Module 3: Writing Requirements

Upon completion of this module, you should be able to:

- Describe requirements within the 12 Agile principles.
- Create a user story and recognize the format of a user story.
- Assess what makes a good user story.
- Recognize when a user story is too large (in other words, you should be able to recognize an epic user story), and why this is likely to occur in the specification phase.
- Create an acceptance test for user stories based on acceptance criteria.
- Explain what a product backlog is, and help prioritize user stories.
- Describe what product backlog is and how it works within Scrum.
- Create a story map, and identify the benefits of using a story map.
- Identify any missing or inconsistent user stories within a story map.

As we just saw, requirements can be expressed through techniques such as use cases, wireframes, and storyboards, but they can also be expressed through many written formats.

Agile Requirements

Client needs and requirements may be expressed within the philosophy of Agile Software Development. There are 12 principles of Agile Software Development, as explained in the **Introduction to Software Product Management** course and the **Software Processes and Agile Practices** course. These principles support the belief in Agile that software is dynamic. In other words, in Agile, software cannot be simply defined once and then created off that singular definition—it is constantly evolving.

In concurrence with this philosophy, clients often change their minds about product functionality. This can happen even in the middle of development. In Agile software development, however, changing requirements should be welcomed and expected. The ability to welcome changing requirements can make the difference between the success or failure of a project. Development teams should understand that although requirements are created with full intentions of following through, requirements also inevitably change.

12 Agile Principles

Early and Continuous Delivery

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Working Prototypes as Progress

Working software is the primary measure of progress.

Technical Excellence & Good Design

Continuous attention to technical excellence and good design enhances agility.

Focus on Simplicity

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

Self Organizing Teams

The best architectures, requirements, and designs emerge from teams.

Encourage Face to Face Interaction

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Deliver Frequently

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Welcome Changing Requirements

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Sustainable Development

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

Build Projects around Motivated People

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

Daily Collaboration

Business people and developers must work together daily throughout the project.

Reflect on Team Behaviour

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

For more information please visit:
http://agilemanifesto.org

According to Agile, face-to-face interactions with clients is key to ensuring requirements are right, even as they change. Requirements should be elicited in an open and collaborative setting, as discussed in **Eliciting Requirements**. Although the vision of the product should come from the client, software product managers often help narrow the vision and outline what requirements are within scope of the project. These meetings are also an opportunity for discussion of the inevitable changes in requirements. As explained in the **Software Process and Agile Practices** course, this is frequently practiced in Scrum.

Collaborative discussions with the client help to ensure that project requirements are the best possible, and they set the precedent for communication.

Requirements are an integral part of Agile. In turn, if Agile is followed in creating requirements, quality is enhanced.

User Stories

User stories are a major technique used to express requirements, like use cases, wireframes, and storyboards. User stories are special because they use a consistent format to express requirements that is easy to write, read, and evaluate.

User stories take the following format:

> "As a '**who**,' I want to '**what**,' so that '**why**.'"

The "**who**" of the requirement is the stakeholder role for whom the requirement is being formed. The requirement should be written as if it is from this person's point of view.

The "**what**" of the requirement is the specific task or functionality the stakeholder wants to achieve by using the product.

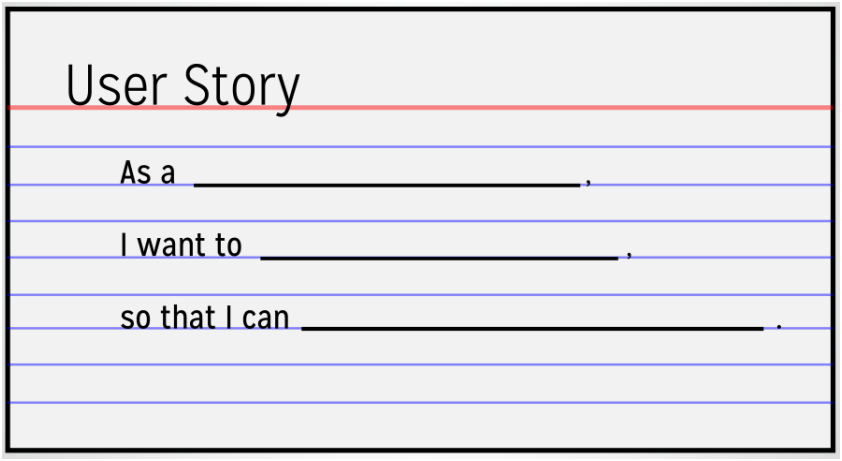
The "**why**" of the requirement highlights the goals or visions of the product, and it provides insight into the value or benefit of the requirement.

An example of a user story based on the restaurant example previously used in this course could be, "*As a customer, I want to be able to identify dietary restrictions, so that I know I can eat the food I order*."

User stories are useful, as they provide a clear and structured way to express a requirement that also does not use too many technical details. Unlike freely formed requirements, user stories ensure the "who," "what," and "why" of a requirement are always accounted for. User stories are also useful because they are short and can be easily written on physical index cards or sticky notes. This practice allows easy organization of requirements, which can be re-organized when requirements change.

Classically, user stories were written on index cards. The front of the card contained the user story, and the back held the acceptance criteria, which we'll cover later. With software build to help track software development progress, user stories are no longer constrained by the size of the card. However, when creating your user stories, keep in mind that shorter is usually better, as that forces you to make your requirements understandable and actionable.

If you have any doubts about the size of a story, try writing it on an index card in the format below. If you have trouble getting your ideas to fit, try re-wording things to be more clear and concise.



Although clients should write user stories, as they know best what they want in a product, software product managers often write or help write user stories because they are more experienced in doing so.

User stories can be evaluated using the mnemonic **INVEST**. A good user story outlines a specific software requirement in the product.

I	**Independent** This means the requirement can exist outside of other user stories and still be meaningful. This allows for requirements and their user stories to be freely re-arranged if necessary.
N	**Negotiable** User stories should also be general enough for the development team and client to work around their implementation. They should not focus on specific technical details. Instead, focus should be on the most important aspects of requirements, while remembering that these could change.
V	**Valuable** User stories should bring value to the client.
E	**Estimatable** It should be possible to estimate how much time it would take to design and implement the requirement in the user story.

S	**Small** A user story should be small because it is meant to be developed in a short time period. If the time to design and implement a requirement is uncertain, the user story is likely too big, so it should be broken down into smaller, manageable ones.
T	**Testable** User stories should be verifiable against a set of criteria in order to determine if it is "done", meaning that the user story has accomplished what it set out to do, and does not need further work. This is usually accomplished with acceptance tests.

If a user story contains descriptions that are too vague or broad, and if it is difficult to estimate how long it will take or how it can be done, it is probably an **epic user story**. Epic user stories usually occur at the beginning of a project when the product is still developing and may not yet have definite form. This is due to the pattern known as the "**cone of uncertainty,**" which suggests that the time estimates to develop a user story become less accurate the further into the future the feature is intended to be developed. Even experienced development teams encounter epic user stories.

In order to avoid writing epic user stories, it is important to be able to identify when one has been created. If an epic has been identified, it can be broken down into smaller stories, which can be estimated. A good strategy to avoid epic user stories is to provide just enough information for a developer to understand how to implement it, but not so much information that implementation details become part of the story.

Example of an epic user story:

"As a customer, I want to pay for my bill, so I can settle what I owe quickly."

This user story could be broken down into smaller ones, such as:

- "As a customer, I want to be able to see a bill, with all of the items in that order, so I can see how much my order will cost."
- "As a customer, I want to be able to select a "pay now" option when I view my bill, so I can pay the bill immediately."
- "As a customer, I want to be able to enter my payment details for VISA and MasterCard credit cards, so I can pay using a convenient method."

Acceptance Tests

An **acceptance test** is used to verify whether or not the requirements of a user story have been completed. Acceptance tests are often used in the testable part of the mnemonic INVEST. A user story is considered satisfied if the test is passed.

Acceptance tests are evaluated based on a set of **acceptance criteria**. Acceptance criteria are simple with specific conditions used to check if a user story has been implemented correctly.

Clear, straightforward language should be used in creating acceptance tests.

Acceptance criteria are usually determined by the client's specific needs. In order to turn acceptance criteria into an acceptance test, it is helpful to go through the steps of the criteria.

Each test should assess one verifiable condition that makes up one small part of the user story. If all tests are passed, then the acceptance criterion is passed. If all acceptance tests are passed for all of the user story's acceptance criteria, then the user story is considered successfully tested.

Building on the restaurant example used throughout this course, examples of acceptance criteria for the user story "As a customer, I want to be able to enter my payment details using VISA and MasterCard credit cards so I can pay using a convenient method" include:

- Payment can be made using a VISA credit card.
- Payment can be made using a MasterCard credit card.
- Payment can be made using an online financial service.
- When paying with a credit card, filling in the "card number" field auto-detects the card type.
- The customer sees only the relevant input fields, depending on the selected payment method.

To turn a criterion into an acceptance test, it is helpful to create a few steps. For example, for "Payment can be made using a VISA credit card," tests could be:

- Insert a VISA card into a chip reader.
- Enter the VISA's PIN number.
- Confirm the payment was accepted.

Going through these steps verifies the acceptance criterion.

Acceptance tests have many other added benefits. They allow the development team to view requirements from user perspectives. They also help with determining functionality of the product, as they provide details not present in the user story. These details can help outline developer tasks and how those might be finished. Acceptance tests can also help avoid epic user stories. An excessive number of acceptance criteria suggests that a user story should be broken up into smaller pieces.

Like user stories, acceptance tests should be developed by the client with help from the software product manager and development team.

Product Backlog

A **product backlog** is a set or list of software features that the software product manager and development team plan to develop for a product over the course of a project. They are a means of organizing work, prioritizing tasks within the project, and planning those priorities. Product backlogs are a popular technique because they provide a lot of flexibility for both clients and development teams. They are critical to Scrum and therefore Agile.

Features in product backlogs include work tasks (physical jobs that must be done on the project but are not necessarily related to developing product features), knowledge tasks (work for parts of the project that need to be learned), bugs (errors in product code), and most commonly, user stories. All tasks in product backlogs tend to become more refined over time.

In order to create a product backlog, after user stories are created, they should be placed in a list. Each user story should also be assigned a unique identifier. Identifiers can be as simple as sequential numbers.

The next step in creating a product backlog is to prioritize the user stories and features. The process of prioritization helps clients identify their needs and wants. It also gives developers perspective and focus by highlighting what is most important in a project. Prioritization also helps determine what features can feasibly be accomplished in a project with technology and given resources.

As prioritization is important for the entire team, prioritizing user stories in product backlog is best done through discussions between clients, the software product manager, and the development team. Why a user story is important will be clearer to everyone through such discussions, as well as why an effort estimate has value.

Using the product backlog, the development team can start to plan the project using priorities as reference points (for more information on this, see the **Agile Planning and Software Products** course). User stories from product backlogs are grouped together into units of work to be done at certain time intervals. In Scrum, these intervals are known as **sprints**. The most important features from product backlogs should be finished earlier, while less important ones can be finished later.

Product backlogs are very flexible in light of this kind of development. Scrum focuses on developing one sprint at a time. Although the sprint in development should not change, sprints beyond the one being worked on are able to change. At the end of a sprint, clients can evaluate the work done and add new user stories or requirements for the product, or change the priority of existing user stories or requirements. The next sprint can be adjusted accordingly.

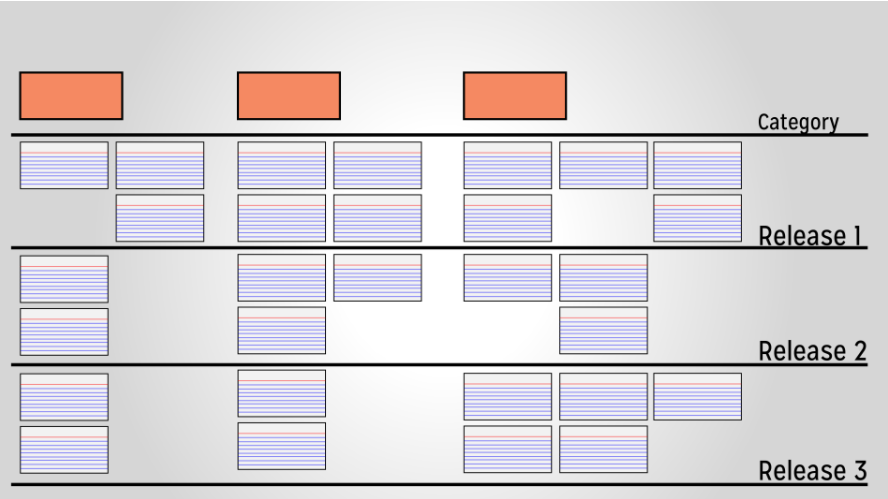
Product backlogs will therefore change over time and can become smaller, bigger, or have changed orders. Such adaptability and versatility is encouraged in Agile methods.

Story Maps

Story maps are used to organize requirements and help structure a project. Story maps support change on a higher level than even product backlogs. They present product backlogs in a visual manner, with user stories grouped into specific functional categories. By covering and prioritizing user stories across multiple categories, story maps provide holistic views of the product being developed.

Story maps have a specific structure. They are created as sets of columns. Each column represents a category for grouping user stories together. Within each column, user stories are prioritized from most to least important. This structure allows the client, software product manager, and development team to see the highest priority user stories for an entire project. While a list of requirements with the structure of a product backlog might be overwhelming, story maps turn such lists into manageable, organized sets of features to be implemented over the course of a project.

Below is an example of what a story map structure might look like.



Story maps have many benefits, including:

- Simplifying prioritization of user stories
- Giving product backlogs a trackable, visual feel
- Giving perspective on how user stories may relate to one another
- Helping identify what might be missing from each category
- Providing context for developers by showing that other simple functionality may best be implemented before more complex work
- Providing a holistic view of the product by not allowing focus on one category only, but instead emphasizing the multiple functionalities (or categories) of most products
- Giving a better understanding of how the product will develop and fit together over stages—it is possible to see how a product will evolve row by row, which could save time over the course of the project.

Story maps are a good example of the Agile principle of building working software because of its emphasis on multiple functionalities.

It is important to remember that a story map is not a Kanban board. A Kanban board, as discussed in **Software Process and Agile Practices**, is a visual tool used to display the project's current state and to keep track of progress of project tasks. In contrast, a story map is used only as a tool to plan and organize user stories. Story maps, however, are not full-fledged development plans either.