Agile Project Planning

Properly estimating and prioritizing project requirements is critical when planning projects. When managing Agile projects, you need to be able to map out the work necessary for your product release.

In this course, you'll learn about vital Agile planning activities including creating personas and wireframing. This course also covers the most common Agile estimation techniques used when managing projects such as story points, wideband Delphi, and affinity estimation. This course also covers requirements prioritization methods and other Agile project management activities you perform when completing your release plan.

This course is one of a series in the Skillsoft learning path that covers the objectives for the PMI Agile Certified Practitioner (PMI-ACP)® exam. PMI-ACP is a registered mark of the Project Management Institute, Inc.

Table of Contents

1. Agile Project Planning
2. Defining Project Scope and Roadmap
3. Use Cases and Personas
4. Formatting User Stories
5. Using Story Maps
6. Agile Estimating Techniques
7. Estimating Team Velocity
8. Value-based Prioritization
9. Prioritizing User Stories with MoSCoW Model
10. Tools for Prioritizing Requirements
11. Completing the Release Plan

Agile Project Planning

[Course title: Agile Project Planning. The presenter is Barbara Waters, PMI-ACP.] Agile project planning involves establishing the proper project requirements, conditions of satisfaction, and priorities. This course covers how to create and prioritize user stories. A key activity for clearly defining customer-centric requirements and features. You'll also learn about agile estimating techniques, estimating team velocity, and prioritizing user stories and requirements.

Defining Project Scope and Roadmap

[Topic title: Defining Project Scope and Roadmap.] Two key planning stages in Agile are release planning and iteration planning.

At the release planning level, we define the plan for delivering a certain increment of product or business value by a target release date. Release planning helps stakeholders start to set expectations on what they can start to see within each iteration as they approach the final release. The key tasks of release planning include, establishing project goals, creating user stories, prioritizing stories so that the team and stakeholders identify what must be developed, estimating stories, grouping stories, and setting a release date.

Iteration planning is sometimes referred to as sprint planning. It defines specific goals for the team in the next iteration. The key tasks of iteration planning are, confirming user stories and priorities for the next sprint, decomposing selected stories into tasks, refining estimates, and updating requirements if needed and based on new learning.

In Agile planning, we have the concept of a cone of uncertainty. *[A graph displays. The X-axis depicts time and the Y-axis depicts variance.]* Essentially the concept is that there is a decrease of variance overtime and estimates. So as we move forward in a project, we have last project unknowns and our estimates versus actual start to approach each other, and become closer until there is no variance between them. Estimates may have been created using historical project data. They're used to create a high to low estimate range, and they vary depending on the type of project.

In Agile planning, we also have the concept of progressive elaboration. Progressive elaboration means that the project plan is continuously and constantly modified, detailed and improved. Progressive elaboration helps us account for changes in scope or requirements or even priorities. Progressive elaboration is performed throughout the project life cycle and it allows the project to be managed to a greater level of detail. So as the project evolves, we're actually able to manage better, because we understand more detail about the requirements, about the users and about how they're going to use our product.

In the context of Agile projects, scope is defined as the extent of the work that needs to be completed in order for our project to be considered complete. Overtime, scope usually increases. Scope is likely to change over the course of a project because requirements change and because requirements are refined as the project progresses.

In Agile, we understand and acknowledge this fact. And we don't try to run away from the idea that scope may change as our project progresses. When we incorporate learning and feedback, we actually can create better products. Ultimately our goal is to create the best products for our end users and for our stakeholders. Scope can be controlled by continually maximizing value for the customer, such as by minimizing waste and increasing effectiveness of teams.

Scope creep is another concept in project management. Scope creep is uncontrolled change to a project which can result in delays or higher costs. Scope creep should not be confused with controlled change, which is accepted and even encouraged if it adds value for the customer. We must avoid scope creep by introducing change in a managed way that considers the impact to other parts of the project and is agreed to by the customer and key stakeholders.

So what do we do if our budget and schedule are fixed, but we want to introduce a new feature or requirement? We may decide to sacrifice a less valuable feature to make room in the budget and schedule for this one. This is one way to prevent scope creep.

In summary, two key planning stages in Agile are release planning and iteration planning. The key tasks of release planning include, establishing project goals, creating user stories, prioritizing stories so that the team and stakeholders identify what must be developed, estimating stories, grouping stories and setting a release date. The key tasks of iteration planning are, confirming user stories and priorities for the next sprint, decomposing selected stories into tasks, refining estimates and updating requirements if needed and based on new learning.

Use Cases and Personas

[Topic title: Use Cases and Personas.] Use cases are common in both software and system engineering and they're part of defining requirements.

Use cases describe a sequence of activities that provide measurable value to an actor who represents the user. You may think of use cases as different scenarios. So for example, one use case may be where a user already has an account on the system. What happens then? Another use case may be where a user is trying to use our system, but they haven't yet established an account. In this context, actors represent the roles that users may have in relation to a system. So actors may be persons, organizations, or external systems. A person may be a user who directly uses the software. An organization may be an organization that benefits from some reporting or some other feature of the software. And an external system may be the recipient of some data that our software is sending to it.

We use personas in Agile projects as a way to define examples of typical users of our systems. They should be concise and visual. When we develop personas, we think about certain things such as providing a name, a personality, what their motivation is to use our software, providing professional details, how much they use the technology, etc., and potentially a picture of that person. It helps teams to identify personas and to start to create a relationship with them, in order to understand their mindset as they're using our software. It may seem interesting to go into details of these personas, however they actually really do help us understand how users are going to use our system, and what things they may be taking into consideration as they're interacting with it.

We may want to use or create more than one persona. We implement personas in order to understand exactly what a product may need to do in different scenarios, and how it may need to react when providing different inputs from different personas. Creating personas may require community research. So, for example, we may have to conduct some focus groups. We may have to talk to potential users or even customer support, since they really understand some of the struggles that our users currently have, and product managers, since a part of their job is to understand our customers.

In order to write effective personas, we don't want to make up personas that don't actually exist. We want to base our personas on reality. They should be based on who will actually use and benefit from our software. We also want to develop specific personas so we want to be able to differentiate between the different personas. We also want to define the persona goals. Why would they use our system? We may also consider negative personas. If there were some personas that would want our system to fail, or wouldn't want to use our product, what would their motivations and reasons be? We may consider creating primary personas and then some secondary ones. The primary personas are the ones that will really drive our decision-making in our understanding of the user.

In summary, use cases and personas help you define requirements. When creating personas, best practices include, basing them on reality, being specific, defining persona goals, including negative personas, and creating primary and secondary personas.

Formatting User Stories

[Topic title: Formatting User Stories.] It's beneficial to depict user stories visually for team members and stakeholders. A great way to do this is to use wireframes. Wireframes are also known as page schematics or screen blueprints. Wireframes provide a visual guide and define information hierarchy on a page. Wireframes also make it easier for our user experience designers to plan the layout of a page. And with wireframes, users can interact with the interface before anything's been developed.

The benefits of wireframes are many. For example, for clients, wireframes help them obtain a better understanding of our solution. They also provide a format for providing feedback. For product and project managers, wireframes ensure that stakeholders are on the same page. For designers, they can rely on wireframes as blueprints for design visuals in a high-level, low fidelity way. For developers, wireframes can help obtain understanding of technical requirements.

There are several best practices to follow when wireframing. First, avoid making wireframes fancy. We want to keep wireframes at a high-level and low fidelity, so that we're not committing to a certain layout. It's interesting, but once we get into designing in a high-level of detail, we start to commit to memory and we start to commit to certain elements on the page. It's harder to detach from a high fidelity design than from a low fidelity wireframe.

With wireframes, we also want to gather feedback before we start creating them, and once we've created them in order to continually improve our design. Wireframes also provide us with an opportunity to experiment with different ways of placing things on a page, or different ways of depicting a requirement that a stakeholder has given. You should experiment with the options to ensure the best decisions are made. You should also use storyboards when wireframing to help visualize interactions.

Another best practice is to use annotations and notes, in order to note certain things that developers might want to take into consideration, or certain ways that something may need to be developed or implemented. And we

always want to provide an explanation with wireframes since they don't stand on their own.

In summary, wireframes are a great way to start to depict our requirements in a visual way. And to ensure that people are on the same page. Best practices when wireframing include keeping them simple, gathering feedback, experimenting, using storyboards, using annotations and providing explanations with them.

Using Story Maps

[Topic title: Using Story Maps.] Story mapping, originally introduced by Jeff Patton, is a powerful, visual way to plan Agile projects. Story maps are a way to organize and present information, similar to the concept of a roadmap. However, in story maps, we order users' activities and tasks as they interact with our system.

Story maps depict users' activities overtime or sequentially. User stories help us to really focus on the user in our planning efforts. When we put them in the context of a story map, it also helps us focus on business value, since that's what helps us determine priority.

In users' story maps, the horizontal axis shows a high-level overview of the system. Meaning, if we were to read the cards at the very top, we would have a high-level understanding of what the system does. So for example, the blue cards on this diagram may say something like Log In, Enter Banking Information and View Reports. We can guess that this is probably some type of financial software. The horizontal axis is also ordered based on the order of how users use the system overtime. So in the previous example, a user would most likely first log in, then enter banking or transaction information and then view reports. The first row of the story map is called the walking skeleton and is considered the minimum working version of the system that can provide value to a user.

On the vertical axis of a users' story map, we start to get deeper into additional scenarios, more sophisticated use cases and more robustness. As we go further down the vertical axis, the functionality is less necessary than the top rows.

Story mapping has many benefits. First, story mapping helps with knowing what to build first. Story mapping also helps us to build using an iterative approach. It helps us start to see things in terms of building Release 1 based on what's most necessary, and then Release 2, what's necessary for this iteration, and then Release 3, and so forth. Story maps help us to see the scope of a project and then also to plan our releases at a high-level. Story mapping also helps us continue to prioritize the product's backlog, as things may shift in-between releases as we're implementing or executing on the plan. Story mapping also has an open approach to the supporting goals and activities of each story.

So for example, when we're looking at how to solve or how to implement certain cases, we want to think about things like manual workarounds. What are the different ways that we can implement a requirement that can be the most iterative and open and easiest approach to getting something done? Story mapping allows you to explore those options.

In summary, story maps are a visual planning tool that help you organize and present your project plan. Story maps help you to identify what to build first, support an iterative approach, identify scope, help in planning releases, help with prioritizing product backlog, and provide an open approach.

Agile Estimating Techniques

[Topic title: Agile Estimating Techniques.] Understanding the different estimating techniques and when to use them is critical for your Agile project success.

There are several estimating techniques to use in Agile, depending on what your estimating goals are. Story points, one option, are a tool used by teams to perform relative estimation. By relative, we mean that when we estimate using story points, we're estimating by comparing the relative effort of user stories as compared to each

other. Story points are what we call an arbitrary measure. Which means that an estimate of three or a medium by one team, may mean a completely different thing than an estimate of three or a medium given by another team.

Story points are usually used by Scrum teams, and they express how much effort is required to implement a story or what we call the size of a story. We take three items into consideration when we're coming up with the relative size of a story. Number one, the Level of complexity involved in implementing. Number two, the Level of unknowns or risk. And number three, the Effort required to implement once we're ready to develop.

Ideal Days are a measure that can be used instead of story points. And one reason why teams might want to use Ideal Days is that they can be more intuitive than thinking about estimation and points. Ideal Days are units of time, whereas points are an arbitrary measure. They are used to estimate the relative size of user stories, product backlog items, and projects. Ideal days are how many days it will take a single developer to Build, Test, and Release the functionality described by a user story, under the ideal conditions. Ideal conditions refers to no interruptions, and that all resources are available.

Another estimating method that can be used to estimate user stories in Agile projects is Relative Sizing. With Relative Sizing, we're expressing the overall size of an item. So for example, we could be using Relative Sizing for users' stories or tasks. The Absolute value is not considered and what we mean by that is that, it's a very different method of sizing or estimating than saying that something will take exactly one hour, or three hours, etc. Relative Sizing is a cheap and fast estimation technique, there's not a lot of math involved. There isn't a lot of time involved, either. We really get together the people who are experts on implementation and ask them what the relative complexity is of one story as compared to the others.

Wideband Delphi is another estimation technique, and was developed in the 1940s by RAND Corporations as a forecasting tool. Wideband Delphi is a repeatable process. It's a group estimation technique that is used to estimate effort, and is straightforward to implement. It includes six steps, which are, scheduling an estimation meeting, describing what the team is estimating, tasking members to estimate individually, revealing the individual results, discussing, and repeating.

Planning Poker is another commonly used estimation technique by Agile teams. And it's used to determine user story size, as well as to build consensus among team members. And finally, Affinity Estimation is a technique used to quickly and easily estimate a large number of user stories and story points. This is done by quickly categorizing user stories, and then applying estimates to those categories of stories. The participants of the Affinity Estimation meeting are the Product Owner, the Development Team, and the Scrum Master.

In summary, Story Points, Ideal Days, Relative Sizing, Wideband Delphi, Planning Poker, and Affinity Estimation are all estimating techniques you can use in Agile.

Estimating Team Velocity

[Topic title: Estimating Team Velocity.] A team's velocity is defined as the actual amount of development work completed, usually measured in story points, for a certain amount of time or timebox. Velocity is usually measured using a sprint as the amount of time we're measuring for. For some teams, that would mean one week. For others, it means two, three or four. Velocity can be used as a way to plan projects by understanding the capacity of the team, then using their current capacity as a predictor of future capacity. So we're using velocity to estimate how quickly a certain amount of work can be completed.

Velocity is typically expressed in terms of points. For example, if a team has an average velocity of 20 points per sprint and we've estimated the remaining work at 70 points, we probably need four sprints to complete all the work. Since we know how long a sprint is for our team, we can then extrapolate an estimate on how long it will take to finish the project. So from the example, if one sprint is two weeks and the remaining work requires four more sprints, then we need about eight more weeks.

When trying to figure out our team's velocity, we typically look at historical velocity data and take an average. One challenge we may face in trying to determine velocity for the first time, before our team has used this way of measuring capacity, is that it's hard to come up with an initial value. There are a few ways, however, to come up with a velocity at this point. One way is to use historical values of this team's velocity on other projects. However, this requires that the team has worked together before and has used velocity as a measure.

The second way to come up with a velocity estimate is to actually run an iteration with the team. This way, we can start to get a sense of the team's velocity and we can use that velocity to plan the rest of the release. *[A graph displays. The X-axis depicts time and the Y-axis depicts initial velocity.]* Note that teams usually start out with an initial velocity that changes somewhat overtime as they learn to work better together.

[The Velocity Charts graph displays. The X-axis depicts week.] One tool some teams use to keep track of velocity overtime is velocity charts. Velocity charts show a sum of the estimates of work delivered across all iterations to date. Since velocity typically stabilizes overtime, these charts can be a valuable tool for planning projects. In addition to planning, velocity charts are a helpful tool in providing insight into project progress so far for a team.

There are a few things to consider when thinking about velocity. In some organizations, there's a strong focus on continually improving or increasing velocity. This can be dangerous as it can lead to team burnout and overwork, and cause velocity to go up at the expense of bugs and issues, which will ultimately cancel out any velocity gains.

Changes in team structure may cause changes in velocity, since different people have different capacity and team members bring different dynamics to the team. So it's important to note and keep track of changes in team structure and how those changes can impact velocity. Also, technological complexity is not always something teams can predict going into a project. So estimates may change even though velocity has stayed typically the same. Miscommunication can also cause estimates to be inaccurate, and changing customer requirements means things change in terms of the estimates overtime. Estimating a team's velocity is a tool that can help us plan projects and plan release dates. However, in Agile projects, we want to be aware of the considerations that may change velocity estimates and release estimates overtime.

In summary, estimating team velocity is a useful way to plan projects by understanding the capacity of the team. However, when estimating team velocity, there are several considerations to keep in mind. They include too much focus on increasing velocity, changes in team structure, technological complexity is unpredictable, miscommunication, and changing requirements.

Value-based Prioritization

[Topic title: Value-based Prioritization.] There are a number of ways to calculate the expected value that a project might generate for us. You could look at return on investment or ROI, net present value or NPV, internal rate of return or IRR, and compliance.

ROI compares the values of project realized to the value of the investment put into it. When the ROI is positive then the project is profitable. NPV is a capital budgeting technique. A project with a positive NPV is also profitable. IRR is defined as the interest rate that makes the net present value zero. The higher the IRR, the more profitable and thus more desirable. And compliance determines how we are conforming to rules, such as industry specifications, standards, and policies. This also includes regulatory compliance, which relates to adherence to any relevant laws.

Relative prioritization is one way of prioritizing based on user requirements. It involves user-related considerations such as the positive benefit of a feature and the negative impact of its absence. Minimum Viable Product, or MVP, is another value-based concept. MVP considers which features or deliverables are sufficient to satisfy early adopters and that demonstrate enough future benefit. It provides a feedback loop to guide future development.

Another way to look at prioritization is from a customer's perspective. So we might try and look at how much value a customer will perceive, or how much value we expect to generate for a customer through a project. We want to always ensure that we're delivering the highest value to customers as early as possible.

One way to prioritize in terms of the customer's perspective is the 100-point method, which assigns every customer 100 points that they can then distribute amongst the user stories, or the requirements for a project. So let's say we give one of our stakeholders 100 points and we say, assign these to the user stories according to which ones are most important to you. So they may give one of the user stories for example, 50 points, and then distribute the rest of their 50 points amongst five other user stories or two other user stories. This gives us a strong indication that the user story that was given 50 points is really the highest critical or most important user story for that stakeholder.

Another customer value-based prioritization method is called the Kano Analysis. The Kano Analysis compares the degree of implementation of user stories or requirements against the degree of customer satisfaction generated by developing or implementing those user stories. The third method for customer-valued prioritization is Multivoting. With Multivoting, we take a number of ideas that have come up, sometimes as a process of brainstorming, and then each person is allowed to vote on the top one-third of all of those ideas. This will help us to see which third of the ideas are starting to appear as the most important for a majority of our stakeholders.

And lastly, the CARVER Technique categorizes user stories or requirements based on a number of criteria which are criticality, accessibility, return, vulnerability, effect and recognisability. Once we start to put user stories into those categories, we can start to see certain trends or start to prioritize based on those categories.

In summary, there are many value-based prioritization methods, including ROI, NPV, IRR, compliance, relative prioritization, and MVP. Customer-valued prioritization methods can be especially useful in Agile projects. The methods that focus on customer value are the 100-point method, Kano analysis, Multivoting, and the CARVER technique.

Prioritizing User Stories with MoSCoW Model

[Topic title: Prioritizing User Stories with MoSCoW Model.] The MoSCoW method is a tool that's used to help a team prioritize user stories. It's a technique for helping to understand priorities and ensure a common understanding across the different team members.

The MoSCoW method is usually used by analysts and stakeholders when they're coming up with user stories or requirements for a project. It can help determine a clear set of requirements, and a clear priority for those requirements, by categorizing each user story. Once you establish which category each of the user stories falls within, you can determine which requirements are important, which aren't as important, and which can be excluded from the scope of your project.

The first category in MoSCoW is must have, which is the minimum usable subset. These are the users' stories that must be included in order for us to go forward with releasing anything into production. The next category is should have, which are the user stories that aren't critical, but are still deemed important for the project. The third category is could have, which are the user stories that would be useful and would add additional value to the project. And the final category is won't have, which are the user stories that are explicitly excluded from the scope of this project. Sometimes we also think of these as the user stories that we would like to have in the future, but that we know for sure we will not have in this release.

The process for rating user stories using the MoSCoW method is that we assign a priority letter of either M, S, C, or W, to all user stories in our project. Once we've created a user story backlog, whether that's on a whiteboard or a wall, or an Excel spreadsheet. The project team will then work through the map, take a look at the priority that's been assigned to each of the user stories, and identify or discuss any issues that may happen, or may come up in discussion when they're looking at the priority.

So for example, some of the team members may have an opinion about what requirements may constitute a must have, or a should have, but that may be categorized incorrectly as a could have or a won't have. If a team is using the MoSCoW method, we can actually use those categories to help us create our story map.

So for example, we'll use the M-ranked or the must have stories to create our backbone of our story map. The M-ranked user stories are sequenced according to their themes and logical order of development to create that backbone. Once we have the backbone, we'll then take the S-ranked stories, or the should have stories, and place them beneath the must have stories. After that, we'll add the C-ranked and the W-ranked stories, which fall lower and lower on the story map. The W-ranked stories may actually be excluded from the story map if we know for sure that we're not going to develop them. Once we've arranged these in a story map, we can start to work through it and talk about the different issues that may relate to each of the user stories and each of the layers, and discuss potential issues with the team.

Using the MoSCoW method is a way to start thinking about user stories in terms of must have, should have, could have, and won't have, and is a great way to identify the scope of a project, and then create a user story map.

In summary, the MoSCoW method is a tool that's used to help a team prioritize user stories. The four MoSCoW method categories include must have, should have, could have, and won't have.

Tools for Prioritizing Requirements

[Topic title: Tools for Prioritizing Requirements.] Two very useful tools for prioritizing requirements in an Agile project are the Kano model and priority matrices.

The Kano model is a graph-based tool *[The Kano Model displays. It contains four quadrants. The X-axis represents the degree of implementation, such as Not Implemented or Fully Implemented. The Y-axis represents customer satisfaction, such as Disappointed, Not Unhappy, Immediate Happiness, and Delight.]* that plots the degree of implementation of the user stories or requirements, against the degree of customer satisfaction generated by developing or implementing those user stories. Priority matrices use a two by two matrix to compare two variables to one another.

The Kano model is generated by surveying end users and customers about how they feel about the implementation or absence of certain features or requirements. By creating the Kano model graph, we're able to determine which requirements are high priority, or have high importance to customers. A delighter is a feature that wasn't expected, and it is new or exciting to the customer. *[The delighter curve is drawn above X-axis which passes through the two quadrants above X-axis.]* Delighters are high value features. *[An upward-sloping satisfier line is drawn which passes through the origin.]* Satisfiers are features that bring value to the customer when they are implemented. *[The dissatisfier curve is drawn below X-axis which passes through the two quadrants below X-axis.]* Dissatisfiers are those features which are basic expectations of customers. If these features are missing, the customer will be dissatisfied. Non-essentials make no difference to the customer, whether they are present or not. So those would be a waste of time and resources if we were to develop them.

When we look at customer satisfaction as a measure, we're able to determine the most efficient and effective ways to satisfy the customer directly. And how to minimize production costs by not pursuing the development of some requirements that might not drive customer satisfaction.

Priority matrices are a sophisticated prioritization method that we use to weigh various prioritization factors against each other. Factors may include the value that a certain user story is expected to generate, the cost that it will take in order to develop that user story. The risk involved in implementing or not implementing a certain user story, and the releasability of a user story or feature.

We create a two by two matrix, which is a tool used to compare the two variables to one another. The different quadrants of a priority matrix are high importance and high urgency, which is the top or right quadrant, high

importance, low urgency, which is the top left quadrant. High urgency, low importance, which is the bottom right quadrant, and low urgency, low importance, which is the bottom left quadrant. User stories that are high in value, cost, and risk are prioritized.

In summary, the Kano model and priority matrices are two valuable tools for prioritizing requirements. The Kano model is a graph-based tool that compares degree of requirements implementation against degree of customer satisfaction. It is generated though surveys of end users and customers. The model differentiates priorities, identifies non-essential priorities, and focuses on prioritizing requirements that will result in highest customer satisfaction.

Priority matrices are created using a two by two matrix, and are used to compare any two of four factors against each other. The four factors are value, cost, risk, and releasability. Each requirement is placed into one of four quadrants of classification, and the focus is on prioritizing user stories that are highest in value, cost, and risk.

Completing the Release Plan

[Topic title: Completing the Release Plan.] When completing your release plan for your Agile project, two activities you'll most likely need to do include splitting and combining stories, and assigning user stories to iterations.

A good guideline for user stories is that they should satisfy the INVEST criteria, which stands for Independent, Negotiable, Valuable, Estimable, Small, and Testable. Sometimes user stories don't fulfill that criteria in that they are either too large or too small. And by the way, you might also hear the word estimatable. Either way, it simply means, able to be estimated.

Smaller stories can be combined to create larger stories, which is a relatively easy process. On the other hand, larger stories can also be split into smaller ones, but this process requires more consideration and effort on how to split those user stories. The goal, ultimately, is to have stories that are small enough so that we can get each story done within a matter of days. Ideally, no user story is so large that it requires the whole sprint to complete.

There are a number of ways to split user stories. The split could be based on operations performed within the story. For example, the story could be split into separate CRUD operations, Create, Read, Update, and Delete. Another technique is to consider cross-cutting concerns such as security, logging, and error-handling. The split could then be made into one story version with support for cross-cutting concerns, and one without support. The split could also be based on functional and non-functional aspects of the story. For example, non-functional aspects would include performance, stability, and speed.

Finally, the split could be made into smaller stories if different aspects of a large story have different priorities. In some cases, we may have user stories that are too small. And typically, we know that they're small when a developer says that it will take them longer to estimate a story than it will to actually execute it. Stories that are too small can be combined into a larger related story. Combining stories that are related will allow them to be better prioritized. The combined user story should be estimated as a whole, rather than summing the estimates of its individual parts. One example of combining stories is combining multiple bug reports.

Once user stories meet INVEST criteria, they can be assigned to iterations. An iteration is a single development cycle that is anywhere between one and four weeks. In iteration planning, we update and elaborate on the release plan, which is a key step in Agile development. We create the release plan at a high-level, but then we want to start translating those initiatives or those conceptual user stories into specific tasks. The objective of iteration planning is to generate the level of detail necessary to drive the work for a particular iteration. This helps development team members to understand how to specifically start executing on the user stories.

When we're doing iteration planning, we determine the amount of work that a team can complete. *[A graph displays. The X-axis depicts Iteration in Days and the Y-axis depicts Work Remaining in Hours.]* At this point,

we can get an understanding of team velocity. And by knowing iteration length, we have a way of estimating how much work we can complete after we've broken down user stories into tasks.

We also update the list of work for the release. Once we've done the process of breaking user stories down into tasks, sometimes we find out that there are new user stories or new tasks for the development team that we need to add to the release plan. We also want to set a goal for the iteration at this point. That goal should maintain focus as the development work proceeds. So we may have a single theme or a goal that we want to achieve within an iteration, even though a number of user stories may cover that goal. We're also selecting the functionality to develop, which has been determined in the release planning stage.

So as we've talked about before, when we're doing release planning, we may have to split user stories into different categories, and then, in our iteration planning, we're taking a certain category of those user stories and developing them.

The work required during iteration planning is typically managed in an iteration planning meeting. So in the iteration planning meeting, typically the product owner, who is considered the customer representative, presents the user stories to the development team. This is a key information transfer point between the product owner and the development team, as it provides the development team with one-on-one time with the product owner to ask questions about how they expect something to be developed, or their acceptance criteria, and to get into a little bit more detail on requirements as needed.

Also, during the iteration planning meeting, the development team translates user stories into specific development tasks, which make execution much more manageable.

Once user stories are translated into development tasks, they can be assigned to appropriate members of the development team. And with a good understanding of both the time commitment required, as well as the sprint length, commitment from the team is also confirmed. This requires developers to confirm their commitment to completing the assigned tasks, and helps ensure that they have manageable workloads.

In summary, splitting and combining user stories is required when stories don't meet INVEST criteria and are typically too large or too small. Once user stories meet INVEST criteria, they can be assigned to iterations. This is typically done at an iteration planning meeting, where user stories are presented, translated into specific development tasks, those tasks get assigned to iterations, and then, the development team confirms commitment to completing the assigned tasks for a planned iteration.