Agile Planning

Agile planning is a project **planning** method that estimates work using self-contained work units called iterations or sprints. ... **Agile planning** defines which items are done in each sprint, and creates a repeatable **process**, to help teams learn how much they can achieve.

Agile Planning

Agile Planning

So my next blog is on Agile planning, follow my course to learn about this new planning method. Over the next couple of weeks we will be looking at what goes into to making a project Agile.

Planning with Users Stories

Agile projects are adaptive and fast moving. To be successful with agile, you need to follow some simple planning steps to keep the project on track. There are only a few simple practices, but it's important to understand these practices and follow them correctly. We'll start this course by introducing the agile sprints. Working in sprints is a significant departure from traditional projects.

Next, we'll see how to create an agile user story. These stories are the centrepiece of agile planning. I'll demonstrate how to estimate each story and then roll them up into a planned sprint.

Finally, we'll take these estimates and demonstrate how you calculate your team's velocity. That way, you can create a release schedule for your final product delivery. This course is the second in this Agile At Work series.

Whether you're a project manager, a software developer, or a senior manager, this series is designed to help you get greater agility from your team, so let's get started planning your project with agile user stories.

Making Agile Predictable

Imagine you're an executive and you've convinced your supervisors that they need a new software product by the end of the year. Then you're given a large sum of money to deliver that project. You'll get a plan from the development team about how they intend to deliver the software.

The plan shows that by the end of the year you'll have working software within that budget. What often happens with these plans is that they don't turn out to be reliable. The software takes more time, more money, or both. Milestones disappear and commitments go unmet. This may not happen with every project, but it happens enough to be a problem.

If the development doesn't go according to plan, they have to make a couple of tough decisions. Should they double down on the project and add more money or should they accept a business quarter of wasted effort and abandon the project? Either one of those is a tough call. Many executives realize that planning and predictability isn't the same thing.

They could plan on getting something really valuable at the end of the year, but if they depend on that plan, then they could end up in danger. Sometimes it's better just to get something that's completed a little bit over time.

Having a detailed plan is great, but it's only valuable if it's accurate. When it's inaccurate, it can actually cause an executive a lot of headaches. It means that they've committed to something that they can't deliver. Sometimes it's better to be uncertain than to be certain about the wrong thing.

So, the predictability in agile can actually be a big motivator for executives. Imagine if the executive had sponsored an agile team for the software. They wouldn't get a detailed plan from the team. Instead, they'd just establish a deadline.

The product owner would coordinate with the executive team and create the software. The team may run into the same challenges. There'll be changes; the team might rely on software vendors that don't deliver. Agile doesn't promise to eliminate all those unknown blockers that can crush a project. Instead, an agile team builds up working software a little bit over time.

The project might not deliver everything, but it will deliver something, and that something will work. That makes a big difference for the executive at the end of the year. They'll have working software. They won't just get an unfulfilled promise. The agile software might not have all the features and functions, but they won't have to settle for an IOU. It's important to keep these pain points in mind when thinking about your executive sponsor. You should always use the language of predictability and delivery. This is especially true if you're trying to convince an executive to sponsor your agile project.

Incremental Planning Methods

Iteration

" A general Agile term for a fixed duration of time during which development takes place".

Sprint

"Scrum-specific term for fixed iterations of developing a usable and potentially releasable product".

The agile framework uses short durations of work called iterations. In Scrum, these are commonly called sprints. A sprint is a short, completed deliverable. This deliverable can be improved over time. It's much different from a traditional project.

In a waterfall project, there's 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 bit-by-bit improved 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 help you build a vegetable garden.

Say you wanted to have three planters, each with vegetables: tomatoes, peppers, and zucchini. Each planter has to have soil, a specific fertilizer, and seeds. You get proposals from two different gardening teams. The first waterfall team 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 you add the appropriate fertilizer. In the final phase, all three planters get seeded for each vegetable. The second team recommends that you finish your garden bit by bit. The tomatoes are your favourite vegetables, so you ask them to start with this planter. For sprint one, they add soil, fertilizer, and

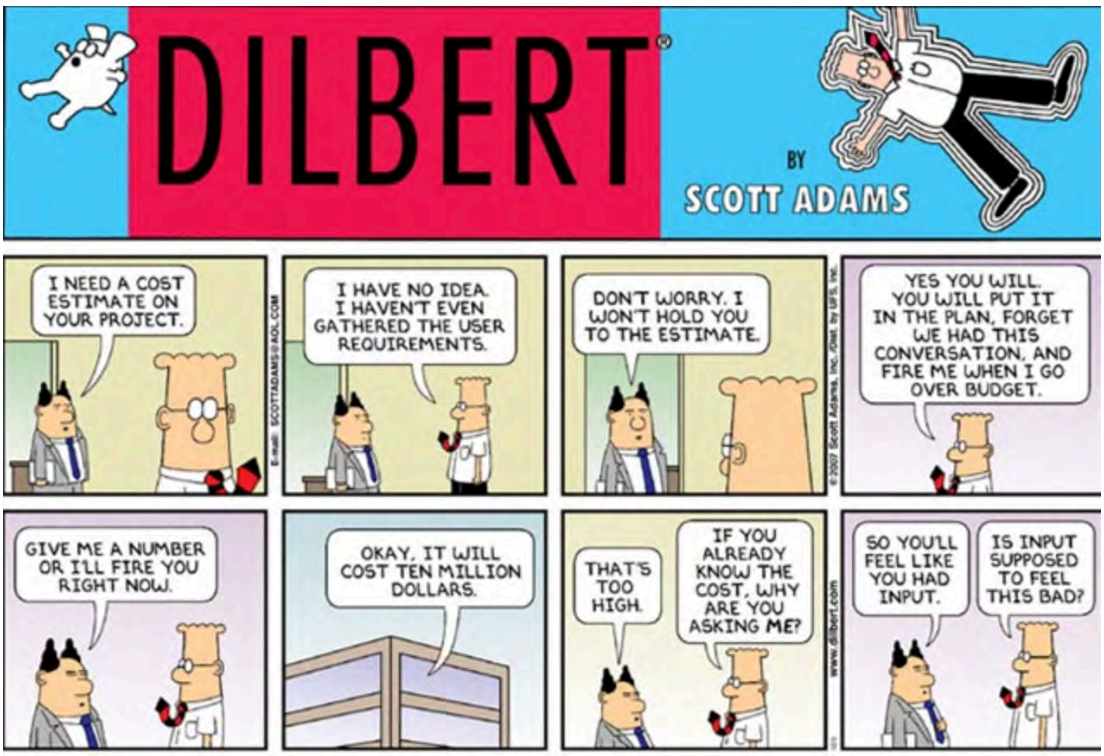
tomato seeds to one planter. For sprint two, the second planter. For sprint three, the final planter.

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 favourite plant will be delivered first. It will start growing after the first two weeks. If you went with the waterfall team, you had to wait for six weeks before anything started growing. You'd also have a lot more flexibility with the agile team. Let's say that you decide to cancel your pepper and zucchini plants. You ask the agile team to plant tomatoes in all three. There's a good chance that you won't have much rework. At worst, you'll just have to redo one whole planter.

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

Now, let's say you're a do-it-your self-person and wanted to create your own garden. Chances are, you would do the waterfall approach. That's just how most people think about working. They try to finish one phase, and then they go on to the next phase, finally wrapping things up at the end. Incremental delivery is not just a way to divide up work; it’s an alternative way to deliver work. One of the biggest challenges with agile teams is breaking loose from this phase mind-set. It's very common for an organization that has a lot of traditional project managers to struggle with this difference.

How to Plan with Estimates



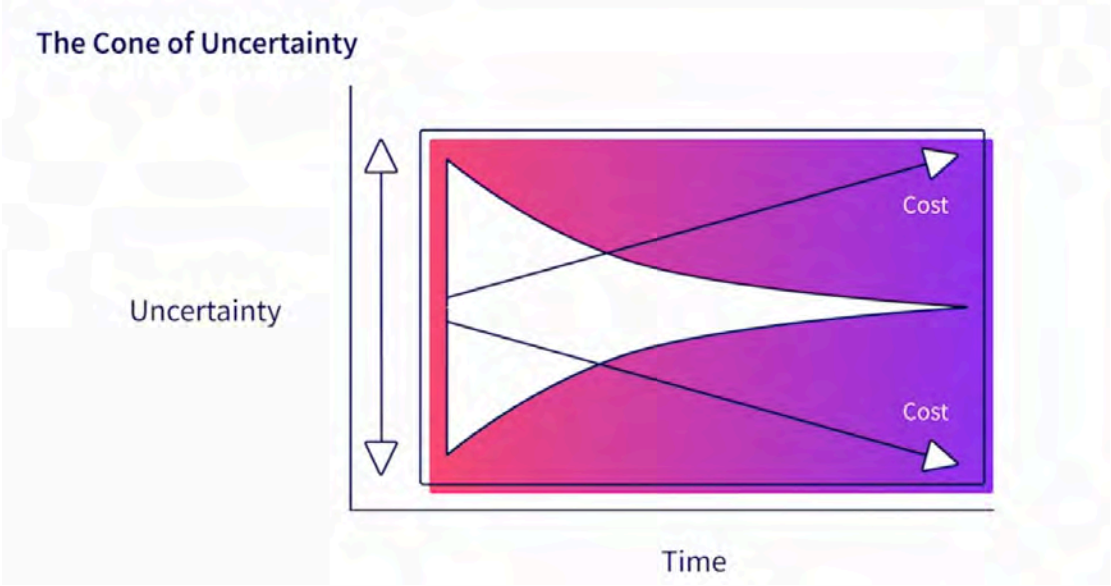
Projects are unique by definition. There'll be something in your project that you haven't done yet. Usually the greater the uncertainty, the more trouble you'll have with traditional project planning.

Traditional projects depend on upfront planning. If you have a lot of uncertainty, then you'll have a lot of guesses in your plan. If your guesses are wrong, then it will create a lot of instability. Traditional projects will work much smoother if your predictions are very accurate.

Think about a typical project. The longer the project runs, the more you know about what it takes to finish. That's usually because you're learning a lot more with working with the project, and less from predicting.

Unfortunately, the longer you work on a project, the more expensive it is to make changes. This is called the project's Cone of Uncertainty.

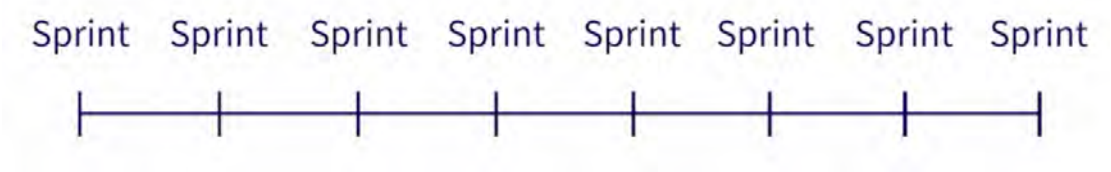
See example below……



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 the projects are the most uncertain before they start. As a project moves forward through time, you can see that it was more expensive to make changes. 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 upfront. 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 address the problems.

The Agile approach to the Cone of Uncertainty is quite different. Agile breaks it down bit by bit over time. These are the iterations of the project. They're also called sprints. The team will also start with the highest value items first. Each sprint would have a stakeholder check-up at the end of the dashed
line.



The product owner will present to the stakeholders to make sure they're building the right thing. In a sense, each one of these dashes is the team asking, "Am I building the right thing?" If the answer is yes, then they'll move on to the next dash. The team will use this rhythm to complete the project. The project will continue that way bit by bit until the stakeholders agree that all the functionality is complete. The traditional project team will always be working on the entire project.

The only way to divide up the work is by level of effort and time. 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. Agile breaks things down into smaller bites that you can estimate. This can minimize the rework and increase the accuracy of the estimates.

User Roles and three CCC

The product owner is the team's North Star. They're the light that represents the customer 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. The customer's language is much different from what you'd see in a traditional project. It focuses on the user's experience, and not what goes into creating that experience.

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. For a website, there could be many user roles. There could be a user role called 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 customer. You could then divide this into other roles, maybe creating roles called customer premium or customer standard.

If you come from traditional project management, you might be tempted to confuse user roles with stakeholders. For stakeholders, you should think of them as a person. Maybe they are a customer stakeholder. Another stakeholder could be the sponsor for the site. Someone might enter the website as a visitor, then they'll register on the site and become a standard customer. Once they are a customer, they might decide to be a premium customer. 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 on the site. Usually, the product owner will create all the user roles for the project. It'll be a real challenge for the product owner 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? Do we want to divide up user roles by those who are premium from those who are standard customers? These are tough questions a product owner will have to sort out early. When creating these roles, it's sometimes easier to think about their context, character, and criteria.

Context

The context is the bundled experiences the person will have based on where they are in your product. In the website, the product owner will probably have separate premium customers from standard customers. The premium customer will have a different context, but probably expects some extra content. They probably won't expect the first page of the website to be a login and a password box.

Character

The character is a little bit of insight into the experience a role might expect when interacting with your product. A visitor might be more curious about who runs a website. A standard customer might be more curious about how to upgrade their account.

Criteria

Finally, the product owner might create new user roles based on the person's criteria. They can create a role based on someone's expected outcome. That premium customer might be more interested in the available services. For them, the criteria are getting something that a visitor wouldn't get. If you're starting up creating user roles, it's easier to start broadly and create subgroups of experiences within those larger roles. With the website, the product owner could start with the customers and then break down the roles into different sub roles. Try to use the context, character, and criteria of the role to break down the bundle further.

Creating User Stories

Project requirements strive to be written in clear, direct language. It's usually when you turn this clear language into work that you realize that there's a lot of room for misinterpretation. Unfortunately it's almost impossible to predict how each person will interpret your language. Often a quick conversation is the only way to make sure that everyone knows the work.

Agile projects don't use traditional project requirements. Instead the product owner maintains a set of user stories for the project. A user story is a conversation vehicle. It's a short story from the users perspective about what they find valuable in the product. A common user story format is as a user role, I want, need, can, et cetera, a goal so that reason. For a website there could be many user roles. Say you've created a user role called premium customer. Now you've attached some business value to that role. For this story you might say, as a standard customer, I want to see a list of benefits of upgrading to a premium customer so I can see if it's worth the cost.

A user story accomplishes two things. First it uses the customer's language. The product owner doesn't need to know how to build a web server to ask these questions. There's no talk about the 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 the standard customer really wants to find out about the benefits of upgrading. Maybe there's a better way to do this. Maybe a window after the user logs in. A good way to create user stories is to think about the three Cs. This comes from the world of extreme programming and it stands for **card**, **conversation**, and **confirmation**.

User stories are best recorded on a three by five index card. Many product owners are tempted to create a user story spread sheet. If you try to do 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 a list 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 a user story. 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 group conversation.

The final part is the acceptance criteria. This is the confirmation that everybody knows how to deliver the story. 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. It should be the result of the conversation with the development team. This will go into much greater details on how to deliver the story. The acceptance criteria are a good way to keep the user story from becoming to convoluted. You can add a lot of details to the acceptance criteria without putting it on the front of the card.

Writing Effective Stories

User stories are a great way to have conversations between developers and the product owner. It's usually when you start having these conversations that you realize that your user stories are not very good, but that's okay. The real challenge will be figuring out just why your user stories are so bad.

Thankfully, there's a good way to figure out the challenges with your user stories. You can use the acronym INVEST. This is a list of common challenges that was created by Bill Wake to think about when you're creating your user stories. The I stands for independent. It's important to remember that independent doesn't mean your stories are functionally independent. It means that they are independently valued. Let's say that you had a mobile application that identified a bird when you took its picture. The product owner could create two stories. One story to return the list of identified birds, and another to sort the list for likely to unlikely matches. 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 a list.

Maybe the first version will just show all the possible matches. The N in INVEST stands for negotiable. The product owner shouldn't create user stories that focus too much on how to deliver the request. For the bird finder 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." The user story like this isn't focused on what the customer wants. Sometimes you'll get these directive style user stories when the product owner has too much technical knowledge.

Most companies don't care about how you sort the list; they just want to sort it. You want the development team to negotiate with the product owner. The developer should decide the best way to deliver the value in a story. The V in INVEST stands for valuable. The customer value is the most important part of the user story. Without it the product owner can't prioritize the backlog. It's very common for the product owner to struggle with value when they've come from a traditional project management background. What often happens is that they'll see user stories as rewritten requirements on index cards.

Then they even create a requirements document and then break it down into user stories. Typically when this happens you get user stories with little or no customer value, something like, "As a bird finder, 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. The E in INVEST stands for estimable. When developers can't estimate the 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 this developer struggle to estimate a user story, then you may need to go back and break it down further. The S in INVEST stands for small. A user story should be delivered within a two-week sprint. 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 T in INVEST stands for testable.

Independent	We want to be able to develop in any sequence.
Negotiable	Avoid too much detail; keep them flexible so the team can adjust how much of the story to implement.
Valuable	Users or customers get some value from the story.
Estimatable	The team must be able to use them for planning.
Small	Large stories are harder to estimate and plan. By the time of iteration planning, the story should be able to be designed, coded, and tested within the iteration.
Testable	Document acceptance criteria, or the definition of done for the story, which lead to test cases.

Think about a user story like "As a bird finder, I want to see the list of birds quickly "so I can identify my bird." This might be a true statement of customer value; the bird finder will certainly want the application to work quickly and easily. The problem with the story

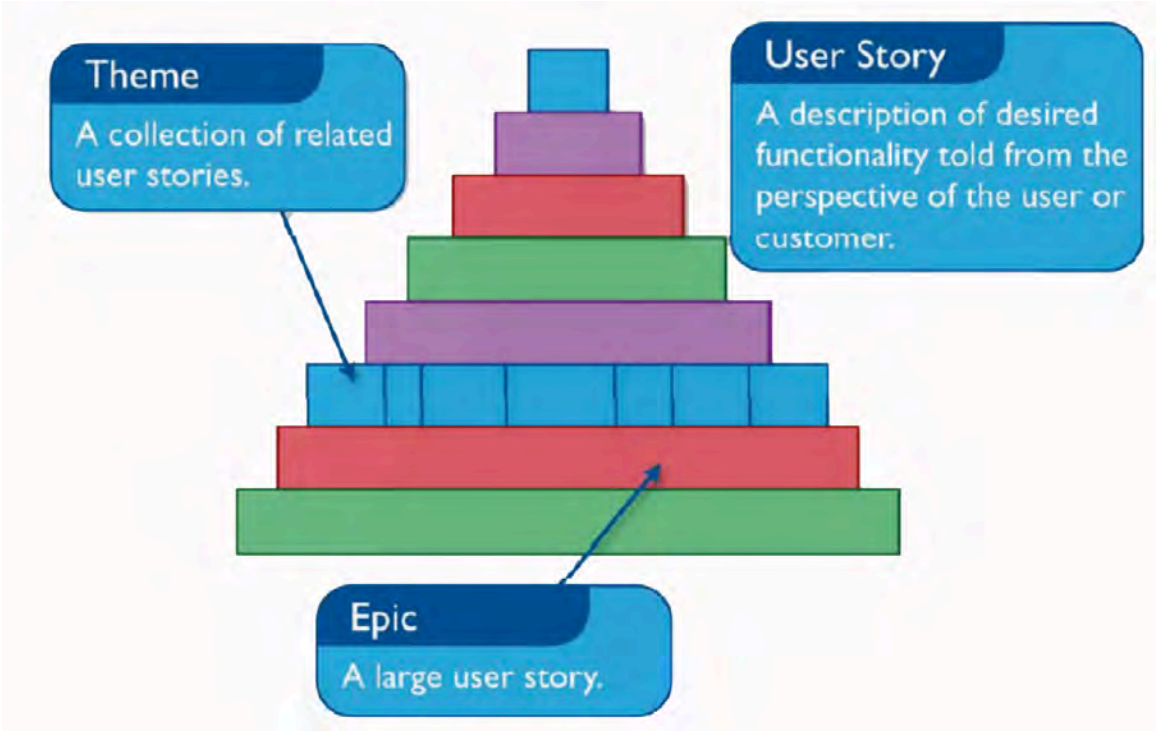
is it's too subjective, it's not testable. What is quick for one customer might be slow for another. How do you test "quickly"? Will the product owner be the one who decides what is quick? Watch out for these adjectives and adverbs in your user stories.

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.

User Story: Key Take Away/ Facts

- User Stories are not detailed narratives.
- User Stories don’t need to tell the development team how the feature works.
- User stories don’t need to come from users.
- User stories aren’t the final word in development.
- User stories aren’t one size fits all.
- User stories don’t need to be independent.
- User stories don’t need to include obvious details.
- Everything in your product backlog doesn’t have to be a user story.
- User stories are not always from an end-user.
- User stories are not a mandated part of Scrum.
- You don’t have to follow a specific user story format.
- There are no real rules for writing user stories.
- User stories are not always shared with customers.

Grouping with Themes or Epics



Sometimes user stories start their lives as epics. The product owner usually creates a large value statement instead of several smaller stories. That's okay. That's just the way most people think. A theme, or epic, is a way to organize stories into groups. You might have an epic to upgrade the database. It's important to remember that these are large big epics and not the same as user stories.

They are naturally grouped together and need to be split up. Epic splitting isn't an easy task. There are several common ways to split your stories. The best way to remember these groups is by thinking of the acronym F.E.E.D.B.A.C.K.

Let's start with a user story that we might use for our Bird Finder application. "As a bird finder, I want to see a list of matching birds "so I can learn more about bird watching."

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

The F in Feedback stands for flow. This is how the story might step through the application's workflow. You can split this epic into a new story. "As a bird finder, I want to see a list "of matching birds on my smartphone, "so I can learn more about birds "when I'm away from my house." And you could substitute smartphone for a tablet for even more stories.

The first E in Feedback stands for effort. Here you might break down the epic based on the developer's level of effort. A new story might be, "I want to see a list of matching birds "sorted by likely matches, "so that I can learn more about bird watching." Here you split the story based on the effort. The developers would spend most of their time creating the logic to find likely bird matches.

The second E in Feedback stands for entry. Sometimes the team might want to break down the epic by how the customer enters the data. A new story might be, "As a bird finder, I want to see a list of matching birds "based on my uploaded GPS coordinates, "so that I can learn more about bird watching."

The D in Feedback stands for data operations. You may want to break up your epic based on common data operations, like read, update and delete. A new story might be, "As a bird finder, I want to see "an editable list of matching birds "so I can learn more about bird watching."

The B in Feedback stands for business rules. Sometimes you'll see there's a lot of complex value in the epic. When that happens, you may want to split that epic down into

business rules. These are the rules that help you expand on the goals of the initial epic. "As a bird finder, I want to see a list of matching birds "based on the likelihood of a match "on GPS coordinates and the image, "so I can learn more about bird watching."

The A in Feedback stands for alternatives. Some epics can easily be broken into stories with alternative criteria. You could create another story based on alternatives. "As a bird finder, I want to see a list "of matching birds located in my zip code, "so that I can learn more about bird watching."

The C in Feedback stands for complexity. Many epics are just the beginning. When the product owner starts thinking, they find greater and greater value. When this happens, the epic is just a simple start. Then you can break it down into stories with increasing complexity. A new story might be, "As a bird finder, I want to see a list of matching birds "based on color, GPS coordinates and recorded audio, "so I can learn more about bird watching."

The K in Feedback stands for knowledge. Sometimes the product owner will present the epic and the team will need more knowledge. They have to research the epic and break it down into stories. When this happens, the developers will create spikes. Story spikes aren't stories but they lead to stories.

They're open questions that the team needs to answer to deliver the story. They're an IOU for the product owner. The developers just need to answer a few questions. The developer might create a spike that researches the ability to upload recorded sound from a smartphone. It's an open question that needs an answer. Once they have the

answer, they can work with the product owner to create the story. It's very common for new Agile teams to create more epics than stories.

Try to remember the F.E.E.D.B.A.C.K. acronym as a way to break down these epics. In Agile, you want to estimate the smallest chunk of work so the better you break down your work, the easier it will be to estimate the story.

Creating your Project Charter

Having once worked for an organization on a large Agile project. The project had started about six months earlier. After I arrived I asked to see the project charter. I figured that the charter is always a good way to see the overall direction of the project. One thing I immediately noticed is there wasn't any success criterion to finish the project. There was also no overall mission. I asked about this and the scrum master said, as long as they keep delivering value the team would be successful.

In some ways the scrum master was right, a key tenant of Agile is adapting to deliver value, not predicting the value before your begin. But there's a difference between the team's success and the project meeting its vision. I wasn't suggesting that a team should build a charter to control the value. The customer should create a formal charter to set the initial direction and record the project's reason for starting.

An Agile Charter should be an agreement and not a plan. It should list what every party wants from the project. There are many successful teams that work on projects that ultimately fell short of their vision. It will be important for the team to see what, if anything, they could have done differently. A good Agile project charter should be one page or less. It should have three main sections. There should be the project's vision, mission and success criteria.

The project vision should answer one question, why are we doing this project.

The project's mission is what everyone will do to accomplish the vision. How do you see the project playing out? How will it meet its vision?

The success criteria are the most practical of the three. These will be the simple, one sentence tests to make sure that the project accomplished its mission.

Let's think about our bird finder application that identified a bird when you took its picture with your phone. Good vision statements for the bird finder application might be, “help people enjoy the outdoors "and learn more about the wonderful birds "that live on our planet." Notice how this vision doesn't talk about technology. It talks about

the initial idea. Even if the technology changes, the idea might stay the same.

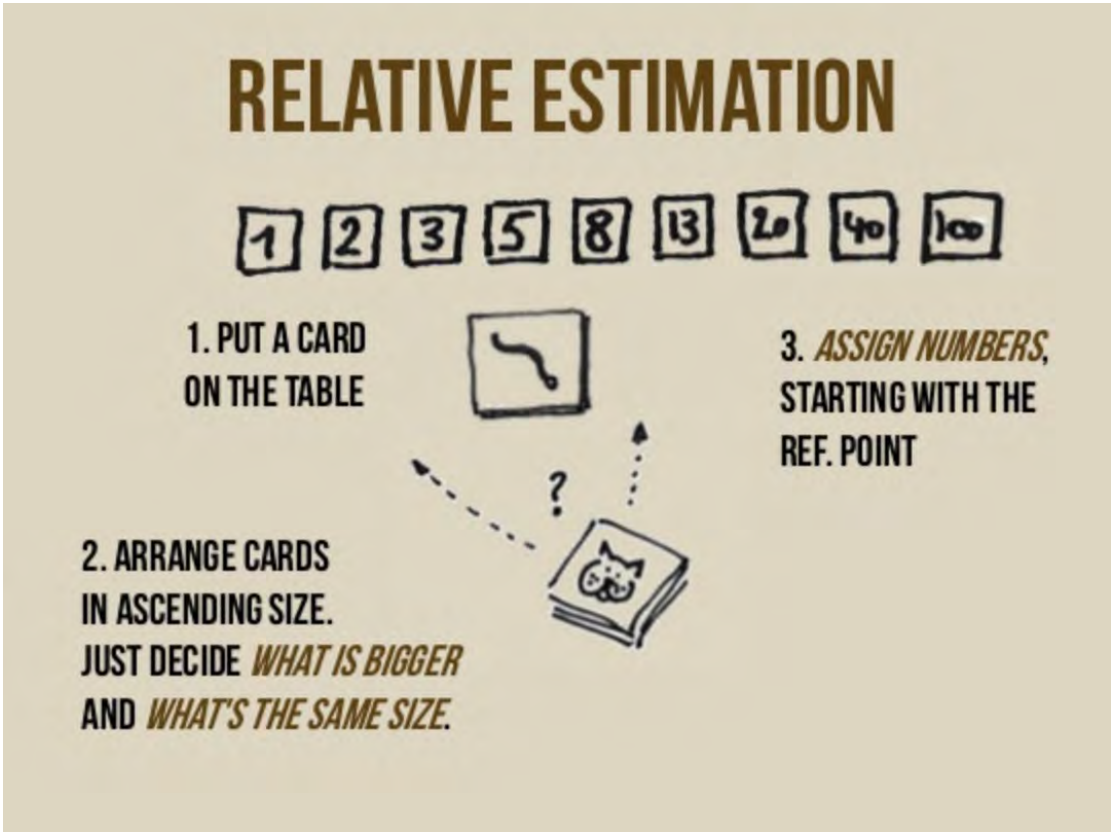
A mission statement for the bird finder application might sound like this. "Create a cutting-edge technology platform "that combines a website, a smartphone application, "and location awareness to help our customers "identify local birds."

The final section is the project's success criteria. For bird finder application a few success criteria might be, sign up 10,000 users within three months of the initial version. Sign up 100,000 users in the first year. Get endorsed by a major ornithology website.

Be careful not to confuse the charter's success criteria with functional success. The smartphone application might have all the functionality described in the mission statement, but it might be too expensive, too ugly, or counterintuitive. The application might be a success functionally, but it wouldn't help accomplish the project's vision. Not enough people would use it and so it wouldn't connect others. Setting up a good project charter is an essential part of closing out the project.

At some point the product backlog will be exhausted, the project will either run out of money or deliver all the value the customer requested. At that time the team will hold their final retrospective. It's when they look at the whole project from beginning to end, and record the lessons learned. It's then that the team, and the customer, can decide if the project delivered on its initial promise.

Using Relative Estimation



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. 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 get it done. The team won't think in hours, days or weeks. They'll think in relative sizing.

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 it might be three or four dogs. This estimation in 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 from 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.

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

An estimate is the useful information you might give a co-worker. 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 20 minutes each way, so 40 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 10 hours a week. Even with the simple task of

driving to work, you have a range of three to seven hours a week. That doesn't really help anyone with planning. Relative estimation is a key part of adapting over planning. There's always limited time in a project. A relative estimate allows you to immediately get into the work. It keeps your project from spending too much time debating how long a task will take. This can free your team from spending too much time trying to create false precision. You might find that time estimates aren't worth the effort it takes to create them.

Playing Planning Poker

In Agile you're not estimating for individuals, you're estimating work for the entire team. Many different developers on the team work on the same story. Some developers might decide the story is more complex. Another developer might think that the story is less complex; they could finish it up before lunch. There's no way to tell which developer is right until the

work begins.

Planning poker is a card game that helps the team get the best story estimates with less than perfect information. Planning poker helps give everyone a voice. Planning poker is not just about creating an estimate. In many ways the estimation is just the byproduct of the game. The activity is really about combating groupthink.

Groupthink is the way that people tend to agree with the most popular idea. The strongest voices will drown out any disagreement. Planning poker tries to combat this tendency. Each team member makes an estimate before they're told what everyone else thinks.

Each attendee will have a deck of planning poker cards. They'll have an exponential number sequence, something like zero, one-half, one, two, four, eight, sixteen. These are the points for each of the stories. The team identifies one of the smaller stories to assign a value of two. This will be the estimation baseline. The product owner will pull out other stories from highest to lowest value.

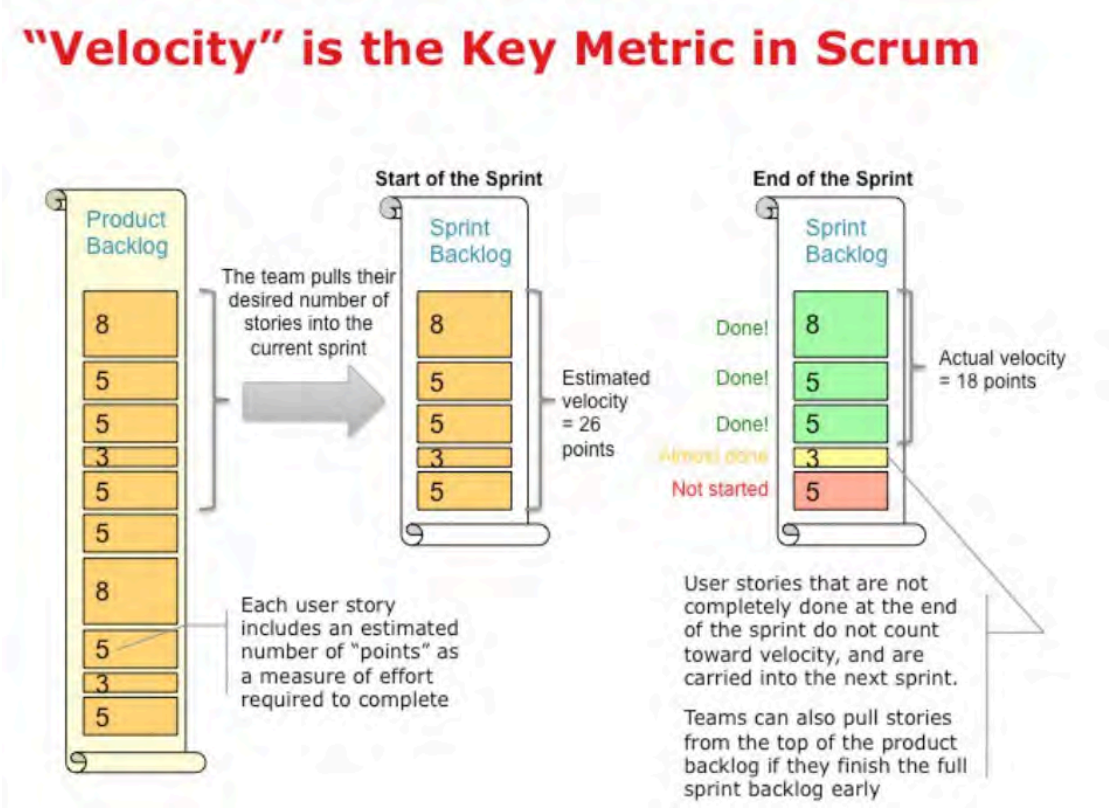
Even the scrum master, the product owner, or one of the developers reads the story to the team. If you're the scrum master for the team, make sure that there's no discussion about the story before the first estimate. Any discussion can lead to groupthink. You could lose individual ideas before you even begin playing. Each team member will make a relative estimate for all of the stories. They'll use story points as an estimate of the relative size. Each team member selects a card for their estimate and places the card facedown. 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 simultaneously. If there's 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 group briefly debates with other members who have different estimates. This should continue until the team reaches consensus and the scrum master records the estimate in story points. Remember that this is a relative estimate.

If you're the scrum master for the team, try to make sure that everyone thinks in story points and not hours, days, or weeks. The team will want to repeat this for each story. Planning poker is a key part of making sure that everyone on the team is part of the estimation. The whole team will only estimate, so the whole team needs to take part in estimating. There will certainly be people on the team who know more or less about a task. The team member who knows more about the task has a duty to explain their understanding to the rest of the team.

Calculating your Velocity



In 1914 Henry Ford noticed that his employees worked best when limited to eight hours. After eight hours their productivity began to slow down. That's why Ford Motor established the 40-hour work week. The change increased their productivity and many companies soon followed. That's why the 40 hour work week became standard.

In the 1980s fewer people worked in manufacturing. They became desk workers and computer programmers. Because of this, many employers felt that the 40 hour work week was dated, it only made sense in manufacturing. Working a 70 hour work week became a badge of honor for desk working go-getters. The Agile framework pushes back on this ever-expanding work week. Agile promotes the idea that people should go home at a reasonable hour. Often developers will think

more clearly when they're driving home, or walking their dog. An Agile project should have a challenging pace. There shouldn't be any project crashes. The team shouldn't have a flurry of activity at the end of the sprint.

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 10, 11, or even 12 hours a day. They're much better off going home then starting fresh the next day. 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 look at their own work history to determine their sustainable pace. They'll see how much they have worked and then that will be how much they will commit to working to in the future. This is commonly called the team's velocity.

Typically the Scrum Master will calculate the team's velocity. They'll create the number based on the number of 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, one story was estimated at three points, and the last story was estimated at five points. This means that at the end of the sprint the team completed 16 story points. Now the team knows that they can complete 16 story points in their first sprint. They'll do the same thing with their second sprint. Let's say after their second sprint they completed 17 points. Then in their third sprint they completed 15 points. Now the Scrum Master 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 16 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. One of the challenges with velocity is that it's often misused. Senior managers will sometimes try to push the team to increase their velocity. They think that means that the team is being more productive. But remember, that story points are a relative estimate. The team assigns them in the planning poker session.

I once worked for an organization where a senior manager was pushing the team to achieve greater velocity. They thought that increasing the velocity meant that the team was finishing more work. The team started out with a velocity of 15. Then the manager pushed the team to increase the velocity to 18. Then to 25, and then finally to 30. The manager was delighted to see the team reaching these goals of hyper-productivity. In reality, after the manager started to push the team they started estimating in three, five, and eight points. The team wasn't really completing more work they were just estimating the work with higher numbers. 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 of the team tries to alter that number it becomes much less useful.

Velocity Overview:

Points from partially completed or incomplete stories should not be counted in calculating velocity. Velocity should be tracked throughout the Sprint on the Sprint Burn down Chart and be made visible to all Team members.

Velocity is a key feedback mechanism for the Team. It helps them measure whether process changes that they make are improving their productivity or hurting it. While a Team's velocity will oscillate from Sprint to Sprint, over time, a well-functioning Scrum Team's velocity should steadily trend upward by roughly 10% each Sprint.

It also facilitates very accurate forecasting of how many stories a Team can do in a Sprint. (In Scrum this is called using Yesterday's Weather.) For forecasting purposes the average of the last three Sprint's Velocity should be used. Of course, this means it takes three Sprints of experience for a Team to determine its Velocity accurately, which can sometimes be difficult to explain to impatient stakeholders.

Without Velocity, Release Planning is impossible. By knowing Velocity, a Product Owner can figure out how many Sprints it will take the Team to achieve a desired level of functionality that can then be shipped. Depending on the length of the Sprint, the Product owner can fix a date for the release.

Writing your Release Plan

In Agile, there isn't a release schedule. Many teams will release working software at the end of every sprint. Some teams, even sooner. But that doesn't mean you can't plan on bundles of value being delivered over time. In many organizations, it's still important to align your Agile delivery with other projects. Agile projects don't have milestones. There's no scope. There's nothing to split into quarters, months, or weeks. Instead, Agile has groups of stories called epics.

Your epic should represent a large chunk of customer value. What you can do with these epics is create a rough

order of value, or ROV. These are planned groupings of epics that you can deliver over time. To create a release plan, you can assemble all the stakeholders and do some real-time horse-trading. In this event, the scrum master puts all the epics on a trading board. The team has a set number of stories that they can deliver in each sprint. This is called the team's velocity. Their velocity might be 50 story points for each sprint. An epic might have five stories that are each estimated with two story points. That epic would have a 10 story point value. That means that the team could deliver this epic and another 40 points for the sprint.

The Agile horse traders need to decide which epic to deliver. It could be another big one for 40 points, or maybe two small ones for 20 points each. The scrum master will pull the epics from the board and into each sprint. The customer decides what's the order of delivery. It's important to keep the customer using the language of value. If they think in milestones, they'll think about scheduled commitments. That's not part of the Agile plan.

The product owner might decide after three sprints, that they want to add more value to the website. That's perfectly acceptable. But adding value to the website, might delay delivery of the mobile application. The product owner should know that many changes would impact the rough order of value.

Let's see how this horse-trading might look in a release plan for our Bird Finder application. The Bird Finder application could have created dozens of epics for this initial release. Let's say that five epics were create bird finder search, create bird profiles, create bird finder login, design landing page for website, create map page with bird identification.

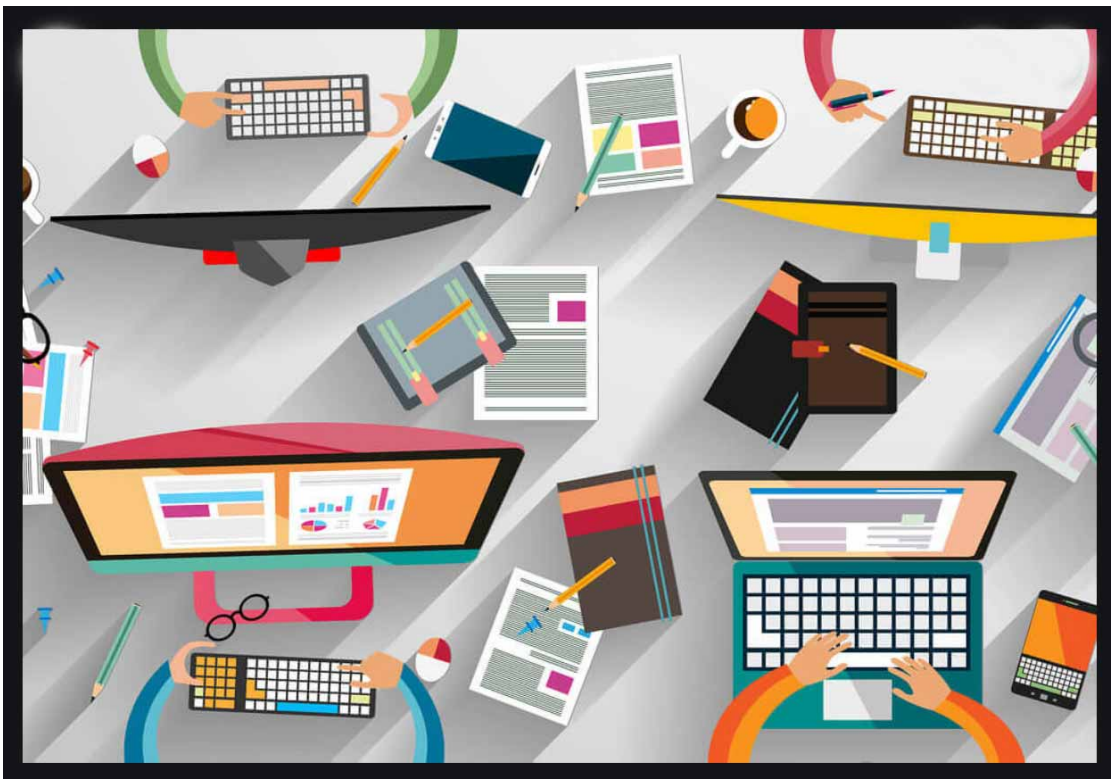
Now imagine that each of these epics were filled with estimated stories. From top to bottom, the total story points were 10 points, 20 points, 30 points, 40 points, and 50 points.

Bird Finder App Epics	Story Points
Create bird finder search	10
Create bird profiles	20
Create bird finder login	30
Design landing page for website	40
Create map page with bird identification	50

On the trading board, each of these epics would have this number underneath them, representing this total. So create bird finder login would have the number 30 written under it. The scrum team has a velocity of 50. That means that every two weeks, the team could complete 50 story points. Now is where the horse-trading begins. Does the customer want to start with the second and third epics, or do they want to start with the last epic because it's the most important?

Let's say the customer didn't care as much about creating the landing page for the website. Instead, they decide to start with epics two and three for the first sprint. Then deliver epics one and four in the second sprint. The team now has a release plan for the first month. The two sprints will amount to four weeks of work. The four epics will give you a rough order of value. At the end of that month, the website should have a bird finder provider search, bird profile, a bird finder login, and a map page with bird identifications. Again, the product owner could change the priority in the second sprint. If they do, the value delivered at that time might change.

Producing Acceptable Documentation



One of the sticking points with new Agile teams is deciding what is considered acceptable documentation. Remember from the manifesto that Agile projects prioritize working software. That doesn't mean that Agile projects produce no documentation. Instead, it should produce just enough documentation.

After the Agile Alliance wrote their manifesto, they created a list of agile principles. Principle seven says, "Working software "is the primary measure of progress." Principle one says that you should "satisfy the customer through early "and continuous delivery of valuable software." You should think of this principle as really saying, "Don't deliver documentation "as a measure of progress." If you think about it, this makes a lot of sense.

Try to imagine you're working on a large software project. So we start our projects the way most traditional projects start. We'll gather all of our requirements. A business

analyst will work with the project stakeholders to create a comprehensive business requirements document. At this point, no one is developing any software. It's just a business analyst interviewing customers of what the product will accomplish. This could take weeks, months or even years depending on the size of the project. At the end of it all, the customer will have a complete set of documentation for the whole project.

Keep in mind that very little software has been developed. Instead, most of the time on the project has been focused on delivering comprehensive documentation. This is a large investment in the project. If the project is canceled outright, the investment will go to waste. If the team were using Agile, it'd have a few months worth of valuable software. It might not have anything near to what the final product would have, but it would be something. To make sure that your team doesn't produce too much documentation, try to remember these three things. An Agile team should document continuously and not wait for a final draft. All the documentation should be collaborative. It shouldn't be a binder dropped on someone's desk. Finally, the primary purpose of the documentation should be the continuation of the project.

Remember, the primary reason for documentation is to make sure that others outside of your project will understand the work. Remember that an Agile team focuses on the highest value deliverable. When the team is working on creating documents, it means that they're not working on creating valuable software.

Sometimes with newer Agile teams, you'll hear them say, "We don't plan because we're Agile." This causes a lot of confusion with the rest of the organization. They'll justifiably ask the question, how can you run a project without any planning? The truth is, you can't.

Agile does have planning, it's just different from traditional project management planning. The planning rules are there, but they're just much more lightweight. You'll find this is true with a lot of Agile frameworks. In Agile, there aren't usually very many rules, but at least at the beginning, the rules that do exist should be closely followed. You don't want to start your Agile project by creating your own version of Scrum, extreme programming, or Kanban.

Being Agile means that you allow your customer to change the product. It doesn't mean that you're free to change the framework. That means that if you're the Scrum master for the team, you should be very careful not to let the team get too creating with how things get done.

Every team needs to have a product owner. The team needs to decide the length of sprint and then stick with it until the end of the project. There's plenty of flexibility in the product, but there should be much less flexibility in

how you deliver it. Your team shouldn't think of Agile as a free ticket to work however they like.

This is particularly true around Agile project planning. Each planning event like the release planning event has a specific agenda. The Scrum master needs to make sure the agenda is closely followed. Again, it's important to not confuse the word Agile with how you run these events. If anything, Agile planning events are much more formal. They'll have a set amount of time with some direct purpose.

One time I worked with a team that was starting to use Agile for the first time. They didn't have the people available for a full Agile team, so they merged some of the roles. The Scrum master and the product owner was the same person. One of the developers also acted as a project manager. They made all these changes and declared that they were creating their own Agile framework. It's important to not take this approach when you're starting out with your Agile team. Start out by following the framework as closely as possible. Only after your team is well formed should you even consider making changes to the Agile framework.

Agile Summary

Agile is a new type of planning system and would not be relevant or the correct system to be using for all projects. There are many industries that it would not be useful and where a clearer and more accurate planning system is required.

I understand that it is trying to cover an industry, mainly the software industry, which has been difficult to plan for in the past. It does what it says on the tin, and tries to

meet the needs of an ever-changing environment at the same time trying to deliver something rather than nothing.