be complete? Why do they need to know what the server does when all they want is a website?

Instead, the product owner will prioritize the work based on their view of the product. They don't view it as a technical achievement. The product owner will view it as a means to an end. The easiest way for a product owner to do this is to create user roles.

A user role is not a person. Instead, think of it as a bundle of user experiences.

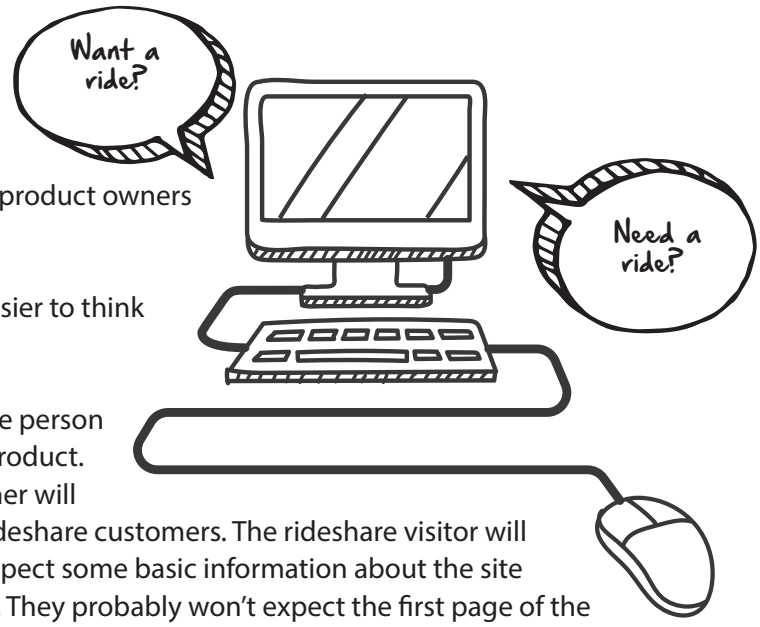
Field Notes

For our ridesharing website, there could be many user roles. There might be a user role called rideshare visitor. This is someone who lands on the site but isn't logged in. For a user who logs in, you could create another role called rideshare customer. You could then divide these into other roles, such as rideshare seeker or rideshare provider.

If you come from traditional project management, you might be tempted to confuse user roles with stakeholders. Stakeholders are often viewed as people. They could be customers or the nonprofit that runs the site.

Agile user roles are a bundle of user experiences. Someone might enter the website as a rideshare visitor. Then they'll register on the site and become a rideshare customer. Once they're a customer, they might decide to be a rideshare seeker. So in this case, you have one person who actually went through three distinct user roles. They had three different bundles of experiences based on what they were doing with the site.

Usually, product owners will create all the user roles for the project. It can be a real challenge for product owners to come up with these bundles. They have to really think through how someone will interact with the final product. They'll have to decide what makes a visitor different from a customer. Another decision will be whether to divide user roles into those who are seeking rides and those who are giving rides. These are the tough questions product owners will have to sort out early.

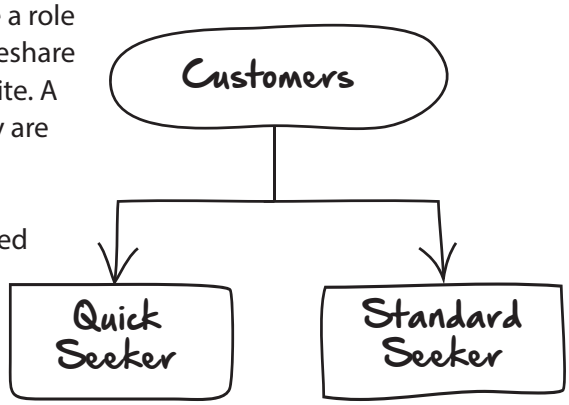


When creating these roles it's sometimes easier to think using their context, character and criteria.*

The **context** is the bundle of experiences the person will have based on where they are in your product. In the ridesharing website, the product owner will probably separate rideshare visitors from rideshare customers. The rideshare visitor will have a different context. They'll probably expect some basic information about the site and maybe a place to create a new account. They probably won't expect the first page of the website to be a login and password box.

*Based on work by Larry Constantine

The **character** is a little bit of insight into the experience a role might expect when interacting with your product. A rideshare visitor might be more curious about who runs the website. A rideshare seeker might be more curious about who they are connecting with for a rideshare.



Finally, product owners might create new user roles based on a person's criteria. They can create a role based on someone's expected outcome. There might be a quick-rideshare-seeker role that is separate from the regular-rideshare-seeker user role. The quick-rideshare-seeker role might have an immediate need to find a ride. For this person, the criteria is getting hooked up with a rideshare quickly and without too much difficulty.

If you're just starting to create user roles, it is easier to start broadly, and then create subgroups of experiences within those larger roles. With the rideshare site, the product owner could start with customers and then breakdown the role into different sub-roles. Try to use the context, character and criteria of the role to breakdown the bundle further.

Creating User Stories

I was traveling through a large airport and I didn't have much time so I jumped into a fast food restaurant. There was a big sign that welcomed me at the door. On the sign it said "supersize everything for a dollar."

While I waited in line I was wondering whether I should supersize my fries. I wasn't interested in supersizing my drink. That's always a bad idea before a long flight. The problem was I was a little bit confused about how to order. Would they supersize everything for a dollar? Then I would get a huge soda and fries. Or is it that I have the option to supersize anything on the menu? Then I could pick and choose what I wanted supersized and for each supersizing event, I'd pay a dollar.

When I got to the front of the line, I placed my order. Before I had a chance to clarify, the person asked me very quickly, "Would you like to supersize anything on your order?" Just a little snippet of conversation cleared up all my confusion. Anyway, it was too late. By that time, I had decided it was a bad idea to eat a pile of fries.

When planning a project, many people assume that when you write something down, it becomes much clearer. Project requirements strive to be written in clear direct language. What could be clearer than, "supersize everything for a dollar."

It's usually when you turn this clear language into work that you realize there's a lot of room for misinterpretation. Unfortunately, it's almost impossible to predict how each person will interpret your language. I'm sure hundreds, if not thousands, of people walk by that fast food sign with little confusion. Often, a quick conversation is the only way to make sure that everyone knows the work.

User stories are a great way to have conversations between developers and the product owner. That's usually a good thing. In software projects, developers need to spend time making sure they're developing the right thing. If you don't have these conversations, then you could end up with unnecessary rework.

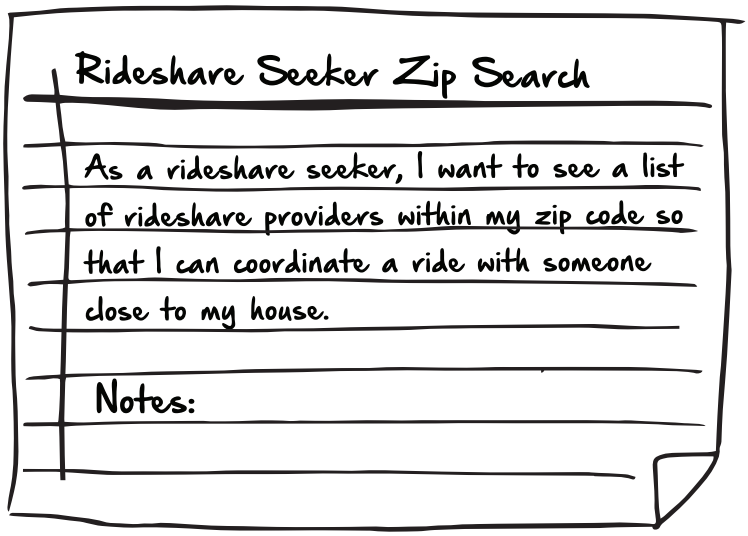
Creating user stories begins with the creation of the user role. This is the bundle of user experiences that a product owner defines before creating user stories. The following format is a common format for creating a user story.

As a <***user role***>, I <***want/need/can/etc.***> <***goal***> so that <***reason***>

Now the product owner take this bundle and attach them to some value.

Let's go back to our ridesharing web application. Say you've created a user role called rideshare seeker. Now you need to attach some business value to that role.

For this story, you might say "as a rideshare seeker, I want to see a list of rideshare providers within my zip code, so that I can coordinate a ride with someone close to my house."



This user story accomplishes two things. First, it uses the customer's language. The product owner doesn't need to know how the application is going to search for a zip code. There's no talk about how the website will work. The story is only about the what and the why. This will be much easier for the product owner to understand and prioritize.

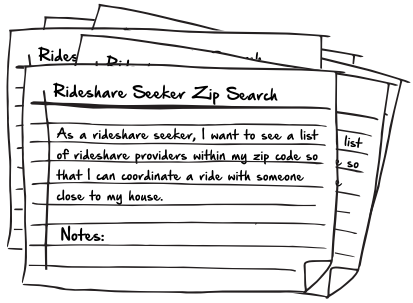
The story format is important because of what it doesn't say. It leaves the technical decisions up to the team. It's very important for the product owner to not swim in the developer's pool. They need to leave the how to the development team.

Second, the story is immediately linked to some business value. You could tell from the story that what the rideshare seeker really wants is to find rideshare providers near his or her house. Maybe there's a better way to do this? The product owner might not know about other options to track the user's location.

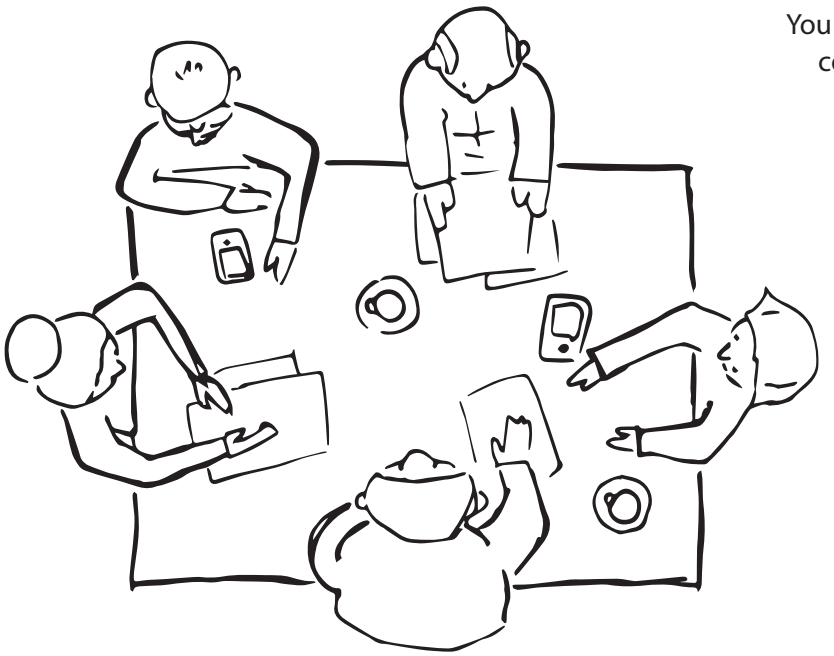
Think about it!

A good way to create user stories is to think about the three Cs. This comes from the world of extreme programming and stands for **card**, **conversation** and **confirmation**.

User stories are best recorded on a 3x5-inch index card. Many product owners are tempted to create a user-story spreadsheet, but if you try this, what you'll find is that it's more difficult to communicate about individual stories. You don't want to overwhelm the group with lists of information.



Instead, you want to have a group conversation. Many times when an agile team is starting out, they're very focused on the format of the user story. They make sure every story starts with the "as a user role" and ends with the "value goal." What they forget is that a user story is a means to an end. It's not about the format. It's about having a back-and-forth conversation between the developers and the product owner.



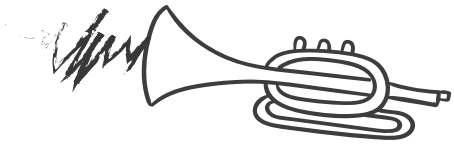
You will have a lot more natural conversations if you have a stack of cards in the middle of a large table. You'll be able to slide the card back and forth as the developers come up with new questions.

The final part is the acceptance criteria. This is the confirmation that everyone knows how to produce the required deliverable. This is typically written on the back of the card. In the language of extreme programming, this will be the "definition of done."

It's very important that the acceptance criteria closely match the user story on the front of the card. The acceptance criteria should be the result of a conversation with the development team and will go into much greater detail regarding how the developers will deliver the requirements to address the user story.

The acceptance criteria are a good way to keep the user story from being too convoluted. You can add a lot of details to the acceptance criteria without putting it on the front of the card. For this story, you could sort out the details like whether you want to use five-digit or nine-digit zip codes. This would add too much bulk to the user story, but it's fine to add this to the acceptance criteria.

Writing Effective Stories



When you start having these conversations about your user stories, you will probably realize that they are not very good. It will be like picking up a trumpet for the first time. There'll be a horrible screeching sound instead of music. But that's okay. Creating user stories is a skill. And like any skill, you usually get better over time. The good news is that even small projects require dozens, if not hundreds, of user stories. So you'll get a lot of practice.

The real challenge will be figuring out just why your user stories are so awful. When you figure that out, then you can start to make them better. Thankfully, there is a good way to figure out the challenges you will face writing user stories. You should use the acronym **INVEST**. This is a list of common challenges you will have to deal with when creating your user stories.

Independent: It's important to remember that independent doesn't mean your stories are functionally independent. It means they are independently valued.

Let's look at the rideshare application again. The product owner could create two stories: one story to create a list of the closest rideshare seekers and another to sort the list closest to furthest.

These two stories are functionally dependent on each other. The team can't sort the list without first displaying the result. But they can be independently valued. The product owner might not place a very high value on sorting the list. Maybe the product owner wants the first list to be based on the rideshare provider's feedback and not distance.

The product owner also might not be interested in sorting the list until the next version of the application. It's important for the product owner to try and create user stories with these small bits of independent value.

Negotiable: You want your development team to negotiate with the product owner. The developers should be the ones to decide the best way to deliver the value in the story.

In a traditional project, the developers would get a list of detailed requirements. User stores are different. They are small narratives about customer value, and they are written from the customer's perspective. The product owner shouldn't create user stories that focus too much on how to deliver their request.

Danger Zone!

For the rideshare application, you wouldn't want to create a user story that says, "In the same database query, return a list with a metric that enables sorting." A user story like this isn't focused on what the customer wants. Most customers don't care how you sort the list; they just want it sorted. You'll get these directive-style user stories when the product owner has too much technical knowledge.

Valuable: Customer value is the most important part of the user story. Without it, the product owner can't prioritize the backlog and there's no way to complete the story.

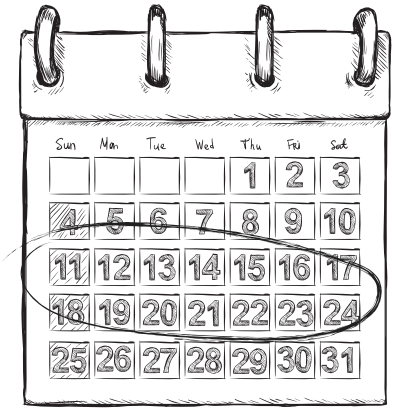
It's very common for product owners to struggle with value when they've come from traditional project management. What often happens is that they'll see user stories as rewritten requirements on index cards. They may even create a requirements document and then break it down into users' stories.

Typically when this happens, you will get user stories that have little or no customer value. Something like, "as a rideshare seeker, I want the search method tested so that I know it works." That's just a restatement of a testing task in a traditional waterfall process.

These challenges also come up when product owners neglect their responsibility to create good stories. They'll ask the developers to create their own stories for the project. Then the developers will just create index cards that break down the project into functional chunks.

Estimable: When developers can't estimate the work required to address the user story, it usually means it's too big or not clearly described. Usually, the smaller the user story, the clearer the description. If you're the product owner and you see the developers struggling to provide an estimate for a user story, then you may need to go back and break it down further.

Small: A user story should be delivered within a two-week sprint. That means the size is closely related to whether the team can make estimates about the user story. When the story is too big for developers, they can't predict everything that will go into delivering the value. In general, the smaller the story, the easier it is for the team to understand everything it entails. That makes it much easier for the team to estimate the tasks required.



Testable: Think about a user story like, "as a rideshare seeker, I want to see the list of rideshare providers quickly so I can share my ride." This might be a true statement of customer value. The rideshare seeker will certainly want the application to work quickly and easily.

Danger Zone!

The problem with this story is that it's not testable. What is quick for one customer might be slow for another. How do you test the concept quickly? Will the product owner be the one who decides what is quick?

Watch out for these adjectives and adverbs in your user stories. Words like simply, clean, easily, fast or nice. These will only make it more difficult to estimate.

Instead, focus on customer value, and if necessary, use the acceptance criteria on the back of the card. On the back of the card, the product owner might put, "the list will come back with ten records at a time and so the rideshare seeker can scroll through them in batches."

Grouping with Themes or Epics

A theme or epic is a way to organize stories into groups. It's inevitable that most user stories start their lives as epics. The product owner usually creates a large value statement instead of several smaller stories. That's ok. That's just the way that most people think. Most people think about what they want. Then later they break it down into valuable little details.

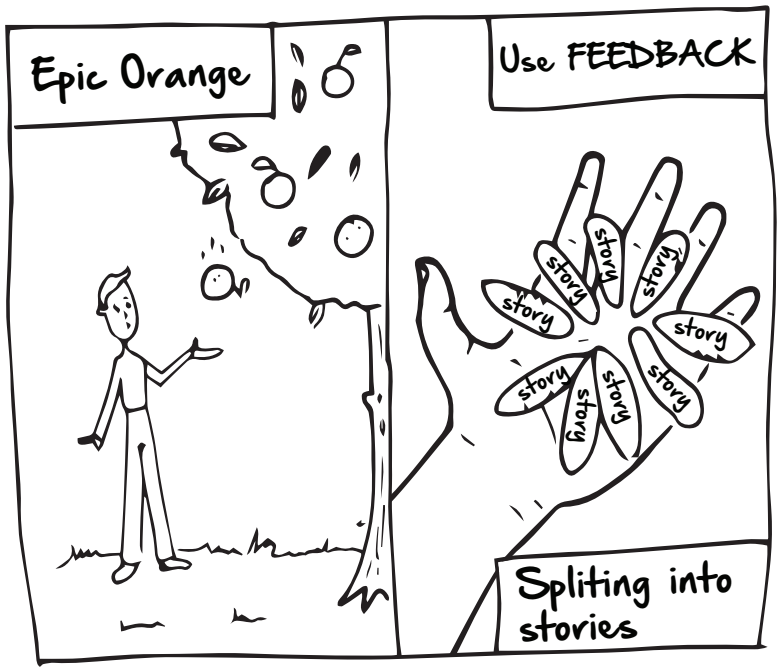
It's important to remember that these are large vague epics and not actionable stories. They are naturally grouped together and need to be split up. It's like being given a whole orange to eat. Once you have the orange, you need to go through the process of splitting it into edible segments.

Epic splitting isn't an easy task. There are many different ways to split your epics into more stories, and

splitting stories often leads to further splitting. This means that you could end up with completely different stories depending on how you split them.

It's like any ordered division. Think about splitting a bag of candy by weight and then by color. You'd end up with different groups than if you split the candy by color and then by weight.

That's why it's important to know some of the common ways to split your stories. That way you can have some control over the final grouping, and you can split the epic using the same criteria each time.



There are several common ways to split your stories. These are the eight most common that I've seen on agile projects. The best way to remember these groups is by thinking of the acronym **FEEDBACK**.

Let's start with a user story we might use for our rideshare application.

"As a rideshare seeker, I want to see a list of rideshare providers within my zip code so that I can coordinate a ride with someone close to my house."

This user story could easily be an epic. Let's try to break it down into groups using our **FEEDBACK** splitter.

F

Flow. This is how the story might step through the application's workflow. You can split this epic into a few more stories.

"As a rideshare seeker, I want to see a list of rideshare providers within my zip code so that I can coordinate a ride with someone close to my house when I'm on my smartphone." And "from my GPS watch."

Effort. You might decide to break the epic down based on the developer's level of effort.

"As a rideshare seeker, I want to see a list of highly-rated rideshare providers within my zip code so that I can coordinate a ride with someone close to my house." And "highly rated rideshare providers within my zip code..."

The developers will need to spend most of their time creating the list of rideshare providers based on the zip code. It should be trivial to add more selective criteria like "highly rated" and "superstar" providers. But it will still give the product owner an opportunity to prioritize the work.

Entry. Sometimes, the team might want to break down the epic by how the customer enters the data.

"As a rideshare seeker, I want to see a sorted list of rideshare providers within my zip code so that I can coordinate a ride with someone close to my house." And "a reverse-sorted list of rideshare providers..."

This is how your customer will see the data. You can break it down based on their entry and view.

Data Operations. You may want to break up your epic based on common data operations like read, update and delete.

"As a rideshare seeker, I want to read a list of rideshare providers within my zip code so that I can coordinate a ride with someone close to my house." And "update that list with my information."

Business Rules. Sometimes there's a lot of complex value in the epic. When that happens, you may want to break down the epic into business rules.

"As a rideshare seeker, I want to see a list of rideshare providers within my zip code and 4 digit extension so that I can coordinate a ride with someone close to my house." And "within my GPS coordinate range so that I can..."

Alternatives. Some epics can easily be broken into stories with alternative criteria. This story uses a zip code. You can create many stories based on alternatives.

"As a rideshare seeker, I want to see a list of rideshare providers within my street address so that I can coordinate a ride with someone close to my house." And "within my city," or "within my county."

Complexity. Many epics are just the beginning. When product owners start thinking about user stories, they may find greater and greater value. User stories can be like a diamond mine with a few scattered pieces off the top that hint at the treasure below. When that happens, the epic is just a simple start. Then you can break it down into stories with increasing complexity.

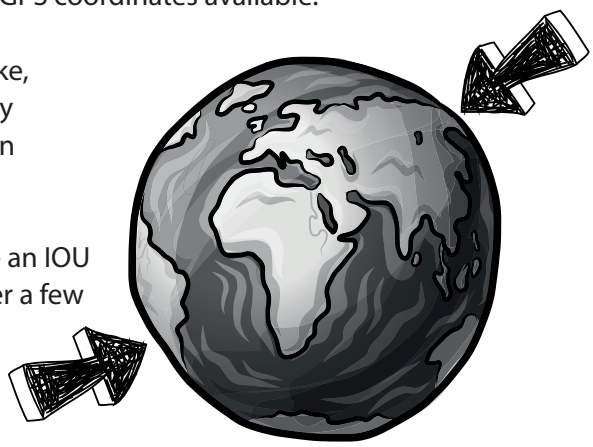
"As a rideshare seeker, I want to see a list of all new rideshares within my zip code so that I can coordinate a ride with someone close to my house." And "a list of all recommended rideshares within my zip code..."

Knowledge. Sometimes the product owner will present the epic and the team will need more knowledge. They have to research the epic to break it down into stories. When this happens, the developers will create spikes. These story spikes have secondhand value since they answer questions needed to deliver the main user story.

In this epic, the real value is finding rideshare providers that live or work near the customer. There might be a better way to deliver this value. Maybe the modern web browsers can deliver the customer's location. A smartphone user might have their GPS coordinates available.

When this happens, the developers might create spikes like, "investigate location services for most web browsers." They might create another spike for "investigate GPS services on smartphones."

Story spikes aren't stories, but they lead to stories. They're an IOU for the product owner. The developers just need to answer a few questions.

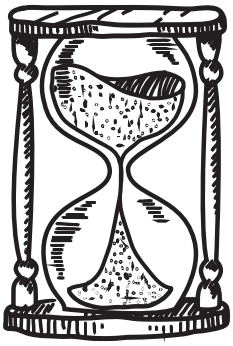


Using Relative Estimation

Early humans had a pretty tough time. They must've lived in near constant fear of being eaten. They were also in danger of not getting enough to eat. Throughout their life they went through a process of relative estimating. If it's big like a bear, I need to run from it. If it's small like a fish, I might be able to eat it.

It wasn't until much later that humans developed metrics. If it's taller than a meter, I should probably run from it. If it's less than five pounds, I might be able to eat it.

Now let's fast-forward thousands of years to an office park. All the project planning is in hours, days and weeks. In these meetings, you'll go around the room and ask for time estimates. Often the metric is more precise than the team's ability to understand the work. So they'll end up giving you a range. They'll say, "It could take one to three days."

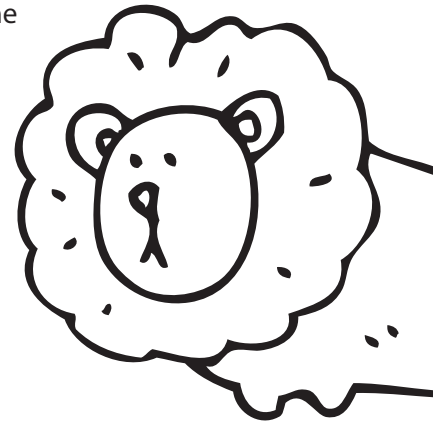


The project manager gathers up all these ranges and creates one big estimate. Maybe they've decided it can take anywhere from four to six weeks. Then they add project buffers just to be sure. The number might seem precise because it's in weeks or months. But it's still just a range in disguise. What's worse is that it took hours of meetings to come up with these questionable results.

Agile doesn't fix how bad teams are at estimating. Instead, the team spends much less time on this activity. From an agile perspective, not doing something is the fastest way to getting it done.

An agile team goes back to the technique used by our early ancestors. The team won't think in hours, days or weeks. They'll think in relative sizing. Is this user story bigger than a bear? Maybe it's smaller than a fish.

Relative estimating compares what you don't know against what you do know. You might not be able to guess how much a truck weighs. But if you saw the truck you can probably guess how many cars equal a truck. You might not know how much a lion weighs. But you can guess that it may be three or four dogs.

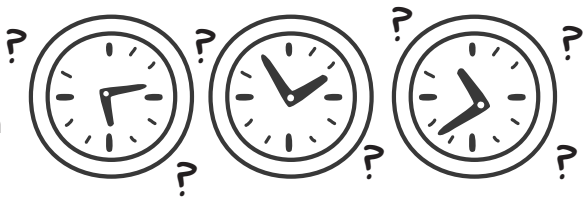


Now dogs, lions, trucks and cars all have different weights. This estimation is not designed to be precise. But that doesn't mean it's useless. Instead, it gives you a starting point. A way to start the discussion on what it takes to deliver your stories.

Agile teams use this relative estimating for two reasons. The first reason is that it takes away the false comfort of precision. The team is accepting the fact that the estimates will be imprecise. So they use a simple method to make a best guess. Everyone knows it's a guess and the team's not going to get hammered for getting it wrong.

Assigning a date to guesses always makes them seem more concrete. It forces developers into a world of precision, even when they don't have enough information to be precise. That's when you find yourself in a Bizarro World where developers take a half hour to decide if something will take five or six hours.

Relative estimating gives these teams freedom to abandon this false precision. It's a way to say, "It's okay that you don't know. Now it's time to guess." That way we can start talking about what it takes to deliver the story.



The second reason agile uses relative estimating is that it keeps the team from confusing estimates from commitments.

Think about your drive home from work. Some days, it may take you up to an hour because of some construction. Other days, it seems like you might make record time and you'll be home in ten minutes. If there's a snowstorm, then who knows when you'll get home? Despite all these unknowns, it still usually only takes you about twenty minutes.

Now imagine that you needed to get home for a deadline. Maybe your dryer is broken and the repairman will be there at six. Or say you needed to be home for an early dinner.

In either case, you would probably commit to an hour to drive home. That way you can meet your deadline and might even be home early. You probably wouldn't commit to twenty minutes. That's a best-case scenario. If there's any traffic or any bad weather, you'll be sure to miss dinner.

Now imagine you needed to make an estimate. Let's say that a coworker lives near your house and asks you how long it takes for you to get to work. For that, you would probably say it usually takes you about twenty minutes. You're giving an honest estimate. You're not worried that your coworker will storm into your office if it takes more than twenty minutes.

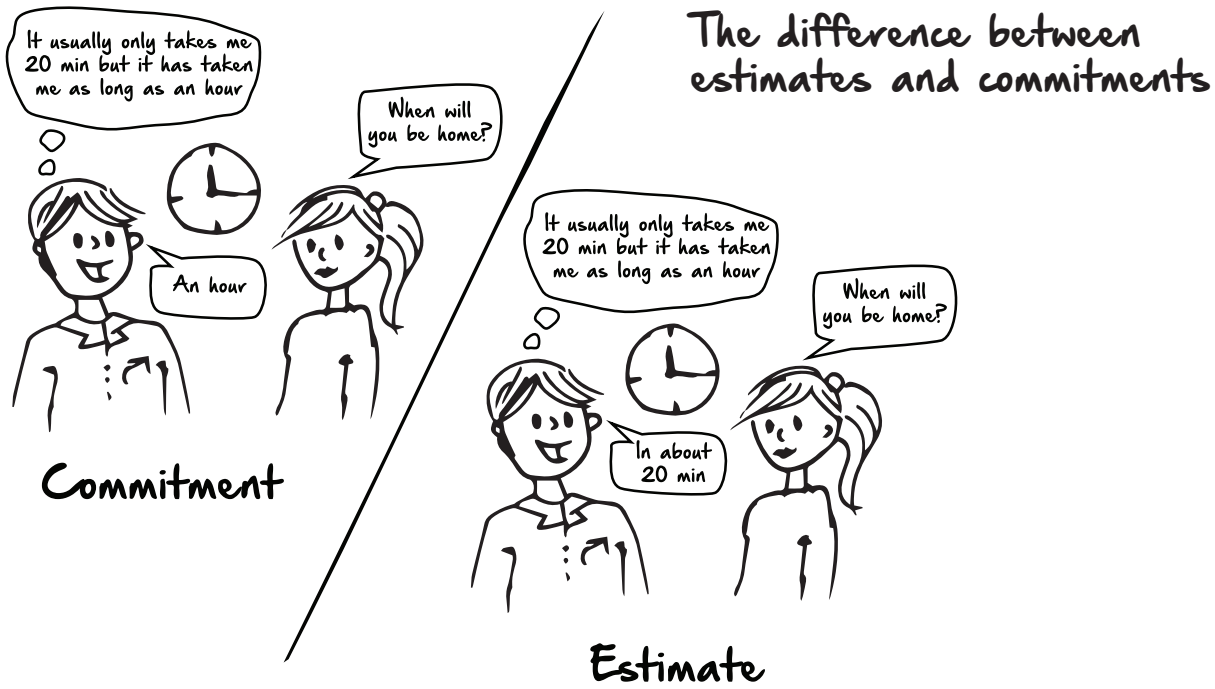
Field Notes

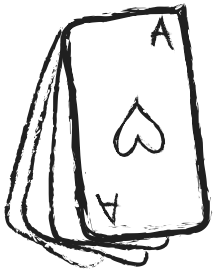
An estimate is the useful information that you might give a **coworker**. A commitment is something that you usually give to a **supervisor**. An estimate is a best guess. A commitment is often a worst-case scenario.

That's why for agile planning you want estimates and not commitments. Let's say you needed to estimate how many hours you'd be telecommuting each week. You would probably guess it would be twenty minutes each way which would be forty minutes a day. Then you would multiply that by five workdays and have an estimate of a little over three hours.

The commitment would be much higher than the estimate. If your supervisor asked, you might say it took you an hour each way. That's two hours a day and ten hours a week.

Even with this simple task of driving to work, you have a range of three to ten hours a week. That doesn't really help anyone with planning.





Playing Planning Poker

Planning poker helps the team get the best estimates with less-than-perfect information. The game assumes that everyone on the team is an expert. As experts, not everyone will agree on how long it takes to meet each user story. There is wisdom in the whole team. Planning poker gives everyone a voice.

One developer might decide the story is complex while another developer might think that the story will be finished before starting lunch. There's no way to tell which developer is right until after the work begins.

In agile, you're not estimating for individuals; you're estimating work for the entire team. Many different developers on the team will work on the same story. An agile team should be cross-functional and filled with many experts. That's why it's important that each team member makes estimates. Everyone should have a say when you're all working together to deliver the work.

Planning poker is not just about creating an estimate. In many ways, the estimate is a byproduct of the game. The activity is really about combating groupthink. There should be a consensus on what's nvolved in delivering each story.

Groupthink is the way that people tend to agree with the most popular idea. The loudest voices will drown out any disagreement. Without any disagreement, the group then assumes that they must be right.

You may have been in a groupthink meeting. The few people who are not on board are easily dismissed or ignored. Then the group endorses the most popular idea even if it's wrong. This is especially true if one of the opinions comes from a HIPPO.

Planning poker tries to combat this tendency. Each team member makes an estimate before they're told what to think.
Planning poker tries to combat this tendency. Each team member makes an estimate before they're told what to think.

Planning poker is simple to play and accurate enough for agile planning. Each team member will make a relative estimate for all of the stories. They will use story points as an estimate of the relative size. Remember that this is a relative estimate. If you're the scrumMaster for the team, try to make sure that everyone thinks in story points and not hours, days or weeks.

Each attendee will have a deck of planning poker cards. They will have an exponential number sequence. Something like 0, ½, 1, 2, 4, 8, and 16. Many cards use the Fibonacci sequence of 1, 2, 3, 5, 8 and 13. These are the points for each story.

The Fibonacci sequence often works better because it gives more options at the bottom. It's easier to estimate with 1, 2, 3, and 5 than 0, ½, 1, or 2. The teens are almost always a sign that a team member doesn't understand what it takes to deliver the story.

There is also a coffee card and a question mark card. The coffee card is a request for the whole team to take a break. Sometimes, the debate is too unproductive. Other times, people just stop thinking clearly.

The question mark is a way for someone to skip the first round of estimates. It usually means they want to hear the first round and then later add their estimate. This card should be used sparingly. It is best used when someone needs a little more information from the rest of the team to understand the story.

To begin poker planning, the team identifies one of the smaller stories and assigns it a value of 2. This will be the estimation baseline. The baseline will be the first half of the relative estimate.

Think about it!

Think about our rideshare application. Let's say the story for the baseline estimate is "As a rideshare seeker I want a checkbox to save my user name so I don't have to retype it every time I log into the site."

That would be the smallest story to deliver in the sprint. All the later stories would be a relative factor of that story. The team can think of it as how many checkbox stories are there in each of the estimates? A larger story might be four times as large as the checkbox story.

The scrumMaster will pull out the other stories in no particular order. The product owner also needs to be there to answer any questions the team might have about the story.

The story is read to the team either by the scrumMaster, the product owner or one of the developers. It's important that the team immediately start estimating. If you're the scrumMaster for the team, make sure that there is no discussion about the story before the first estimate. Any discussion can lead to groupthink. You could lose ideas before you even begin playing.

Each team member selects a card for their estimate and places the card face down. This estimate usually represents a number that takes into account time, risk, complexity and any other relevant factors.

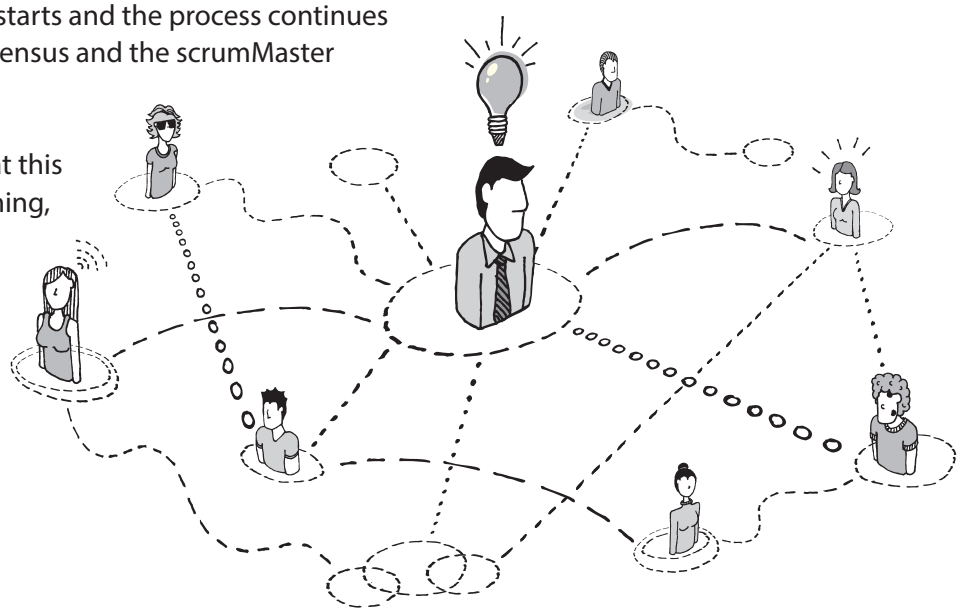
When everybody is ready with an estimate, all cards are flipped over simultaneously. If there is consensus on the number, then the size is recorded and the team moves to the next story.

In the very likely event that the estimates differ, the high and low estimators defend their estimates to the rest of the team. The scrumMaster might ask the person with the highest estimate to explain why the story is so complex. They might also ask the person with the lowest assessment to explain why the story is so easy.

The group briefly debates with other members who have different estimates.

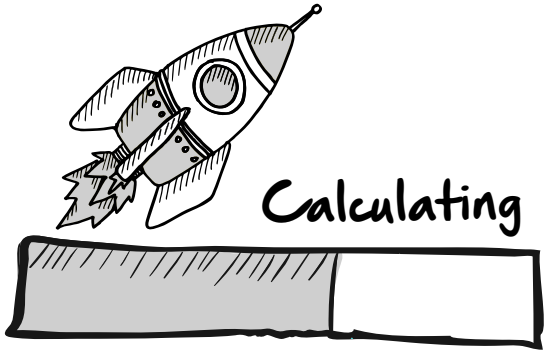
A new round of estimation starts and the process continues until the team reaches consensus and the scrumMaster records the estimate.

The team will want to repeat this for each story. In the beginning, the team might need a lot of guidance from the scrumMaster. Over time, the scrumMaster should be able to sit in the background and simply record the final estimate.



Calculating Your Velocity

After points have been assigned to the user stories, the team needs to calculate the team's velocity. Velocity is the number of story points a team can complete in a two-week sprint.



To calculate the team's velocity, the team will look at their own work history to determine their sustainable pace. They'll see how much they have worked in the past, and that will be how much work they commit to in the future.

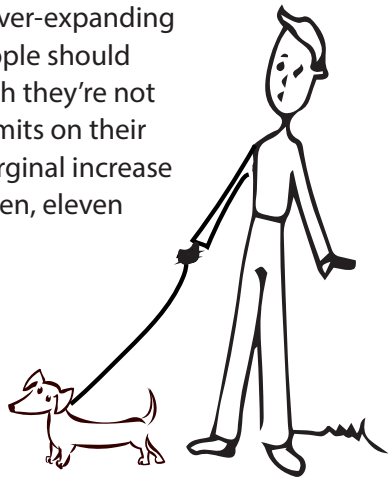
Field Notes

In 1914, Henry Ford noticed that his employees worked best when limited to eight hours a day. After eight hours, their productivity began to slow down. That's why the Ford Motor Company established the forty-hour workweek. The change increased their productivity and many companies soon followed. That's why the forty-hour week became standard.

In the 1980s, fewer people worked in manufacturing. They became desk workers and computer programmers. As a result, many employers felt that the forty-hour workweek was outdated. It only made sense in manufacturing. Working a seventy-hour week became a badge of honor for desk-working go-getters.

Determining a team's velocity is not simply choosing how much work needs to be done, it also needs to take into account productivity and the number of hours in a workweek.

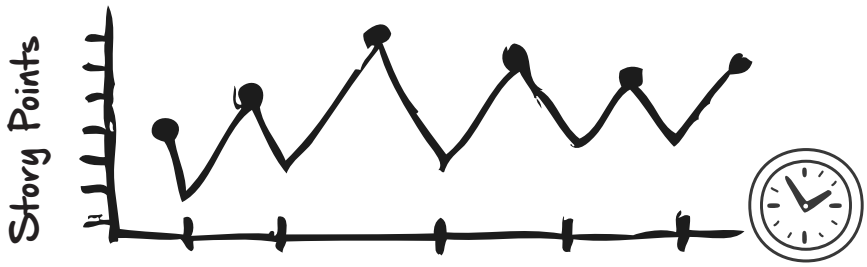
The agile framework pushes back on the ever-expanding workweek and promotes the idea that people should go home at a reasonable hour. Even though they're not building cars, developers have the same limits on their productivity. The team will have only a marginal increase in completed stories if they start working ten, eleven or twelve hours a day.



They're much better off going home, then starting fresh the next day. Often developers will think more clearly when they are driving home or are walking the dog.

An agile project should have a challenging pace but there shouldn't be any project crashes. A crash is when a project manager adds people or time to the project when it is clear that the team is behind schedule. The existing team will have to put in long hours or a group of new people will need to be brought up to speed. The team shouldn't have this flurry of activity at the end of the sprint. Agile projects should run like marathons. They should have a rigorous but consistent work pace.The team should have a predictable and practical workday.

The team will use the story points assigned during the planning poker sessions as a way to manage this sustainable pace. The product backlog will have all the user stories for the project. Most of these stories will have been given a relative estimate.



Typically, the scrumMaster will calculate the team's velocity. They'll create the number based on the story points delivered in each sprint. Let's say at the end of the first sprint, the team completed six stories. Four of these stories were estimated at two points each and one story was estimated at three points. The last story was estimated at five points. That means that at the end of sprint, the team completed sixteen story points.

Now the team knows that they have completed sixteen story points in their first sprint. Let's say at the end of the second sprint, they completed seventeen points, and at the end of the third sprint, they completed fifteen story points.

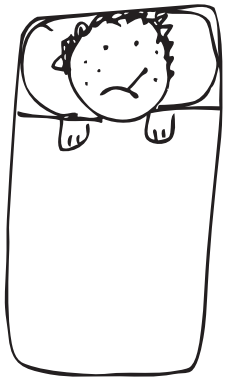
Now the scrumMaster has enough information to make a rough guess at the team's velocity. They'll average out the first three sprints. In this case, the average for the first three sprints is sixteen story points. That will be the team's velocity.

Velocity is a rolling average. That means that the velocity may increase or decrease depending on what happens with the team.

It's usually bad practice to add developers after the work has begun, but if you add a few developers, the velocity might increase. Let's say the team added a developer during sprint four. With the new help, the team was able to complete twenty-four story points.

Then the average for the four sprints would be eighteen story points. The team's velocity will go up by two points.

On the flipside, the velocity might also decrease. Let's say sprint four was in the middle of flu season. Too many people took time off. The team only delivered twelve story points at the end of the sprint. The team's velocity would slightly decrease to fifteen.



One of the challenges with velocity is that it is often misused. Senior managers will sometimes push the team to increase their velocity. They think that means the team is being more productive. But remember that story points are a relative estimate. The team assigns them in the planning poker session.

Field Notes

I once worked for an organization where a senior manager was pushing the team to "achieve greater velocity." They thought that increasing velocity meant the team was finishing more work. The team started out with a velocity of fifteen. Then the manager pushed the team to increase the velocity to eighteen, then to twenty-five and finally to thirty.

The manager was delighted to see the team reaching these levels of hyper-productivity. The team was twice as productive as when they started.

In reality, the team was just reacting to the positive reinforcement. In planning poker, the team started estimating each story at two or three points. After the manager started to push the team, they started estimating the stories at three, five, and eight points.

The team wasn't really completing more work. They were just estimating the work with higher values. The five stories they completed in sprint one may have been worth fifteen points. After the manager started pushing for an increased velocity, the same five stories would have been estimated to be worth twenty-five points.

It's important to keep this in mind when estimating velocity. These numbers are really created by the team for the team. When anyone outside the team tries to alter the numbers, they become much less useful.

Planning Your Sprints

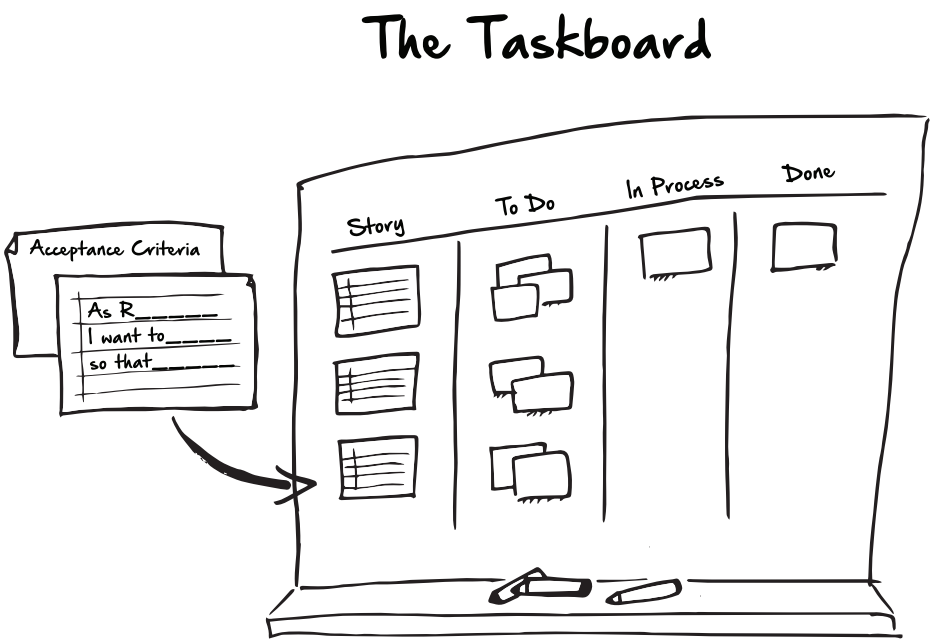
Sprint planning is when the team takes stories out of the product backlog and puts them into the sprint. Some teams will create a separate sprint backlog. This backlog will only have the stories to be completed in that sprint.

Other teams will take stories from the backlog and place them on a large task board. This task board is usually a whiteboard with four columns. From left to right, the columns will be labeled stories, to do, doing and done. The team will pull the stories from the backlog and place them into the first column.

It's more common for a team to just use the task board, although, a sprint backlog works well if you want to keep track of what the team does over time as well.

Sprint planning involves the entire team and is usually time-boxed to four hours. Time-boxed simply means that a strict time limit is placed on the task. In this case, the team has four hours to complete the sprint planning.

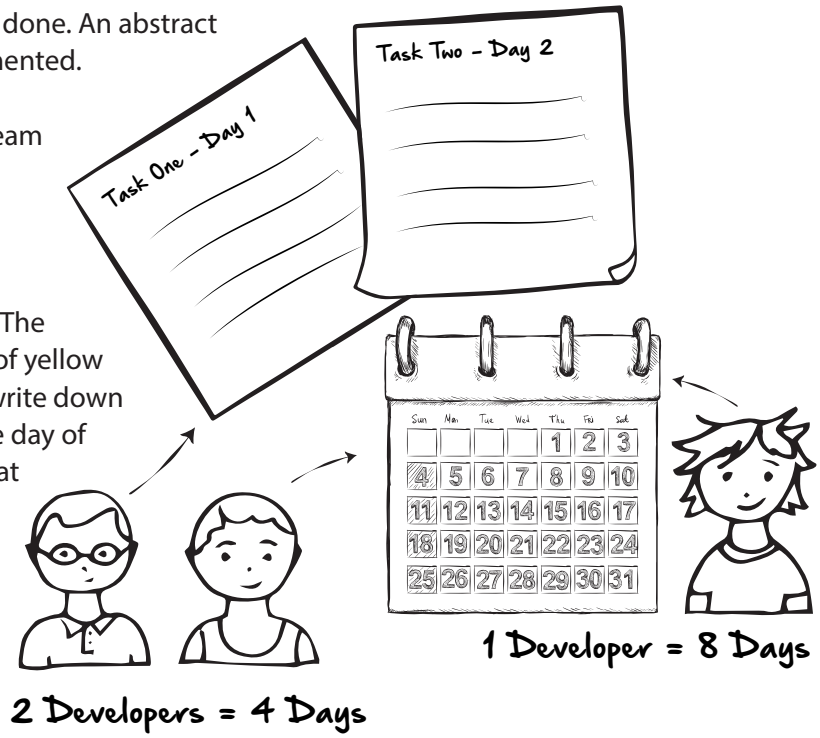
The product owner begins by going through the product backlog and presenting what they want done in the sprint. The development team decides how much of what the product owner wants they can deliver. It should be a back-and-forth between what the product owner wants done and what can be done.



Often the details about the stories get resolved in sprint planning. Up until now, the stories have only been considered in the abstract. The product owner created the user stories. Maybe they were organized into large groups like epics. Then the development team estimated the work required to complete the user stories with a round of planning poker. This was all done at a very high level.

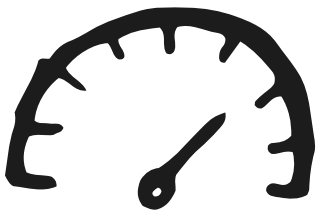
Now the team is very interested in what it takes to deliver the stories—what actually needs to be done. An abstract story suddenly needs to be implemented.

Once the product owner and the team have agreed on a group of stories to be delivered in this sprint, the development team will start to task out the story. Usually the team will be sitting around a large table. The scrumMaster will provide a bunch of yellow sticky notes for the developers to write down their tasks. Each task should be one day of work per developer. That means that if the story has eight tasks, it will take eight developer days. That might mean four developers for two days, or one developer for eight days.



You want to keep the task limited to one day so that the team can track their work easily. You don't want one sticky to represent one day of work and another to represent eight days of work.

Once the team decides what stories to deliver, they'll place them top to bottom under the stories column on the task board. Then next to each of the story cards, they'll place the tasks that are required to complete the story. All the tasks should be recorded on yellow sticky notes and placed in the to do column.



There are two ways for the team to decide how many stories they should deliver in the sprint. The first way is to deliver based on the team's intuition. The second way is to deliver based on the team's velocity. As mentioned earlier, velocity is how many stories the team has historically completed over time.

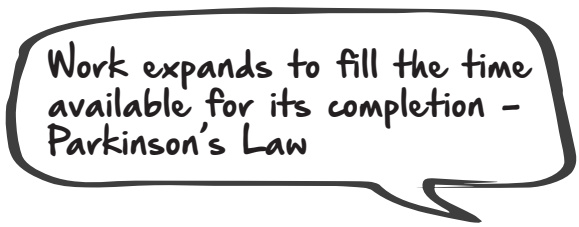
There are advantages and challenges to both methods of deciding on the number of tasks that will be addressed.

Remember, the team's velocity is a rolling average. For example, let's say on average you received a hundred e-mail messages a week. Does that mean you should plan each week to answer a hundred e-mail messages? You probably shouldn't. One week you'll have three hundred e-mail messages. Other weeks you may only have fifty.

For long-term planning, a hundred e-mail messages a week are fine. You can probably guess how much storage space you'll need on your computer, but for week-to-week or day-to-day planning, you may run into problems.

That's the same challenge you'll run into when planning sprints using the team's velocity. The developers will see the velocity as the team's capacity. Even though there are times when the team thinks it will deliver more, they won't commit to more.

There is also a danger that the team will make their velocity self-fulfilling. Let's say after four sprints, the team has a velocity of fifteen. From then on, the team uses fifteen to plan the sprints. Over time, fifteen solidifies as the team's only velocity.



On the flipside, it can also be dangerous to use the team's intuition. They might be wrong about what they can finish. When the team can't complete a story, it goes back into the backlog. That may be fine if it's just a few stories, but it gets to be a problem when there are many incomplete stories over several sprints.

Often the team will spend valuable time on the stories even if they're incomplete. That means a lot of half-completed stories get dumped back into the backlog. This could create a lot of confusion for the product owner. It also makes it more difficult for the team to predict their delivery and maintain a sustainable pace.

To deal with these trade-offs, it's usually better to mix the two approaches. If you're the scrumMaster for the team, you should start out having the team fill the sprint using their intuition. Then when you see their velocity leveling out, you can think about switching over to using their velocity. It usually takes at least six months for the team to have a sustainable velocity.

The advantage to mixing the approaches is that you get a more accurate velocity over time. The disadvantage is that you might have a bumpy six months before the team settles on a consistent number.