Client Needs and Software Requirements

COURSE NOTES

Table of Contents

Upon completion of this course, you should:

(a) be able to explain and create clear requirements in order to drive effective software development.
(b) understand different types of requirements, and how to properly adapt to changes in product requirements.
(c) be able to visualize client requirements using low-fidelity prototypes such as wireframes and storyboards.
(d) be able to identify and characterize different kinds of users and their needs.
(e) be able to maximize the effectiveness of client interactions.
(f) be able to express requirements with the help of tools such as user stories, acceptance tests, product backlog, and story maps.
(g) know how to assess requirements for quality and clarity.

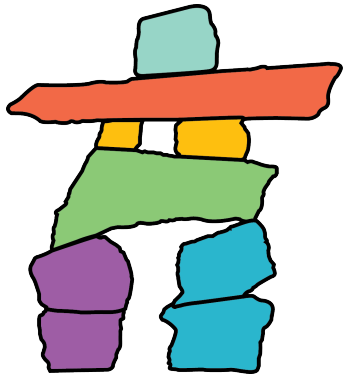
UNIVERSITY OF ALBERTA

Module 1: Introduction to Requirements

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

(a) describe and recognize what a requirement is,
(b) describe the types of activities associated with requirements: *elicitation, expression, prioritization, analysis, management*,
(c) tell the difference between the different types of requirements: business requirements, business rules, user requirements, functional requirements, non-functional requirements, external interfaces, physical settings, and developer constraints,
(d) recognize that changes are common to requirements,
(e) describe the concepts of vision, and scope,
(f) describe and defend against scope creep,
(g) understand the concept of managing expectations, and
(h) understand the difference between design and requirements.

Introduction



Welcome to the Client Needs and Software Requirements course, within the Software Product Management Specialization. This course will provide fundamental elements to the specialization and will serve as one "leg" in the inukshuk that depicts the structure of our specialization.

In the introductory course, you learned that making better software involves three goals. These goals are:

- the right product
- done right
- managed right

The *right product* means meeting the needs of your clients and end-users and not just their wants. This involves understanding the problem they need to solve, and the tasks they need to accomplish with the software.

To achieve *done right* and *managed right*, software development must start with a quality set of software requirements, which are later planned, designed, implemented, and tested.

In this course, you will learn how to elicit needs from your clients and end-users. You will also learn how to express these needs as a quality set of requirements to initiate software development and planning activities.

There are four modules in this course.

Module 1 will cover:

- types of software requirements
- how to handle changes to software requirements
- how to avoid scope creep
- the fuzzy boundary between requirements and design

Module 2 will cover:

- how to outline client and end-user needs
- the role of use cases in designing a product
- the visual design techniques of wireframes and storyboards

Module 3 will cover:

- how requirements become more functional within an Agile framework
- the use of user stories and acceptance tests to express and verify requirements
- the use of a product backlog to prioritize requirements
- creating story maps to organize requirements

Module 4 will cover:

- criteria for quality requirements
- how to make requirements as clear as possible

The techniques you learn through this course will help you create a better software product suited to the needs of your clients and end-users.

What is a Requirement?

One of the most critical aspects of Software Product Management deal with conditions known as **requirements**. Many definitions have been developed to describe "requirements." A requirement is most easily understood as a specific description of your client's needs, which can be used to help create a real-world product.

The Institute of Electrical and Electronics Engineers (IEEE) defines a requirement as:

> (1) A condition or capability needed by a user to solve a problem or achieve an objective.
>
> (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
>
> (3) A documented representation of a condition or capability as in (1) or (2). (IEEE, 1990).

DID YOU KNOW?

The Institute of Electrical and Electronics Engineers (IEEE) is an international professional association of engineers, scientists, software developers, and information technology professionals, among others.

The IEEE is well known for developing standards in both the computer and electronics industry.

Well-defined requirements help make sure client needs are understood and addressed and also help detect potential problems before they become large and expensive to fix.

Requirements Activities

In the previous course, **Software Process and Agile Practices**, software development is an organized process, featuring a series of phases. Grouped within each phase is a series of one or more activities—sets of related tasks. Requirements are established through activities at the beginning of a project. It determines the purpose of a product and how the product can meet that purpose.

This course will explore five important requirements activities:

1. eliciting requirements
2. expressing requirements
3. prioritizing requirements
4. analyzing requirements
5. managing requirements

With the correct activities, the right requirements can be established, written in proper form, and even improved.

Eliciting Requirements

The activity of **eliciting requirements** is an interactive and investigative process, which occurs when meeting with the client and users.

Clients often have ideas about what features they would like in a product and what these features should look like. Many clients, however, have limited knowledge of how software is built and vague ideas regarding what makes a project

successful. It can be difficult then for clients to understand what they truly require in a product. It is the role of the software product manager to help the client figure out what they "want" and what they "need."

A "want" is usually a functionality or feature that the client would like the product to have, which may add value, but is not necessarily core to the product itself. A "need," on the other hand, is a core functionality that the product must have in order to fulfill the product's purpose. Needs should take priority in product development.

The best way to discover and develop "needs" and "wants" with your client is through eliciting requirements, where you engage in discussion about the product with your client. Through discussion, the software product manager and team can help provide insights on what the client "needs" and "wants" the product to do, and how these goals can be feasibly achieved. It may be that the initial vision the client had of the product will have changed, but through involvement with the software product team, any changes should feel proactive and the client should feel reassured that the team understands users' needs.

Note that eliciting requirements as "needs" and "wants" does not necessarily mean that all the client's features and ideas that fall in the "want" category are not doable or should be dismissed. In fact, these ideas may be very good. But it is the role of the software product manager to make sure that the goals are feasible, client expectations are realistic, and the product produced is the best possible result.

Eliciting requirements should not be confused with "requirements gathering." "Requirements gathering" is the more passive approach of simply asking the client what they would like done, and it often puts the development team in a reactive process. Eliciting requirements, however, engages in in-depth discussion and collaboration from the start of product development, so both the client and the development team work together to build a successful product.

Expressing Requirements

Once client needs have been established through eliciting requirements, the activity of expressing requirements comes into play. **Expressing requirements** involves framing the requirements identified through discussion in a way that allows a product to be built.

Often, requirements are first described through notes from meeting with clients. From there, better and more concrete representations can be used for expressing requirements. Typical representations include **use cases**, **user stories**, or **storyboards**, among others. These are often tailored to the project; they could be simple or complex, textual or visual. It is up to the software product manager and team to determine and use representations that would work best for the project at hand.

Prioritizing Requirements

Once a vision of what needs to be done for the project has been established through both eliciting and expressing requirements, it is important to **prioritize** client needs, especially in Scrum methodology.

Questions to help establish priorities include:

- What requirements must be done for the project and product to be successful?
- What requirements should be done? In other words, what is important but is not as time-critical or could be satisfied another way or at a later time on the project?
- What could be done to improve the project or product but is not necessary? These priorities are usually only included if both time and resources allow for it.

DID YOU KNOW?

The questions outlined above closely follow the MoSCoW method of prioritization, developed by Dai Clegg. This method is used to help reach an understanding with clients on the importance of each requirement. “MoSCoW” is an acronym for the categories of “Must have”, “Should have”, “Could have” and “Would like but won’t get.” By asking the questions suggested here, requirements can be placed in these categories.

Analyzing Requirements

The process of examining the listed requirements of a project to ensure that they are clear, complete, and consistent is known as **analyzing requirements**. Analyzing requirements helps ensure that the product is the best one possible. It is an important process, and a constant one. A project must be continually evaluated and requirements improved as it progresses. This adaptive nature is important in Agile systems.

Analyzing requirements helps to resolve any potential conflicting requirements or to identify problems between requirements that might not be easily seen at first glance. It also ensures that the requirements identified truly reflect and relate to the product being built.

Managing Requirements

The activity of **managing requirements** is also a continuous process. It involves the organizing and re-organizing of requirements and possibly reusing subsets of requirements in different stages. It also involves keeping track of priorities, analyses, and changes in requirements. This is very important because everything is connected in a project. If something changes in one requirement, it will affect other requirements and the development of the product.

Managing requirements also means ensuring that the identified requirements are central to the many processes of product creation, including coding, testing, and change logs.

Types of Requirements

Building on understanding requirements and their associated activities, it is possible to consider different types of requirements.

This course will cover the following types of requirements:

- business requirements
- business rules
- user requirements
- functional requirements
- non-functional requirements
- external interfaces
- physical product settings
- development constraints

Business requirements and business rules can influence the project, while user requirements, functional requirements, and non-functional requirements are considered core requirements. Finally, external interfaces, physical product settings, and development constraints are requirements that add context for design and implementation of the product.

Business Requirements

There are many possible definitions for business requirements. In this course, **business requirements** refer to those requirements that involve the purpose of the project. In business requirements, goals are presented in concrete and specific terms. These goals are usually tangible or quantifiable business values that can be used by business analysts.

An example of a business requirement might be: "The client needs to reduce errors in orders made to their company by 25% by the end of next year to raise their yearly revenue by $10,000."

Business Rules

Business requirements should not be confused with **business rules**, although they are often associated. Business requirements deal with why the project was pursued, while business rules are constraints on how the product will function. Business rules are sometimes necessary to make a project appropriate or successful. They are often budgets, policies,

guidelines, or regulations.

Examples of business rules include:

- government or legal regulations
- privacy policies
- brand uniformity requirements

User Requirements

Users or **end-users** are the people who will use the software once it has been created. **User requirements** are the tasks that these users can accomplish with the product, or what the product can do for the user. User requirements are very important requirements to the project, if not the most important. They are part of the core functionality of the product. For this reason, determining user requirements usually end up being very time consuming.

There are many ways to express user requirements. These include:

- use cases
- user stories
- storyboards
- scenarios

Use cases, user stories, and storyboards are explored in more depth below.

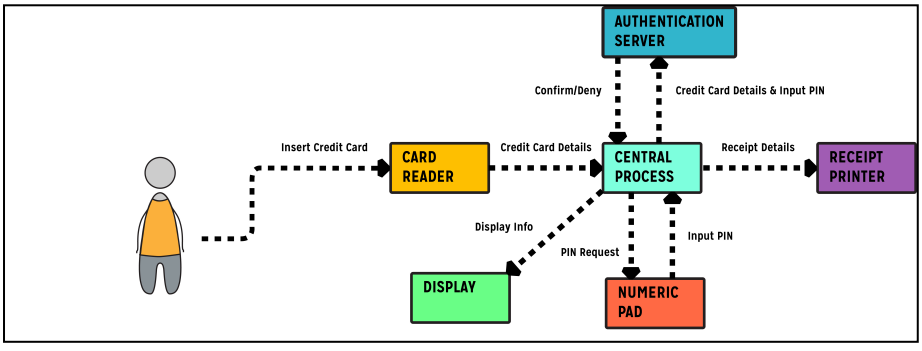
Scenarios refer to cases where the client or end-user describes user requirements in their own words. These descriptions can sometimes be vague or non-specific. It is part of the role of the software product manager to help organize and refine their needs in more concrete manners, so the development team knows how to best build the product.

Functional Requirements

Functional requirements are those behaviours that the developed product should do or support. These are usually expressed as inputs into the product, outputs of the product, or a description of the behaviour itself. For example, input might be data related to a user's online purchase, including name, address, item purchased, and payment information. The output in this scenario might be a notification email after the transaction is performed.

Often, functional requirements need to be expressed with depth and specificity. **Information flow diagrams** are a graphical technique commonly used to show how data flows throughout the system and the dependencies of all the system components. Together, an information flow diagram shows how the entire system functions.

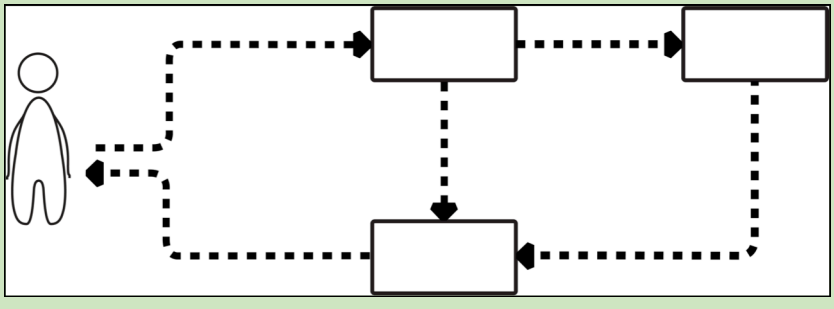
The following is an example of an information flow diagram for a customer using a credit card with a product, and where the credit card information runs before a receipt is generated for the user.



Information flow diagrams or **data flow diagrams** are visual techniques used to show how information or data moves through a system. The information may change as it moves from input to output of the system. These diagrams are useful for show how users can interact with the product.

Information flow diagrams can be partitioned into levels that show progressive flow and increased detail. Information flow diagrams are valuable tools because they can describe the system process without relying on details related to computer systems.

Below is an example of the format of an information flow diagram might take.



Non-functional Requirements

In addition to functional requirements, there are **non-functional requirements** that describe how well a product must perform. They are also known as quality requirements for this reason. Non-functional requirements address product issues including accuracy, dependability, security, usability, efficiency, performance, and maintainability.

Drawing on the example used in function requirements, a non-functional requirement in the same scenario would be that emails should be delivered to users within two hours of purchase. Non-functional requirements are usually complementary to functional requirements.

External Interfaces

External interfaces requirements refer to those requirements related to how the product is situated within a larger system. In other words, external interfaces do not concern themselves with the physical environment of the product, but rather it is concerned with the relationship of the product to other entities outside the product.

For example, a software application that retrieved information from a remote database to display to users sits between the entities of the database and the end-user. An external interface is used between each one. Interfaces also involve the way connections are made through media, protocols, formats, and levels of compatibility.

External interfaces can be represented within data flow diagrams that show all the components of a product, including outside entities. These data flow diagrams make explicit reference to the flow of data from other entities and the product within an entire system.

Physical Setting

Physical setting requirements refer to how the product needs to be designed in order to function in its physical environment. For example, if a product was being developed for underwater mapping, it may need to be waterproof. Similarly, if the product was designed for use in the desert, or in Antarctica, it would need to withstand and function in those environments.

Development Constraints

Development constraints affect everything from implementation technology, conventions, documentation, and the process used to develop the product. They generally refer to constraints related to creating the product, such as which devices or platforms will be supported, or how much memory, bandwidth, or processing power the product is limited to using.

It is beneficial to address development constraints as late as possible in the specification phase because technology changes rapidly.

Changing Requirements, Controlling Scope

When a product is designed, it is guided by a principle known as product vision. The **product vision** is what outlines the value of a product to the client and its place within the wider market. It relates to the purpose of the product, as well as the problem or need the product sets out to solve. Any changes that occur in the project should fall within the vision of the product. In other words, changes to the project should not change the purpose of the product.

Visions can become problematic if they are not realistic. **Scope** is defined as what a project can realistically achieve. It is an important aspect of the project that has the power to greatly affect development teams and product managers.

American software engineer, consultant, and author Karl Wiegers has defined both vision and scope. The definitions are:

> **Vision** is a "long-term strategic concept of the ultimate purpose and form of a new system." (Wiegers, 2012, p. 1)
>
> **Scope** is what "draws the boundary between what's in and what's out for the project." (Wiegers, 2012, p. 1)

We can see that the two concepts are interrelated and why scope is so important to project development. Defining the scope ensures that the project undertaken is not only possible but also successful.

Defining scope is an important part of the project, and it is best done during the requirements elicitation activity of the project. It is also important to manage client expectations during this process and not to over-promise what the development team

can realistically deliver in the product. A good strategy for managing expectations is to identify what will not be included in a product, so client expectations are clear.

Scope creep happens when the scope of a project grows as the result of an increase or change in requirements. Consequently, the likelihood of project success usually significantly lowers over time.

For example, the vision of a project could be designing a game to teach cardiopulmonary resuscitation (CPR) to users. A change in scope could involve switching the platform on which the game is designed once development has already started. This change could drastically affect the success of the project.

Changes are common to software requirements and need to be accounted for. Strategies for defending against scope creep include:

- Making expectations clear between client and team.
- Drawing the scope with the client through tools such as use case diagrams.
- Asking the client to prioritize requirements so that the most important ones can be developed early.
- Asking "Is this in scope?" when refining requirements to avoid spending excess time working on unnecessary requirements or capabilities better suited to later releases.

Providing concrete time frames for developing requirements with clear beginning and end dates also helps determine whether the requirement is realistically doable and within the scope of the project. Estimating time frames is a skill that software product managers refine and improve with time.

A core principle of the Agile manifesto is evaluating each proposed change on a case-by-case basis and how that change might affect the project. Changes should be introduced to increase the project's business success. Any accepted changes should also entail revisions in estimates and plans for product development.

Requirements and Design

In Agile, sometimes the lines between forming requirements and designing a product become blurred. Requirements focus on what the product is designed to do, while **design** focuses on how the product can satisfy user needs.

It is common for both requirements and design to be worked on simultaneously, and requirements could be formed with elements of design in mind. The use of techniques such as product drawings can also help identify whether requirements fit together as a whole and are correct, consistent, and clear. Often, clients will also think of their product and focus on desired design characteristics. It is part of the role of the software product manager to help the client understand what their true problems or needs are in that case.

The software product manager must also be careful not to impose unnecessary constraints on the product by blurring requirements and design. For example, an aspect of design such as the coding language or interface features could potentially limit a product, and it should not be confused with being a requirement.

Strategies for avoiding problems in blurring requirements and design include asking the following questions:

- Is the solution just a possible option?
- Is the solution the only one possible?
- Is the solution addressing the wrong problem?
- Is the solution just to attract developer interest?
- Is the client more “solution focused”?

Module 2: Client Interactions

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

(a) describe who a user is, and tell the difference between primary, secondary, and tertiary users, including stakeholders.
(b) explain strategies for involving clients in software development, including:
    a. managing client expectations,
    b. the kinds of questions to ask to discover and refine requirements and needs,
    c. the benefits of using a glossary.
(c) create a use case description, including: name, actors, goal, trigger, pre-condition, post-condition, basic flow, and exceptions.
(d) create a wireframe and recognize the requirements it expresses.
(e) create a storyboard and recognize the requirements it expresses.

User Considerations

When designing a product, one of the most important things to take into consideration is the end-user. **End-users** are the people who will use the product. They are among the stakeholders of the project.

A **stakeholder** is anyone affected by or who has an effect on the success of the project, such as end-users, clients, managers of end-users, and system administrators. A successful project addresses the needs of all stakeholders.

There are three types of stakeholders or users: primary users, secondary users, and tertiary users.

Primary users are end-users, or the people who will use the product. **Secondary users** are those people who will occasionally use the product or use it through an intermediary. They may not be the target audience, but they could be related to the target audience in some way, such as parents of children who use a product designed for children. **Tertiary users** are those who are affected by the use of the product or make decisions about the product, such as clients.

A product should be designed to be something users can navigate and want to use. This is primarily accomplished through good **user interface (UI)** design. UI is what is seen when using the product, and it can encompass anything an end-user interacts with—features such as windows, buttons, scrollbars, checkboxes, and text boxes. Good UI design is important. If there are many similar products on the market, users will easily move on to another product if they do not like what they are currently using. An entire discipline known as **human computer interaction (HCI)** studies how end-users interact with technology products. More information on HCI is available in the course resources.

A number of issues that may arise in considering user interactions include:

- users have an inability to express what they need
- users are biased by previous experiences; e.g., a user is more likely to use a bad interface that they are familiar with rather than a new one, even if the newer one is actually a better product

- developers sometimes have trouble seeing through a user's point of view because of their advanced knowledge of technology

Creating an intuitive, user-friendly interface is key to addressing many of these issues. A good strategy to consider is to design a product for both beginner users and expert users. In general, the design will then accommodate intermediate users as well.

When designing software, it is also important to consider the numerous limitations users are faced with. These limitations are related to human limitations. They include:

- **perceptual or sensory limitations**, which are caused by restrictions of the five senses. Colour blindness is an example of a sensory limitation.

- **physical limitations**, which affect how a user physically interacts with or uses a product. An example is left- or right-handedness.

- **cognitive or memory limitations**, where people can only remember so many things at once, so it is important to use visuals in design that are familiar or suggestive to help identification.

- **cultural limitations**, which encompass how different cultural backgrounds of potential users can affect interpretation of design elements, such as symbols and icons, layout, multimedia, and translation needs.

More information on cultural limitations is available in the course resources.

DID YOU KNOW?

George A. Miller, one of the founders of the field of cognitive psychology, discovered that humans have an average limit for short-term memory capacity. People can remember between five and nine things, with seven being the average. Distractions limit memory even more.

Good product designers understand this memory limitation and work with metaphors and symbols that users will be familiar with, so there is less for them to remember!

Involving Clients and Users

In **Eliciting Requirements**, the importance of involving clients was emphasized. A good software product manager and development team do not simply gather ideas from clients but actively collaborate with clients. Consequently, client ideas may change from the outset of discussions, but with input from both the team and the client, the best possible product can be made.

Requirements can also come from other sources, including:

- interviews with end-users
- feasibility studies with focus groups
- observing how end-users use the product
- consulting previous products

Involving clients in collaborative discussion over requirements, usually in face-to-face meetings, helps manage client expectations. By being involved in discussion, clients are aware of realistic goals and timelines, what will and will not be done in the product, and subsequently they can prioritize requirements.

Below are some key points to remember when interacting with clients.

Try to keep a good balance between being assertive and open to client ideas and perspectives. It is not good to just accept ideas passively. The software product manager and development team should suggest ideas and perspectives in meeting with the client, but they should not enforce their viewpoints aggressively.

The software product manager should try to provide structure in conversations with the client about requirements. This will help the client organize his or her thoughts. Care must be taken not to steer the client, however.

Clients sometimes come with set ideas about what they want in a product. However, they sometimes don't know what is possible technically, or they have ideas that would actually be hard for end-users to understand or use. The software product manager should try to understand why the client has set upon these ideas. A good way to do this is to ask the client to explain how they envision an end-user interacting with the product. Once the client's reasoning is understood, alternative

ideas can be suggested. It may also be useful to politely highlight why certain ideas may not work. This can also be done through exploring limitations and different scenarios.

Note that it is the role of the software product manager to make sure a client is well informed and understands their options. However, at the end of the day, the client is the one who makes the key decisions regarding requirement, even if the development team would prefer to go a different route.

TIP: A good way to involve clients and to understand their requirements is to ask good questions!

Good questions are open-ended and cannot be answered with a simple yes or no. Questions that ask "why?" are particularly good, because they encourage a client to analyze and identify their core needs. Simon Sinek's Ted Talk on asking "Why?" gives a good explanation of what makes this question valuable, which is available in class resources.

"What do you want?" or "What are your requirements?" are less useful questions because they are too broad and do not provide good structure for understanding client needs.

Requirements should be revisited often with the client. After a first meeting, for example, it is a good idea to show the client mock-ups and prototypes produced from initial discussions. With these concrete tools, clients can more readily identify what they like in a product, what they want, and what they dislike. Revisiting requirements also allows clients to answer questions, advise on design choices, and provide feedback.

TIP: It is good practice to number requirements with unique identifiers throughout the project, so they are easy to reference as the project develops!

An important tool for interacting with clients is the use and creation of a glossary. A **glossary** is a list of terms and their definitions that relate specifically to the product being built. It is common for many different terms to be generated for the same thing if a glossary is not used and agreed upon, which can make discussions confusing.

In summary, key aspects to remember in customer interaction include:

- Involve the client through meetings and continuous re-visitation of requirements.
- Use other information sources, such as end users.
- Ask good questions.
- Be assertive and open in client interactions.
- Clearly communicate realistic requirements and timelines to manage client expectations.
- Use common terms via glossaries.
- Keep track of requirements
- Remember that it is the responsibility of the software product manager to clearly communicate the pros and cons of requirements, but the client is the one who must make the key-level requirement decisions.

Use Cases

Use cases, developed by Ivar Jacobson in 1986, are a good tool for understanding a product. They can be defined as a way to identify, clarify, and organize details of a task within a product. Uses cases take place in particular environments to achieve particular goals. In other words, it's a way of explaining a set of sequential interactions users might have with the features of a product.

A good use case outlines the proposed task from the point of view of a participating actor (usually a user), and it should not require deep knowledge of technology to understand. Use cases may be presented in tables as outlined below.

Name	
Participating Actors	
Goals	
Triggers	
Pre-Condition	
Post-Condition	
Basic Flow	
Alternate Flows	
Exceptions	
Qualities	

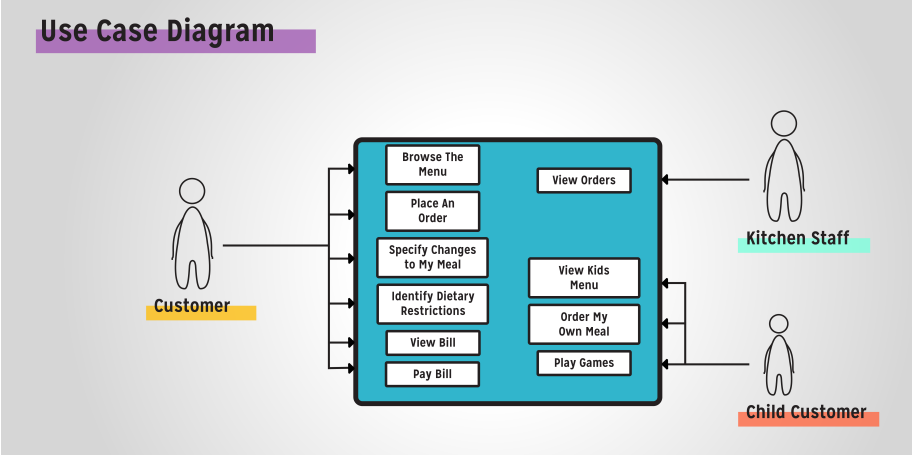
Use case elements are defined as follows:

Name	This refers to the **name** or title given to the use case. It should describe the task being undertaken, but it should also remain short and simple.
Participating Actors	The **participating actors** are the people envisioned to be engaging in the action of the use case. Participating actors are described by the roles of the person and what they do with the product.
Goals	The **goals** describe the intended objectives of the use case.
Triggers	The **triggers** of a use case are the events that prompt the use case to begin.
Pre-Condition	**Pre-conditions** are the conditions that must be in place before the use case can occur.
Post-Condition	**Post-conditions** are the conditions that become true once the use case is complete.
Basic Flow	The **basic flow** is a step-by-step description of what occurs in a use case. Basic flows usually describe **sunny-day scenarios** (also known as best-case scenarios), which are situations where everything works perfectly. Other scenarios are outlined in alternate flows, if necessary.
Alternate Flows	An **alternate flow** is presented as a case where things go in a different way from the **sunny-day scenario** of the basic flow.
Exceptions	**Exceptions** are situations where the use case would not work. The step in the basic flow where the exception occurs should be identified, and alternate steps should be proposed to resolve the problem.
Qualities	**Qualities** are the quality specifications you would like the product to meet. These are often non-functional requirements, such as time frames, or monetary restrictions. These conditions are determined through client and development team discussions and set standards through the project.

For example, a completed use case would look like:

Name	View Bill
Participating Actors	Customer
Goals	View the Bill for the Order
Triggers	Request to View Bill
Pre-Condition	Menu Items on Menu, Selecting Dish, Placing Order
Post-Condition	View Bill and Pay for Bill
Basic Flow	1) User Requests to View Bill 2) User Views Bill
Alternate Flows	User Gets Wait Staff to Print and Bring Them Bill
Exceptions	No Dishes Ordered
Qualities	• Bill Available After Order placed • Bill Takes Less then 10 Seconds to Load • All Dishes That Were Selected Appear on the Bill • Prices on Bill Match Prices on Menu

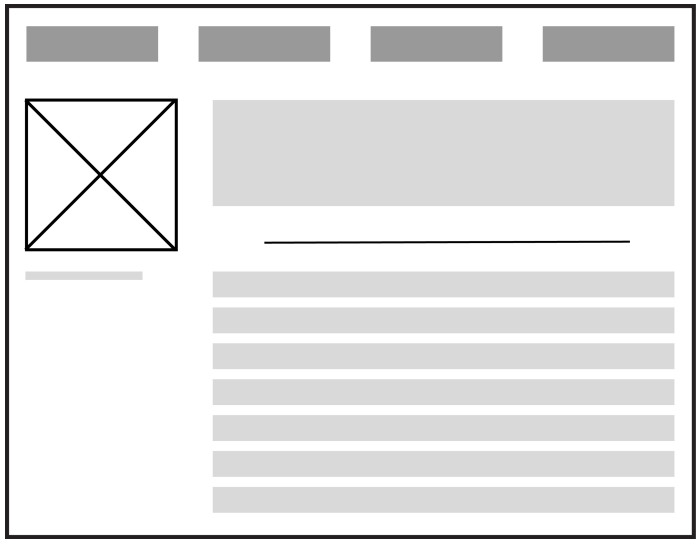
It is helpful to depict the use cases of a product with a diagram (see below). Use case diagrams are high-level, visual representations outlining all the tasks supported by the product being created. Actors and their respective uses cases are also represented. Overall, the diagram should show the entire product, tasks that may be undertaken while engaging with the product, and roles supported by the product.



Use cases are a useful tool for software product managers and their development teams when establishing requirements. By creating use cases and sharing them with clients, the project has more definite guidelines, and development tasks become more organized and elaborate.

Wireframes

One of the most important techniques of product development is the use of **wireframes**. A wireframe, also known as a mock-up, can be thought of as a kind of early blueprint. It is a basic visual representation of the product. See the example below of what a biography web page might look like.



Wireframes are used for many purposes. These include:

- getting an idea for what will be developed
- demonstrating ideas to clients or users and encouraging their feedback and involvement
- communication among the development team
- helping the client or users communicate with the software product manager and team (some people may find it easier to sketch out their ideas to describe them)

Although a visual representation, a key aspect of wireframes is their simplicity. Wireframes are not meant to be a demo of the interface of a product but rather outline basic functionalities and end-user tasks. This means that wireframes do not use

colours or images. In fact, it is important that detailed design features do not creep in to wireframes.

Instead, wireframes may suggest things like where buttons, text fields, and images are placed. The elements displayed should theoretically allow a user to do a task with the product. As in a blueprint for a house, for example, details such as wall colours or lighting fixtures are not outlined. Later, detailed user interface (UI) design can develop further.

Wireframes are usually developed quickly after an initial discussion with the client regarding requirements and presented thereafter. Clients then have a chance to see an early version of what their product might look like and provide feedback. It also ensures that both the client and development team are working towards the same vision.

TIP: There are many tools to help develop wireframes! You can always draw images by hand and then scan them, but there are many drawing tools on the computer that can help you create wireframes.

Some examples are:

- Google Docs
- Microsoft Powerpoint
- Balsamiq (online tool available at https://balsamiq.com/)
- OmniGraffle (a paid app available at https://www.omnigroup.com/omnigraffle/)
- Wirify (a tool which turns any web page into a wireframe, available at http://www.wirify.com/)

Some examples of wireframes can be found at http://wireframes.tumblr.com/

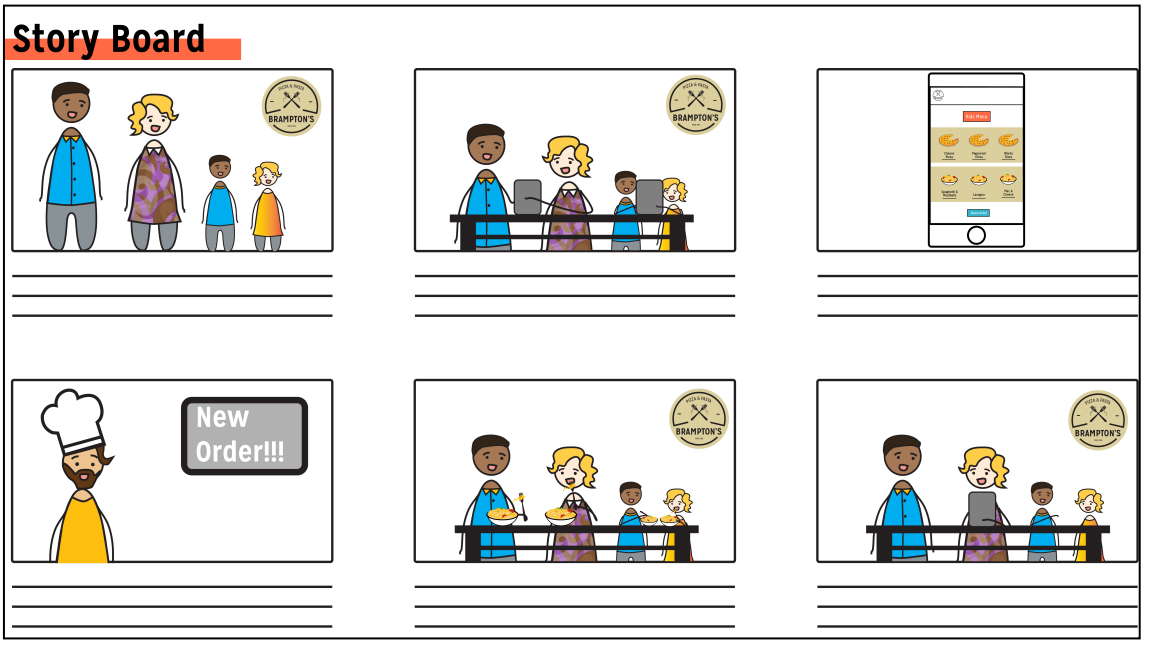
Storyboards

Another technique used to help in forming requirements are storyboards. Many disciplines make use of storyboards. In software development, **storyboards** are sequential, visual representations of an interaction with the software product.

There are two main types of storyboards. Both types can be helpful for discussions between clients and the development team. Storyboards help to elicit further requirements of the product and refine existing ones.

The first type of storyboard describes high-level user experience with the product. This type of storyboard tends to look like a comic strip, where each action is illustrated in the sequence or flow of using a product, including the decision to use the product and outcomes at the end of an interaction. If multiple features are offered in the product, then each should get a storyboard.

An example of this first type of storyboard appears on the following page.



Captions in the panels should read:

1. The Smith family decided to go to Brampton's Pizza for a nice family dinner.

2. They are seated at a table and use the tablets at the table to browse the menu.

3. They browse the dishes on the application and place their order.

4. The cooks in the kitchen receive the order.

5. The food was delivered to the table once it was read and they have a nice family dinner.

6. Once they were finished their meal, the mother viewed and paid for the bill on the application.

Generally, the envisioned user, or participating actor (as described in use cases) in this type of storyboard, is given a persona. Personas provide more elaborate back stories to the participating actors, including details such as age, ethnicity, income level, job, personality, etc. Providing high-level details helps the storyboard address specific issues.

The first type of storyboard has many potential uses, including:

- ensuring the entire development team is on the same page
- enhancing the possibility of identifying features to improve or create in the software product
- ensuring the vision of the product remains clear in its use
- use in marketing or demos

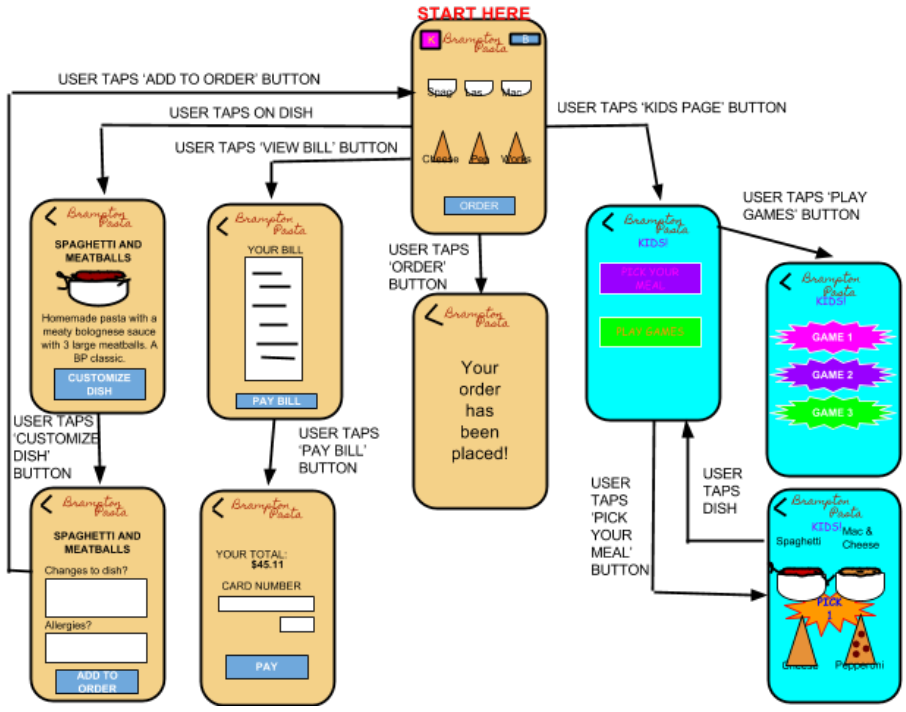
The second type of storyboard combines wireframes and basic flow from use cases in order to show how the end-user interacts with the *user interface* of the product in detail. It is more closely related to software design.

It also shows all the sequences of user interaction with the product, and the outcomes of those interactions. Each state of the product during interaction is illustrated with a wireframe. The user interface element needed to get to the next state is also illustrated. Transitions between states are generally depicted with an arrowed line.

The second type of storyboard has many potential uses, including:

- helping the development team to design requirements
- helping identify what could be overly complicated or missing in supporting tasks the users perform
- ensuring the development team is aware of the functionalities and feel of the product
- guiding technical writers in creating training materials

Below is an example of the second type of storyboard.



Module 3: Writing Requirements

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

(a) describe requirements within the 12 Agile principles.
(b) create a user story and recognize the format of a user story.
(c) assess what makes a good user story.
(d) 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.
(e) create an acceptance test for user stories based on acceptance criteria.
(f) explain what a product backlog is, and help prioritize user stories.
(g) describe what product backlog is and how it works within Scrum.
(h) create a story map, and identify the benefits of using a story map.
(i) identify any missing or inconsistent user stories within a story map.

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.

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.

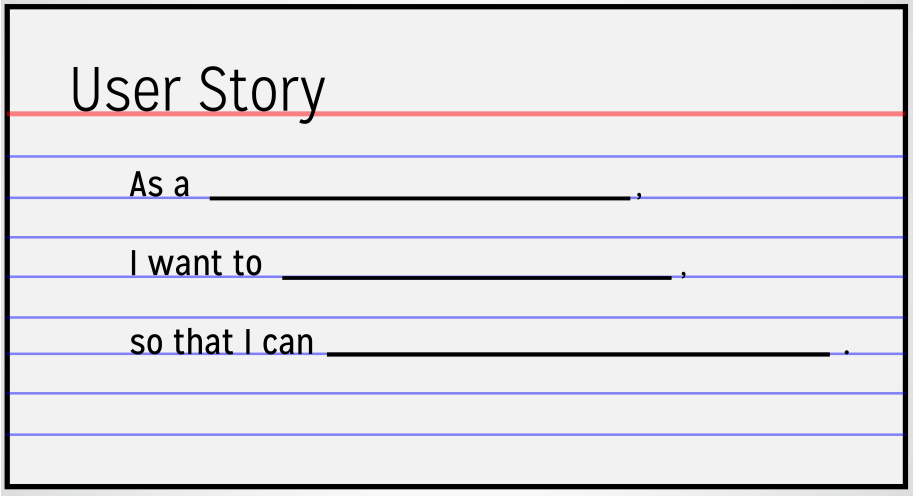
The "what" of the requirement is the specific task or functionality the stakeholder wants to achieve by using the product.

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.

Below is an example of what a user story might look like on an index card.



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.

An example of an epic user story is:

"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 step can become a test if passed.

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.

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.

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. 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.

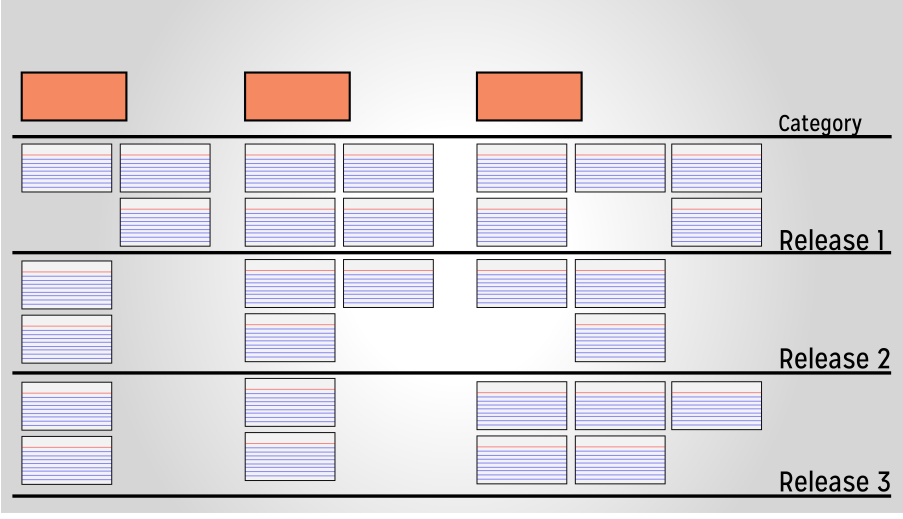
Story Maps

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. These include:

- 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. Kanban, as discussed in **Software Process and Agile Practices**, is a method 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.

Module 4: Quality Requirements

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

(a) describe the criteria used to assess whether or not a user story is good: correct, complete, consistent, clear, realistic, traceable, and verifiable.
(b) recognize a "correct" user story.
(c) recognize when user stories conflict.
(d) identify an ambiguous requirement.
(e) recognize common terms that might contribute to making a user story ambiguous.
(f) clarify an ambiguous term within a user story.

Criteria for User Stories

It is possible to evaluate the quality of user stories using a set of criteria. Quality user stories are important for guiding development in order to avoid mistakes and confusion. Accordingly, there should be no missing user stories, and existing user stories should be easy to understand and free of ambiguity.

Suggested criteria for assessing the quality of user stories are:

Correct	User stories must be correct or the development team will create features that are not in line with the client's vision. Time is then wasted on both the creation of incorrect features and removing them later.
Complete	A complete user story is one where all requirements necessary to describe the problem are included. If requirements are missing, those features will be lost in the final product. Story maps are often used to identify any missing requirements.
Clear	User stories are clear if they are easy to understand and can only be interpreted one way. If ambiguities are left in user stories, then a product may end up being developed differently than the way the client envisioned, or the product could be tested a different way from either of those two interpretations. Requirements Technical Review and Repair exercises are used to reduce ambiguities. These exercises bring in outsiders to the project to review requirements based on the criteria listed here. Wireframes and storyboards, among other visual tools, can also help eliminate ambiguities.
Consistent	This criterion requires verifying that requirements do not conflict with one another.

Verifiable	User stories should be testable or measurable in some way, in order to determine if each requirement has been satisfied. Often, quantifying aspects of requirements helps make them verifiable.
Feasible	A feasible user story is one that can be realistically implemented when the limitations of the product are taken into account. This includes the technology, tools, resources, and personnel available, as well as the budget and schedule. If a user story is not feasible, it will likely incur additional costs and a decline in satisfaction from clients.
Traceable	User stories should be traceable, meaning that each requirement in a user story should be able to be tracked throughout the project. Every requirement has corresponding code and tests. Requirements should therefore be connected to these associated design and implementation artifacts through a tracking mechanism. This ensures that only necessary code is written and that all requirements are tested. A technique where unique identifiers are assigned to each requirement is often used to facilitate the tracking of a requirement within product code, tests, bug reports, and change logs. Unique identifiers can be as simple as sequential numbers.
Manageable	A good user story is manageable, meaning that it is ideally independent of other user stories and can be changed without excessive impact on other user stories. This is not always realistic because user stories will have some dependency on each other, which places importance on traceability of user stories. Grouping or linking user stories often helps keep them manageable, so if one changes, any other related elements can also be accounted for.

Simple	User stories should be simple, meaning that they should be kept minimalistic. A good user story focuses only on the feature that will be implemented; it does not reference *how* it will be implemented. Unnecessary design details are also not included in good user stories. In fact, a good user story should be independent of technology.

Ambiguous Requirements

Requirements may encounter problems if they are ambiguous. An ambiguous requirement can have multiple interpretations, or it will not provide all necessary details. This leads to miscommunication between the development team and the client, and potentially incorrect software. Avoiding ambiguous requirements saves time and money.

In order to avoid creating ambiguous requirements, there are certain words to watch for in user stories.

"a" or "an"	The words "a" or "an" are not specific. They could refer to singular or multiples of an item or event. It is better to use a quantifier.
"after" or "before"	The words "after" or "before" are vague because they can describe either the immediate instance after or before the event or object in question, but they could also refer to *any* instance after or before the event or object in question. It is better to give a firm description of the exact position in the sequence that an event or object occurs.
"all"	The world "all" is confusing in some sentences as it is not clear what is referred to. For example, "All vehicles in the parking lot must be licensed by an owner" could mean: ▪ one owner must license all the vehicles in the parking lot ▪ each vehicle is licensed by a different owner ▪ each vehicle is licensed by an owner, but an owner may license more than one vehicle It is good practice to be as specific as possible.

“also”	The word “also” can be confusing with context. User stories should be able to stand alone, so the word also should only be used if further context beyond the requirement is not needed.
“and”	The word “and” can be ambiguous because it could list two independent conditions together in one sentence, or it could refer to two conditions needed for something to happen. If two separate and independent conditions have been joined together, it is better to break this up into two.
“both”	The word “both” can be confusing because it could be unclear which noun is referred to in what way. For example, “Both restaurants are run by a manager” could mean: ▪ each restaurant is run by a different manager ▪ one manager is running both restaurants It is important to be specific.
“current”	The word “current” can be ambiguous, as it could refer to what is in session at the same moment, or it could refer to what is being referred to at the time of discussion. For example, “The charges were added to the current billing period” could mean: ▪ charges were added to the billing period happening now ▪ charges were added to the billing period being considered at the time
“every”	The word “every” can be confusing because it is unclear which noun “every” refers to, in a similar way to the word “both.”
“following”	The word “following” is ambiguous in a way similar to “after” or “before.” It is unclear if “following” refers to the item or event that immediately appears after the item or event in question, or if it follows at any position or time after the item or event in question (not necessarily the one immediately after).

“for”	The word “for” could refer to an event repeated over a certain time period, or it could refer to an event occurring only once over a specific time period. For example, “The product will show the weather for the next three days” could mean: ▪ the product shows the weather that is predicted on the next three-day period ▪ the product shows the weather on each of the next three days and stops after that
“from”	The word “from” is unclear because it is unclear where something starts. For example, “The product will record data starting from Tuesday, November 23rd” could mean: ▪ the product will begin to record data starting on Tuesday, November 23rd ▪ the product will begin to record data starting on Wednesday, November 24th after Tuesday is finished
“last”	The word “last” could refer to the latest iteration, the previous iteration, or the final iteration of an event or object.
“may”	The verb “may” is ambiguous because it is passive language. It could imply that something must be as described, or it could imply that something could be as described or could be something else entirely. For example, “User may use a credit card to pay” could mean: ▪ users should use a credit card to pay ▪ users could use a credit card to pay or pay with something else.
“only”	The word “only” can cause confusion because sometimes it is not clear which noun the word refers to. For example, “Dogs are only allowed in the dog park” could mean: ▪ dogs are not allowed anywhere except the dog park ▪ only dogs are allowed in the dog park, not owners or other animals
“or”	The word “or” is ambiguous because it does not distinguish what happens in the case that both conditions on either side of the word are true.

“same”	The word “same” is unclear because it could refer to two identical objects, or it could refer to one object acted upon twice. For example, “Two customers bought identical couches” could mean: ▪ two different customers bought identical couches ▪ two customers split the cost and bought one couch together
“should”	The word “should” is problematic like the work “may” because it is passive language. It could mean that the user must do something. It could also mean the user probably should do something, but if the user does not, the system will take alternative actions.
“their”	The word “their” is unclear. It could refer to all of one person’s objects in question plus all of another person’s objects in question. It could also refer to the joint objects in question based on when the action started. For example, “The married couple’s joint bank account contains all their money” could mean: ▪ the bank account contains all the money that one person has as well as all the money that the other person has ▪ the bank account contains only the money earned while the couple was married and deemed as “shared” money
“until”	The word “until” could mean up to a certain point, or could mean up to a certain point unless ended otherwise. For example, “The application will search the database until a match has been found” could mean: ▪ the application will only stop when a match has been found ▪ the application will stop when a match has been found, if not stopped otherwise
“we”	The word “we” is unspecific as to whom it is talking about. It could refer to the person being spoken to and yourself, or it could refer to other people not present and yourself, or it could refer to society at large. More specific terms should be used.

"when"	The word "when" is not specific about what the term applies to if there are multiple phrases. For example, "The timer is set to zero when the runner completes a lap around the track" could mean: ▪ the timer is set to zero by the time the runner completes a lap ▪ the timer is set to zero the first time the runner completes a lap ▪ the timer is set to zero each time the runner completes a lap around the track
"will"	The word "will" is ambiguous because it could mean that someone will have to do something, or it might mean that something will have to happen to someone before other things can happen. For example, "The username will be verified before the password is entered," could mean: • someone will have to verify the username before the password is entered • the user must be verified before anyone attempts to enter a password

There are many other words that may create ambiguous requirements, but the ones listed above are the most common. If these words are used, the client should be asked to elaborate.

Unclear language can also arise during discussions between the client, software product manager, and development team. It is important for the software product manager to ask questions to clarify discussions. Good questions include examining the truth of presented facts and asking for elaboration on uncertain terms or certain points of perceived needs and requirements. Determining what the client means and why is important to creating quality requirements.

Version 0.8.0