



USE-CASE 3.0

The Guide to Succeeding with Use Cases
Refreshed

Ivar Jacobson
Ian Spence
Keith de Mendonca

May 2024

About this Guide	3
How to read this Guide	3
What is Use-Case 3.0?	4
The Use-Case Foundation	5
Underlying Principles	5
Core Concepts	5
A simple example	7
Exploring the Principles	14
Principle 1: Universally Applicable	14
Principle 2: Start with the big picture	14
Principle 3: Focus on value	16
Principle 4: Involve your stakeholders	17
Principle 5: Tell the whole story	18
Principle 6: Trigger conversations	18
Principle 7: Prioritize readability	19
Principle 8: Just enough, just in time	19
Principle 9: Implement in stages	20
Principle 10: Build the system in slices	22
Use-Case 3.0 Content	24
Things to Work With	24
Work Products	31
Things to do	36
Practices	43
Using Use-Case 3.0	53
Use-Case 3.0: Applicable for all types of system	53
Use-Case 3.0: handling all types of requirement	54
Use-Case 3.0: Applicable for all development lifecycles	54
Use-Case 3.0: Scaling to meet your needs – scaling in, scaling out and scaling up	63
Conclusion	64
Appendix 1: Work Products	65
Glossary of Terms	75
Acknowledgements	76

About this Guide

This guide describes how to apply use cases in an agile and scalable fashion. It builds on feedback and lessons learned since the initial publication of the Use-Case 2.0 e-book in 2011. Our goal is to provide you with an integrated family of practices which help you get the most out of your use cases. These practices are applicable for small co-located agile teams, but are also applicable to large distributed teams, outsourcing, and complex multi-system developments. Select, mix and match the use case practices described in this guide to support your work; add use-case practices later if you need more detail and structure to guide your work.

This guide also outlines how to create user stories from use cases - it connects the power of use cases to the complementary benefits of user stories. Use-Case 3.0 is 100% compatible with teams that are already using User Stories.

This guide presents the essentials of use-case driven development as an accessible and re-usable set of practices. It also provides an introduction to the idea of use cases and their application. It is deliberately kept lightweight. This is not a comprehensive guide to all aspects of use cases, or a tutorial on use-case modeling. It may not be sufficient for you to adopt the practice. For example, it is not intended to teach you how to model, for this we refer you to our previously published books on the subject.

How to read this Guide

The guide is structured into four main chapters:

- What is Use-Case 3.0? - A two-page introduction to the Use-Case 3.0 practice family.
- First Principles - An introduction to use cases based around the 10 principles that act as the foundation for the practice family.
- Use-Case 3.0 Content - An overview of the practice family as a whole presented as a set of key concepts, activities, work products, and the rules that bind them together.
- Using Use-Case 3.0 - A summary of when and how to apply the practices that make up the Use-Case 3.0 Practice Family.

These are topped and tailed with this brief introduction, and a short conclusion.

If you are new to use cases then you might want to read the “What is Use-Case 3.0?”, the “First Principles”, and the “Using Use-Case 3.0” chapters to understand the basic concepts. You can then dip into the “Use-Case 3.0 Content” as and when you start to apply the practice.

If you are familiar with the basics of use cases then you might prefer to dive straight into the “Use-Case 3.0 Content” and “Using Use-Case 3.0” chapters once you’ve read the “What is Use-Case 3.0?” chapter. This will help you compare Use-Case 3.0 with your own experiences and understand what has changed.

If you have previously read or applied Use-Case 2.0 then you will recognize the structure of the new practices. Read the “Exploring the Principles” chapter to familiarize yourself with the update, and then follow the advice above for people already familiar with use cases.

Throughout this document you will discover how use cases can be used to generate user stories, and be used as an integral part of any agile development practice.

What is Use-Case 3.0?

To get to the heart of what a system must do, focus on who or what will use it, and then look at what the system must do for them to help them achieve their goals.

A use case is all the ways of using a system to achieve a goal of a particular user.

Use-Case 3.0 is a scalable, agile practice family that uses use cases to capture a set of requirements and drive the incremental development of a system to fulfill them.

It drives the development of a system by first helping you understand how the system will be used and then helping you evolve an appropriate system to support the users. It can be used alongside your chosen management and technical practices to support the successful development of software and other forms of system. As you will see Use-Case 3.0 is:

- Lightweight
- Scalable
- Versatile
- Easy to use
- Presented as a family of practices to aid adoption and focus on addressing your immediate concerns

Use cases make it clear what a system is going to do and, by intentional omission, what it is not going to do. They enable the effective envisioning, scope management and incremental development of systems of any type and any size. They have been used to drive the development of software systems since their initial introduction at OOPSLA in 1987. Over the years they have become the foundation for many different methods and an integral part of the Unified Modeling Language. They are used in many different contexts and environments, and by many different types of team. For example, use cases can be beneficial for both small agile development teams producing user-intensive applications and large projects producing complex systems of interconnected systems, such as enterprise systems, product lines, and systems in the cloud.

The use-case approach has a much broader scope than just requirements capture. Use cases can and should be used to drive the development, which means that Use-Case 3.0 also supports the analysis, design, planning, estimation, tracking and testing of systems. It does not prescribe how you should plan or manage your development work, or how you should design, develop, or test your system. It does however provide a structure for the successful adoption of your selected management and development practices.

Use-Case 3.0 exists as a proven and well-defined set of practices. Although the term Use-Case 3.0 suggests a new version of use cases, it does not refer to an update of the Unified Modeling Language, but rather to cumulative changes in the way software developers and business analysts apply use cases. A use case is still a use case but the ways that we present, address, and manage them have all evolved to be more effective. The changes are not theoretical but are pragmatic changes based on 30 years of experience from all over the world and all areas of software development.

The Use-Case Foundation

The Use-Case Foundation document by Ivar Jacobson and Alistair Cockburn presents a set of principles and concepts that underlie all successful applications of use cases including Use-Case 3.0.

Use-Case 3.0 extends the foundation adding the concept of the Use-Case Slice; a foundational element of Use-Case 2.0 and this update.

Underlying Principles

The foundational principles are:

1. Use cases apply to systems of all types and sizes: businesses, IT systems, physical systems or any combinations thereof.
2. Use cases help you understand the big picture: the system's purpose and how it will be used.
3. Use cases focus on value: the users' goals and how best to achieve them.
4. Stakeholder involvement is essential: bring all the involved parties together to establish the intent and scope of the system.
5. A use case tells the whole story, as a story, from the initial event to the realization of the value it provides or the eventual failure if it can't be met. It includes how to handle any problems and alternatives that may occur on the way.
6. Use cases trigger conversations: while discussing the possible alternate flows, you and your co-writers will think of missing steps and missing alternatives. These conversations help you find situations that often get overlooked.
7. Prioritize readability: the goal is to communicate the big picture to everyone involved, generating comments, spotting any gaps, and getting their buy-in.
8. The amount of detail and the format used will vary to match your circumstances: You can start with a sketch of the flow of events and add detail as needed.
9. A use case can be implemented in stages: develop and put into place some key flows of a use case early to capture value and feedback, add less used or less critical flows over time strategically.

Use-Case 3.0 adds a tenth principle:

10. A system can be developed in slices where each slice is one or more paths through one of the system's use cases plus the relevant design, code and tests used to implement and verify them.

In this section we will look at the core concepts that underpin Use-Case 3.0 and how these principles are applied in Use-Case 3.0.

Core Concepts

1. A system of interest
2. An actor with a goal
3. A flow-of-events (there will be several)
4. One or more use cases to collect the flows.
5. A use-case model to contextualize and visualize the use cases.
6. One or more use-case slices to scope the work.
7. One or more actionable work items (user stories, features, or tasks) to complete the work.

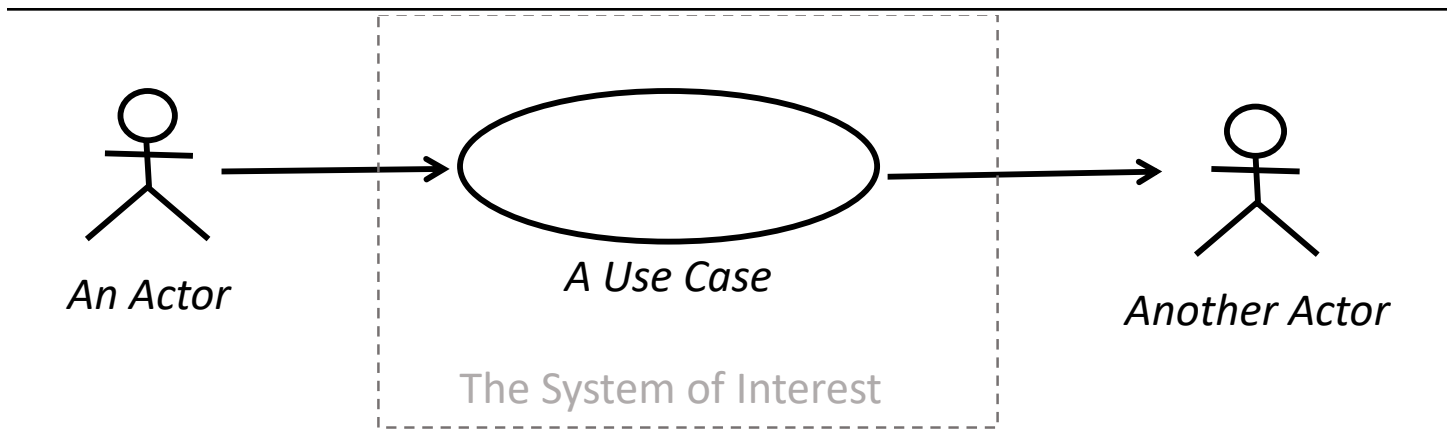


FIGURE 1: A USE CASE

A Use Case - A use case is all the ways of using a system to achieve a goal of a particular user.

Notes:

- This includes all the successful, challenged and failure paths.
- It may be described textually or visually.
- It is independent of implementation, technology, and platform.

The System of Interest - The system used to achieve the goal.

An Actor - An actor identifies a role played when interacting with the system.

“Actor” is intended to cover anything with behavior. It can be a person, an organization, a piece of software, or any combination.

A use case might involve many actors: the actor that initiates a use case is known as the “primary actor” and the actors called upon by the system are known as “supporting actors”.

The Goal - The reason that the user will use the system and the value that they will receive when successfully using the system.

The Flow of Events - A use case is presented as a network of flows, each describing a path to value. Taken together the set of flows capture all the ways of using the system to achieve the goal of the primary actor.

A Use-Case Model - A model that captures and visualizes all the useful ways to use a system.

A Use-Case Slice - A slice of a use case that provides clear value to the user or other stakeholders. Typically, a use-case slice captures one of the ways of using a system to achieve a goal.

Notes:

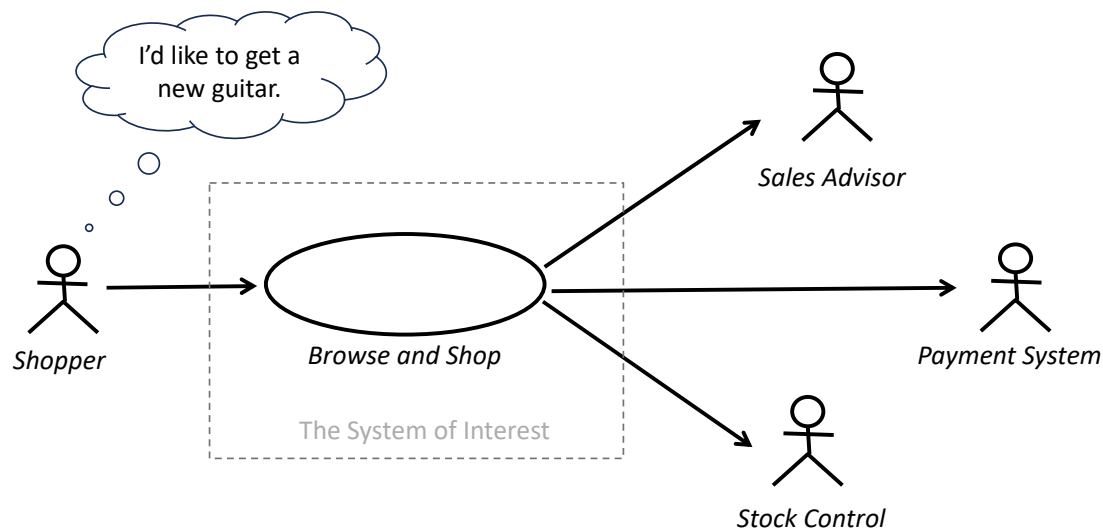
- A use-case slice always starts at the beginning of the use-case and ends at its end.
- A use-case slice will traverse one or more of the use-case’s flows.
- A use-case slice is always testable and will complement the flow of events with explicit test cases.

A Work Item - a small, actionable piece of work that can be given to an individual or team to complete. Most use-case slices will require multiple work items to be completed to complete their development and testing. Common forms of work item used to implement a use-case slice include User Stories, Features and Tasks.

A simple example

Let's assume you are a musician who would like to find a new guitar to purchase from a music store over the internet. With respect to an online music store, what role would you be playing and what would be your goal with respect to the system?

When using use cases, we don't want to have a separate actor for every type of shopper (guitarist, bassist, singer, parent, school, manager etc.) and a different use case for every different product we sell (guitars, oboes, microphones, effect pedals etc.) as the way they interact with the system to browse and shop for products will be the same. We would probably end up with something like the use-case shown in [FIGURE 2](#).



Primary Actor

In this case a shopper with the goal of selecting and purchasing a product.

The System of Interest

In this case an on-line portal providing advice on all things musical. One of this system's use cases is 'Browse and Shop'

Supporting Actors

Other Actors that can be involved in the successful completion of the use case. These can be other systems or other people. In this case the system of interest needs to interact with a Stock Control System, a Payment System, and for specialist, high value products a Sales Advisor.

FIGURE 2: THE BROWSE AND SHOP USE CASE

Although this use case captures the most valuable thing that our on-line music shop can do it probably isn't sufficient to make a usable system. Some other use cases may be required such as those shown in bold in **FIGURE 3:**

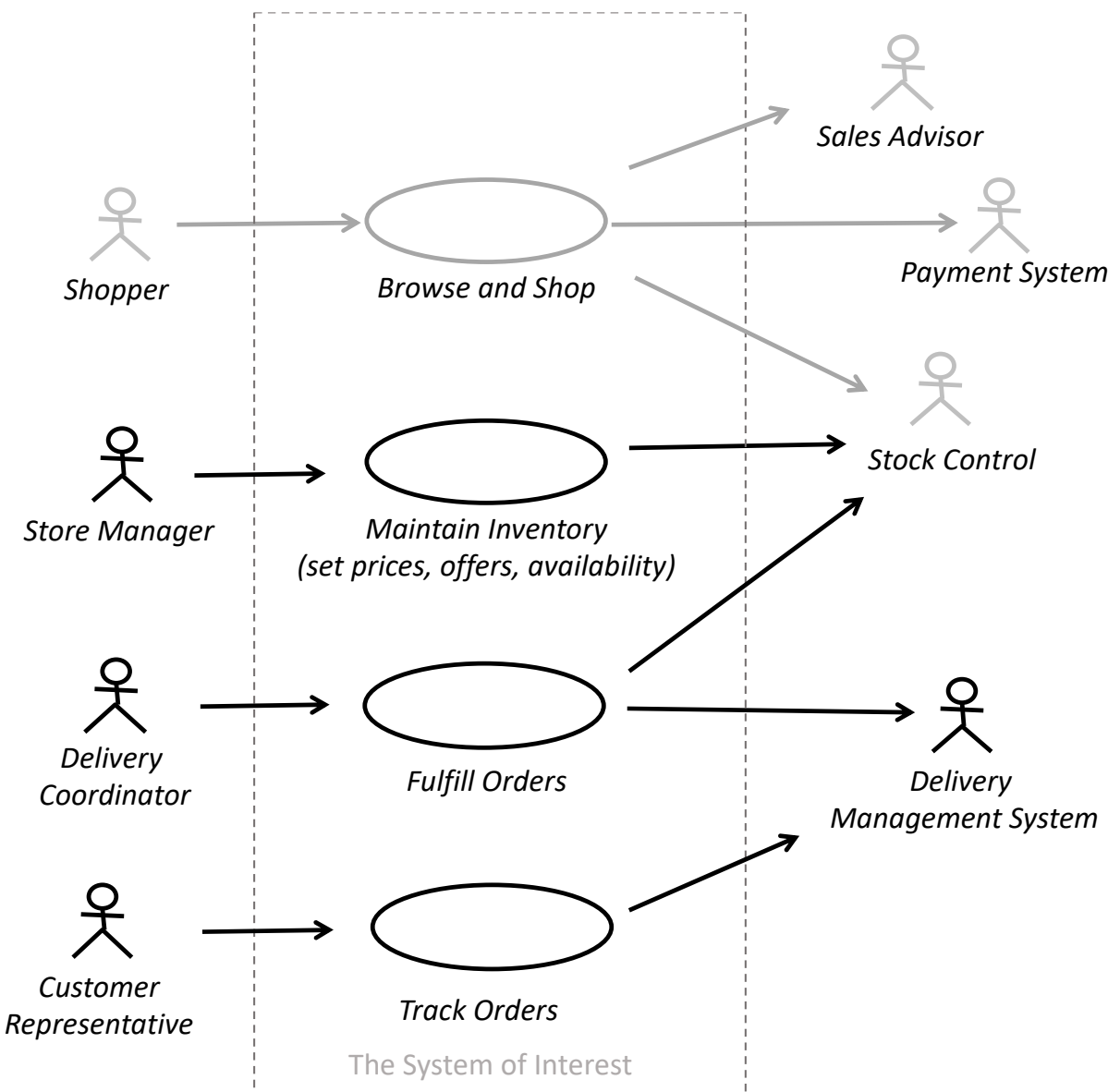


FIGURE 3: EXPANDING OUR SET OF USE CASES

If you look closely, you might find issues and problems with this diagram such as missing actors, missing use cases and confusing names. This is deliberate. This is exactly the sort of conversation that we want the Use-Case Model and any supporting overview diagrams to start.

Each use case is presented as a network of flows, each describing a path to value.

Basic Flow / Main Scenario	Alternate Flows / Alternatives
1. Step 1 – The use case starts when...	Alt1 – Something that can go wrong that needs to be handled
2. Step 2	Alt 2 – Something optional that should be provided
3. Step 3	Alt 3 – A special case that needs to be handled differently
⋮	⋮
N. Step N	Alt N
The use case ends.	

FIGURE 4: BASIC AND ALTERNATE FLOWS

The Basic Flow - The normal, happy path to value often referred to as the ‘main scenario’ or the ‘happy path.’ This is described as a simple sequence of steps each of which involves the system and / or one of the actors doing something.

Alternate Flows - A list of all the special cases, alternative paths, optional steps, and errors that need to be handled.

The key aspect of a use case is its structure: the way it identifies the basic and alternate flows - this acts as a map of how the system will be used. The flow of events can be described as simply as a bulleted list of steps and alternatives, or elaborated to fully describe what should happen at each step or within each alternative. It can be described in text, as above, or in some graphical form.

What is important is the *accuracy* of the flow of events and not *how detailed* you write out the steps and alternatives.

FIGURE 5 presents a simple example of the flow of events of the Browse and Shop Use Case.

Primary Actor: Shopper	Help the shopper to find the most suitable product to meet their needs and help them to purchase it.
<p>Basic Flow / Main Scenario</p> <p>The use case starts when a Shopper indicates they'd like to find a product</p> <ol style="list-style-type: none"> 1. Browse Products 2. Select Products for Purchase 3. Provide Payment Details 4. Provide Delivery Details 5. Confirm Purchase <p>The use case ends.</p>	<p>Alternate Flows / Alternatives</p> <p>Alt1 – Keyword search for products</p> <p>Alt 2 – No products selected</p> <p>Alt 3 – Invalid payment details</p> <p>Alt 4 – Payment system unavailable</p> <p>Alt 5 – Retrieve stored payment and delivery details</p> <p>Alt 6 Invalid delivery details</p> <p>Alt 7 – Product out of stock</p> <p>Alt 8 – Stock control system unavailable</p> <p>Alt 9 – No purchase confirmation</p> <p>Alt 10 – Quit shopping with no purchase</p> <p>Alt 11 – Shopper stops responding</p> <p>Alt 12 – Shopper needs expert advice</p>

FIGURE 5: AN OUTLINE OF THE BROWSE AND SHOP USE CASE

If you look closely, you might find issues and problems with this use case such as missing steps and missing alternatives. This is deliberate. This is exactly the sort of conversation that we want the use-case to start.

You will also notice that this is too much to implement in one go, and potentially more than you would ever want to implement. Even the Basic Flow may be too much to address in your first pass.

This is where the idea of Use-Case Slices comes into play.

Primary Actor: Shopper	Help the shopper to find the most suitable product to meet their needs and help them to purchase it.	
Basic Flow / Main Scenario		Alternate Flows / Alternatives
The use case starts when a Shopper indicates they'd like to find a product		Alt1 – Keyword search for products Slice 3: Keyword Search
1. Browse Products		Alt 2 – No products selected
2. Select Products for Purchase		Alt 3 – Invalid payment details
3. Provide Payment Details		Alt 4 – Payment system unavailable Slice 4: Handle Customer Data Issues
4. Provide Delivery Details		Alt 5 – Retrieve stored payment and delivery details
5. Confirm Purchase		Alt 6 Invalid delivery details
The use case ends.		Alt 7 – Product out of stock Slice 5: Product Unavailable
Slices 1 and 2: Slice 1 – Buy a single product Slice 2 – Buy multiple products		Alt 8 – Stock control system unavailable
		Alt 9 – No purchase confirmation Slice 6: Customer Walks Away
		Alt 10 – Quit shopping with no purchase
		Alt 11 – Shopper stops responding
		Alt 12 – Shopper needs expert advice

FIGURE 6: SLICING UP BROWSE AND SHOP

FIGURE 6 shows an initial slicing up of the Browse and Shop use case. In this case six slices have been identified. These are considered to be sufficient to provide a robust, usable user experience.

Things to note here are that:

1. Not all of the flows have been included in the slicing. There is no need to slice up the entire use case. Flows that are considered out of scope or that are not important at the moment can just be left unsliced as part of the use case.
2. Multiple slices have been identified from the basic flow. In this case we have decided to separate buying a single product from buying multiple products. This is to enable the early implementation and testing of an end-to-end thread through the use case without having to wrestle with the issues of handling multiple product selection and deliveries.
3. We can tackle the first two (or even all 6) slices without having to find and enumerate all of the alternate flows.

We'll talk more about slicing in later sections. The important things to remember about the Use-Case Slices are that each slice:

- Provides clear value to the user or other stakeholders.
- Typically captures one of the ways of using a system to achieve the use case's goal.
- Starts at the beginning of the use case and ends at its end.
- Traverses one or more of the use-case's flows.
- Is testable and will complement the flow of events with explicit test cases.

The use-case slices make good items for scope and priority management. For example, it would be easy to apply a prioritization scheme such as MoSCoW (Must, Should, Could, Won't) to the slices identified above.

The slices are also a better mechanism for dividing up a use case than the use-cases flows and steps as they leave more room for evolution during their analysis and design. For example, when working on Slice 4 Handle Customer Data Issues we may discover that our original Alternate Flows become a set of discrete Alternate

Flows or that there are other relevant Customer Data Issues that should be handled. These concerns will often be ignored when addressing a single flow in isolation.

As you become more familiar with Use Cases and Use-Case Slices you may even start to discover the slices before detailing out the individual flows.

The final step is to ensure that our slices are actionable by our development teams - we need to turn them into the right types and size of work item to fit into their team backlogs and plans.

Note: As use cases and use-case slices are focused on end-to-end value they will often involve integrating changes to multiple parts of the system and often the work of multiple teams.

The two most common types of work item we see in use now are User Stories and Tasks. Regardless of the types of Work Item to be generated the basic principle is the same. To bring this idea to life we will take a quick look at using Use-Case 3.0 with User Story and Task based work management approaches.

In this case we only have a single small team working on the system and so we don't need to worry about splitting the work across multiple teams.

So, taking our first slice and a User Story Approach we could get the team together and brainstorm the User Stories needed to action the slice. **FIGURE 7** shows the results from the initial brainstorm.

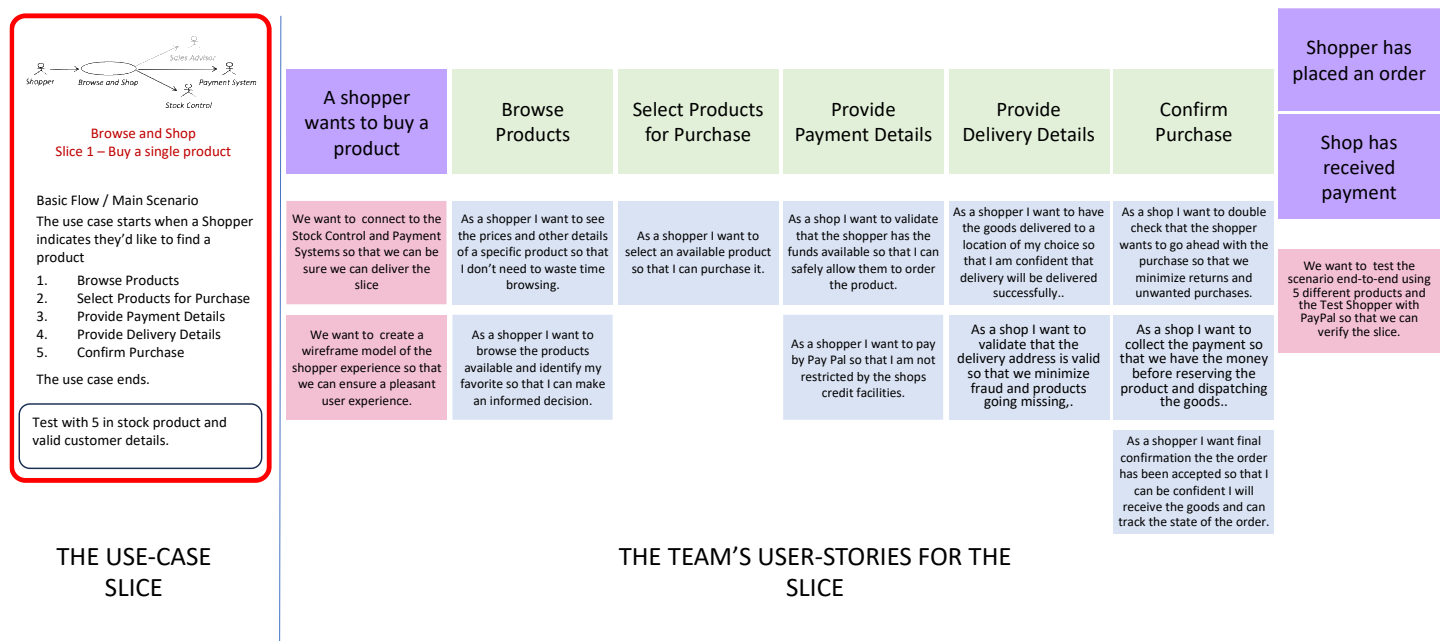


FIGURE 7: BRAINSTORMING USER STORIES FOR A SLICE

There are some interesting things to observe about this set of stories including:

1. There are some technical stories to prepare the ground for implementing the steps in the flow (these are the two red stories at the left of the figure). In this case the team has decided to do a technical spike to integrate the two supporting systems and to do a design spike to create some wireframes to look at the shopping experience.
2. There is a technical story to ensure the end-to-end integration testing of the slice as a whole. (this is the red story on the right of the figure). This will be in addition to the developer testing of each individual story. If you are adopting a test-driven approach the creation and automation of these tests will be the first thing the team does.

If you look closely, you might find issues and problems with this set of User Stories such as missing or incorrect stories or spikes. This is deliberate. This is exactly the sort of conversation that we want the use-case slice to start. Remember this is the development team taking ownership of the delivery of the slice and demonstrating amongst other things their understanding of the slice, their approach to its development and the amount of work they expect it to be. This will typically be done as a whole team event where the team interact with the subject matter experts in, and creators of, the use-case slice. By laying out the User Stories in relationship to the flows being addressed we keep the development team connected to the original use case.

If adopting a more task driven approach, then you could end up with something more like this:

- Task 1 - Screen Design
- Task 2 - Implement Stock Control Interface
- Task 3 - Implement Payment System Interface
- Task 4 - Set up the Middleware
- Task 5 - Code and Unit Test - UI
- Task 6 - Code and Unit Test - Business Objects
- Task 7 - Integration Test
- Task 8 - System Test

As a result of undertaking some analysis and planning to prepare the slice for implementation for example producing some overview diagrams such as a wire frame and identifying the components to be added and changed

The important thing to note here is that Use-Case 3.0 will support, focus and complement whatever planning, work management and development approaches your teams have selected.

Exploring the Principles

In this section we look at the underlying principles in more detail and use them to introduce the ideas behind use-case storytelling, use-case modeling, and use-case driven development.

Principle 1: Universally Applicable

Use Cases apply to systems of all types and sizes: businesses, IT systems, physical systems, or any combinations thereof.

Although traditionally associated with capturing requirements for software systems use cases can be used for many other purposes and for many other types of system.

For example:

- Use Cases actually originated at Ericsson where they were initially used in the development of physical switching systems
- Business Use Cases have been used for many years to help understand what a business does and how it does it.

In fact, all we need to be able to benefit is system of interest that we intend to build, analyze, or change. A system that has some users that want to use it to achieve their goals.

Use cases can be used to understand any kind of system including:

- Organizations
- Businesses
- IT Systems
- Physical Systems
- Infrastructure

In fact, any system as long as it has some actors that it interacts with.

This universal applicability is incredibly powerful as it:

1. Allows us to provide a consistent viewpoint on all our systems regardless of the type of system it is. This makes all our work more accessible to our stakeholders.
2. We can start to understand the usage and context of our proposed solution before we decide what type of system we should be build.

Principle 2: Start with the big picture

Use cases help you understand the big picture: the system's purpose and how it will be used.

Whether the system you are developing is large or small, whether it is a software system, a hardware system, or a new business, it is essential that you understand the big picture.

In our example above we started with a simple user goal and a use case exploring all the ways the system could be used to achieve that particular goal.

Each use case presents a big picture of one aspect of the system and is often enough to get started. In the example above if there are no shoppers turning up to browse and shop then our online shop is of no value.

We can then move on to consider other users and their goals until we have created a usable system of clear value to its users.

In many cases this will be insufficient and without an understanding of the system as a whole you will find it impossible to make the correct decisions about what to include in the system, what to leave out of it, what it will cost, and what benefit it will provide. This doesn't mean capturing all the requirements up front. You just need to create something that sums up the desired system and lets you understand scope and progress at a system level.

A use-case diagram is a simple way of presenting an overview of a system's requirements. Figure 1 shows the use-case diagram for a simple telephone system. From this picture you can see all the ways the system can be used, who starts the interaction, and any other parties involved. For example, a Calling Subscriber can place a local call or a long-distance call to any of the system's Callable Subscribers. You can also see that the users don't have to be people but can also be other systems, and in some cases both (for example the role of the Callable Subscriber might be played by an answering machine and not a person).

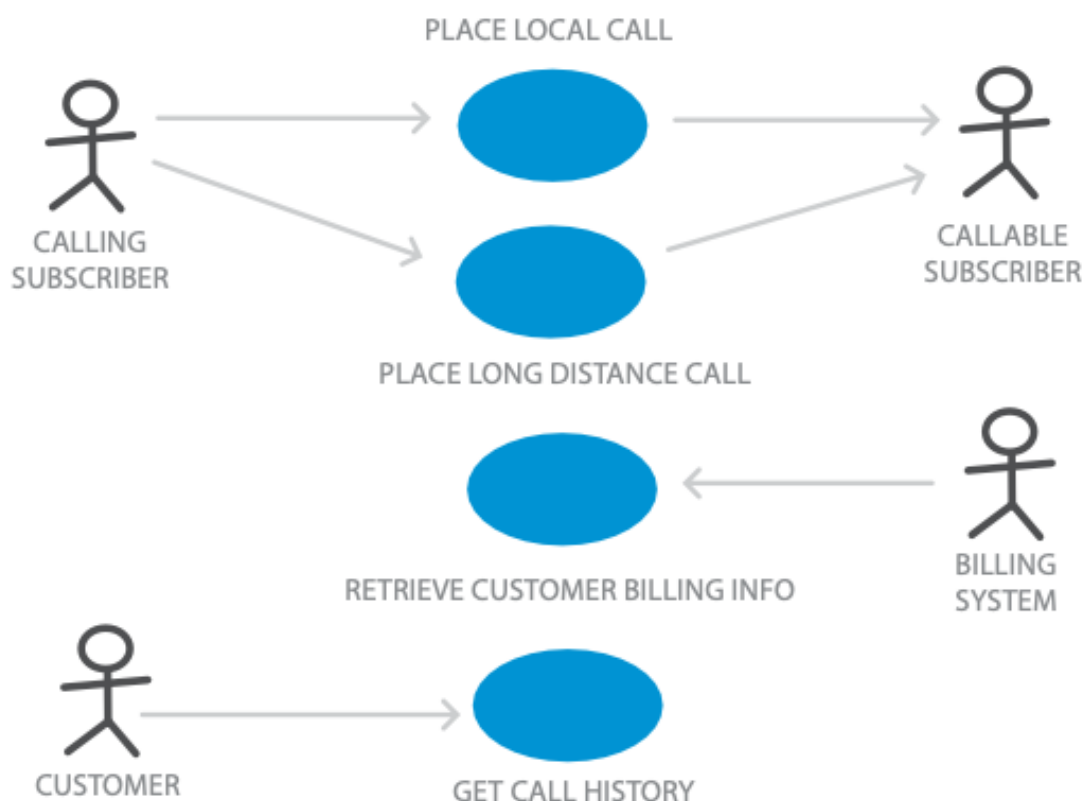


FIGURE 8: A USE-CASE DIAGRAM FOR A SIMPLE TELEPHONE SYSTEM

A use-case diagram is a view into a use-case model. Use-case models acknowledge the fact that systems support many different goals from many different stakeholders. In a use-case model the stakeholders that use the system and contribute to the completion of the goals are modeled as actors, and the ways that the system will be used to achieve these goals are modeled as use cases. In this way the use-case model provides the context for the system's requirements to be discovered, shared, and understood. It also provides an easily accessible big picture of all the things the system will do. In a use-case diagram, such as [FIGURE 8](#), the actors are shown as stick-people and the use cases as ellipses. The arrowheads indicate the initiator of the interaction (an Actor or the System) allowing you to clearly see who starts the use case.

A use-case model is a model of all the useful ways to use a system. It allows you to very quickly scope the

system - what is included and what is not - and give the team a comprehensive picture of what the system will do. It lets you do this without getting bogged down in the details of the requirements or the internals of the system. With a little experience it is very easy to produce use-case models for even the most complex systems, creating an easily accessible big picture that makes the scope and goals of the system visible to everyone involved.

Principle 3: Focus on value

Use cases focus on value: the users' goals and how best to achieve them.

When trying to understand how a system will be used it is always important to focus on the value it will provide to its users and other stakeholders. Value is only generated if the system is actually used; focus on how the system will be used and avoid creating long lists of the functions or features it will offer.

Use cases provide this focus by concentrating on how the system will be used to achieve a specific goal for a particular user. They encompass many ways of using the system; those that successfully achieve the goals, and those that handle any problems that may occur. To make the value easy to quantify, identify and deliver you need to structure the use-case narrative. To keep things simple start with the simplest possible way to achieve the goal. Then capture any alternative ways of achieving the goal and how to handle any problems that might occur whilst trying to achieve the goal. This will make the relationships between the ways of using the system clear. It will enable the most valuable ways to be identified and progressed up front and allow the less valuable ones to be added later without breaking or changing what has gone before.

In some cases, there will be little or no value in implementing anything beyond the simplest way to achieve the goal. In other cases, providing more options and specialist ways of achieving the goal will provide the key differentiators that make your system more valuable than your competitors.

FIGURE 9 shows a use-case narrative structured in this way. The simplest way of achieving the goal is described by the basic flow. The others are presented as alternative flows. In this way you create a set of flows that structure and describe the work products (e.g. user stories) and help us to find the test cases that complete their definition.

BASIC FLOW	ALTERNATE FLOWS
1. Insert Card	A1 Invalid Card
2. Validate Card	A2 Non-Standard Amount
3. Select Withdraw Cash	A3 Receipt Required
4. Select Account	A4 Insufficient Funds in ATM
5. Confirm Availability of Funds	A5 Insufficient Funds in Acc't
6. Return Card	A6 Would Cause Overdraft
7. Dispense Cash	A7 Card Stuck
	A8 Cash Left Behind
	etc....

FIGURE 9: THE STRUCTURE OF A USE CASE NARRATIVE

FIGURE 9 shows the narrative structure for the Withdraw Cash use case for a cash machine. The basic flow is shown as a set of simple steps that capture the interaction between the users and the system. The alternative

flows identify any other ways of using the system to achieve the goal such as asking for a non-standard amount, any optional facilities that may be offered to the user such as dispensing a receipt, and any problems that could occur on the way to achieving the goal such as the card getting stuck.

You don't need to capture all the flows at the same time. Whilst recording the basic flow it is natural to think about other ways of achieving the goal, and what could go wrong at each step. You capture these as alternate flows but concentrate on the basic flow first. You can then return to complete the alternate flows later as and when they are needed.

This kind of bulleted outline may be enough to capture the work products and drive the development, or it may need to be elaborated as the team explores the detail of what the system needs to do. The most important thing is the additive structure of the use-case narrative. The basic flow is needed if the use case is ever to be successfully completed; this must be implemented first. The alternatives though are optional. They can be added to the basic flow as and when they are needed. This allows you to really focus on the value to be obtained from the system. You no longer need to deliver the whole use case but can focus on those parts of the use case that offer the most value. It also means that you don't need a complete use-case model or even a complete use case before you start to work on the development of the system. If you have identified the most important use case and understood its basic flow then you already have something of value you could add to your system.

This structure makes the work products easy to capture and validate for completeness, whilst making it easy to filter out those potential ways of using a system that offer little or no real value to the users. This constant focus on value will enable you to ensure that every release of your system is as small as possible, whilst offering real value to the users of the system and the stakeholders that are funding the development.

Principle 4: Involve your stakeholders

Stakeholder involvement is essential: bring all the involved parties together to establish the intent and scope of the system.

The success of use cases has been so widespread that the term use case has even entered common everyday use. For example, the [Meriam Webster](#) on-line dictionary contains the following definition:

use case:

a use to which something (such as a proposed product or service) can be put

There are various *use cases* for the cloud: website hosting, disaster recovery, ... etc.

- Users seem to want single-purpose mobile apps that nail a specific *use case* quickly.—Josh Constine
- The pandemic has pushed the adoption of technology for various *use-cases*.—*businesswire.com*

In fact, it's rare for a new product to be launched these days without some reference to its use cases.

Imagine the concern it must provoke in those organizations where the leaders talk about one set of use cases when publicizing the new system and the developers talk about a different set of use cases, or have no idea what the intended use cases are, when they are building it.

It would be much better if everyone was on the same page - and ideally that page is a use case.

So involve your stakeholders in conceiving both the use-case model and the individual use cases, and keep them involved in their evolution and implementation.

Use cases really are the best way to get everyone on the same page.

Principle 5: Tell the whole story

A use case tells the whole story, as a story, from the initial event to the realization of the value it provides or the eventual failure if it can't be met. It includes how to handle any problems and alternatives that may occur on the way.

Storytelling is how cultures survive and progress; it is the simplest and most effective way to pass knowledge from one person to another. It is the best way to communicate what a system should do, and to get everybody working on the system to focus on the same goals.

The use cases capture the goals of the system. To understand a use case we tell stories. The stories cover how to successfully achieve the goal, and how to handle any problems that may occur on the way. Use cases provide a way to identify and capture all the different but related stories in a simple but comprehensive way. This enables the system's requirements to be easily captured, shared and understood.

As a use case is focused on the achievement of a particular goal, it provides a focus for the storytelling. rather than trying to describe the system in one go we can approach it use case by use case. The results of the storytelling are captured and presented as part of the use-case narrative that accompanies each use case.

When using storytelling as a technique to communicate requirements it is essential to make sure that the stories are captured in a way that makes them actionable and testable. A set of test cases accompanies each use-case narrative to complete the use case's description. The test cases are the most important part of a use case's description, more important even than the use-case narrative. This is because they make the stories real, and their use can unambiguously demonstrate that the system is doing what it is supposed to do. It is the test cases that define what it means to successfully implement the use case.

Principle 6: Trigger conversations

Use cases trigger conversations: while discussing the possible alternate flows, you and your co-writers will think of missing steps and missing alternatives. These conversations help you find situations that often get overlooked.

Use cases and use-case models are intended to trigger conversations - one of the most powerful things about use cases is that they put the conversations into context.

We've already discussed how a simple outline of a use case will trigger conversations about the steps in the use case and the coverage of the alternative flows.

Now you may think this sounds quite trivial and that these things are boring and easy to capture but bear this in mind:

We were once running a use-case workshop for one of the world's biggest banks who were in the process of building a new platform for electronic funds transfer to help them seamlessly integrate all the various banks they had been acquiring around the world.

As is not unusual at this kind of event the techies sat together on one table and all the business people sat together on another. One of the introductory exercises we did was to outline the basic flow of the Transfer Funds Use Case. A simple exercise you'd have thought but when it came time to share their outlines the techies stood up and walked through the steps of their basic flow. Much to their, and our, amazement the business people just started laughing. What could be so funny? They had just presented the steps involved in the existing automated process for the main bank. Well, it turned out that they were in the wrong order. "Do you know how many millions that has cost the bank?" asked the main business sponsor.

Well clearly they didn't, as it was exactly what they had intended to implement in the new system.

Luckily the outline had triggered a new conversation and brought everyone involved much closer together in their understanding of the intent of the new system.

Just as a use case can start a conversation about the ways that a user goal should be addressed, the use-case model starts conversations about the scope of the system as a whole. For example, is there anything missing, are we doing too much, is everything identified really needed, what do we need to have in place to go live, what would be a good MVP (minimal viable product), are we talking to the right actors and so on. They can also be used to trigger conversations with customers and to support the business case.

Principle 7: Prioritize readability

Prioritize readability: the goal is to communicate the big picture to everyone involved, generating comments, spotting any gaps, and getting their buy-in.

It's worth taking a few moments to think about how we typically present the requirements for our development work.

On one hand we have the agile teams who like to capture everything on post it notes. This often results on a huge number of small requests being captured and collected into a prioritized list. Individually these are small and accessible but can make it very difficult to get an overview of what is being built, what has been achieved so far and what there is left to do.

On the other hand, we have more waterfall projects where traditionally all the detailed requirements are captured up front before development starts. This often results in the production of a huge impenetrable document that takes hours to read and once again obscures the big picture.

Use cases done well hit the sweet spot in between these two extremes but you need to prioritize readability in both the use-case model and the use cases. Even when, for legal or other reasons, you need to capture a lot of detail in your use cases you should make sure that you keep them readable and start with a simple one page overview of the flow of events.

Principle 8: Just enough, just in time

The amount of detail and the format used will vary to match your circumstances: You can start with a sketch of the flow of events and add detail as needed.

Unfortunately, there is no 'one size fits all' solution to the challenges of system development; different teams and different situations require different styles and different levels of detail. Regardless of which practices and techniques you select you need to make sure that they are adaptable enough to meet the ongoing needs of the team.

This applies to the practices you select to share your intent, capture your requirements and drive the development as much as any others. For example, lightweight requirements are incredibly effective when there is close collaboration with the users, and the development team can get personal explanations of the requirements and timely answers to any questions that arise. If this kind of collaboration is not possible, because the users are not available, then the requirements will require more detail and will inevitably become more heavyweight. There are many other circumstances where a team might need to have more detailed requirements as an input to development. However, what's important is not listing all of the possible circumstances where a lightweight approach might not be suitable but to acknowledge the fact that practices need to scale.

Use-Case 3.0 is designed with this in mind and is as light as you want it to be. Small, collaborative teams can have very lightweight use-case narratives that capture the bare essentials of the requirements. These can be handwritten on simple index cards. Large, distributed teams can have more detailed use-case narratives presented as documents. It is up to the team to decide whether they need to go beyond the essentials, adding detail in a natural fashion as they encounter problems that the bare essentials cannot cope with.

The structure of use cases also lends itself to adding the detail just-in-time at the last responsible moment. For example, although we like to identify all the in-scope use cases up front we don't need to detail their flow of events unless they are needed. The same goes for the alternate flows, the detailing of these can be deferred until we are going to work on them.

Principle 9: Implement in stages

A use case can be implemented in stages: develop and put into place some key flows of a use case early to capture value and feedback, add less used or less critical flows over time strategically.

Most systems evolve through many generations. They are not produced in one go; they are constructed as a series of releases - each building on the one before. Even the releases themselves are often not produced in one go but are evolved through a series of increments. Each increment provides a demonstrable or usable version of the system. Each increment builds on the previous increment to add more functionality or improve the quality of what has come before. This is the way that all systems should be produced.

The use cases themselves can also be too much to consider delivering all at once. For example, we probably don't need all the ways of placing a local call in the very first increment of a telephone system. The most basic facilities may be enough to get us up and running. The more optional or niche ways of placing a local call such as reversing the charges or redialing the last number called can be added in later increments. By slicing up the use cases we can achieve the finer grained control required to maximize the value in each release.

FIGURE 10 shows the incremental development of a release of a system. The first increment only contains a single slice: the first slice from use case 1. The second increment adds another slice from use case 1 and the first slice from use case 2. Further slices are then added to create the third and fourth increments. The fourth increment is considered complete and useful enough to be released.

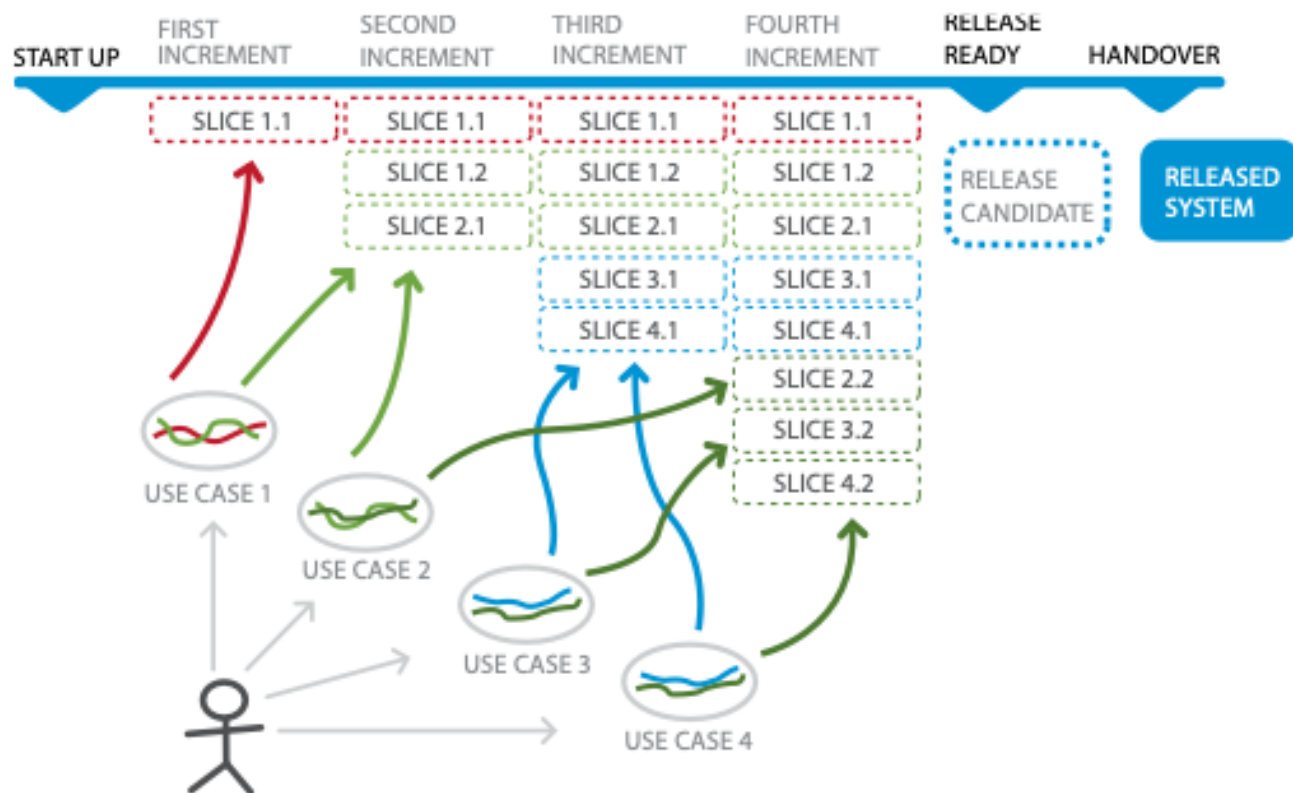


FIGURE 10: USE CASES, USE-CASE SLICES, INCREMENTS AND RELEASES

Use cases are a fabulous tool for release planning. Working at the use-case level allows whole swathes of related requirements to be deferred until the later releases. by making decisions at the use-case level you can quickly sketch out the big picture and use this to focus in on the areas of the system to be addressed in the next release.

Use-case diagrams, showing which use cases are to be addressed in this release and which are to be left until a later release, are a great tool for illustrating the team's goals. They clearly show the theme of each release and look great pinned up on the walls of your war-room for everybody to see.

Use-case slices are a fabulous tool for building smaller increments on the way to a complete release. They allow you to target independently implementable and testable slices onto the increments ensuring that each increment is larger than, and builds on, the one before.

Use-Case Slices: The most important part of Use-Case 3.0

The concept of a use-case slice is as integral to Use-Case 3.0 as the use case itself. It is the slices that enable the use cases to be broken down and delivered in stages. Imagine that you are part of a small team producing working software every two weeks. A whole use case is probably too much to be completed in one two-week period. A use-case slice though is another matter because it can be sliced as thinly as the team requires. Use-case slices also allow the team to focus on providing a valuable, usable system as soon as possible, shedding all unnecessary requirements on the way.

A simple recipe for success

Applying our recipe above, the use cases identify the useful things that the system will do. Select the most useful use case to find the most useful thing that the system does. To find the most central slice you will need to shed all the less important ways of achieving the goal and handling problems. You can do this by focusing on the basic flow. A slice based on the basic flow is guaranteed to travel through the entire concept from end-to-end as it will be the most straightforward way for the user to achieve their goal.

Estimate the slice and start to build it. Additional slices can then be taken from the use case until there are enough slices to provide this particular user with a usable solution. The same can then be done for any other use cases you need to complete a usable system.

Slicing up the use cases

The best way to find the right slices is to think about the flows and how we capture them. Each flow is a potential slice. Each slice is defined by part of the use-case narrative and one or more of the accompanying test cases. It is the test cases that are the most important part of the use-case slice's description because they make it a verifiable deliverable.

A use-case slice isn't limited to a single flow - often a use-case slice will focus on a number of conceptually related flows - for example handling all the potential card handling errors in our example cash withdrawal use case. In fact we may create the slice before we know the details of exactly how many flows it will contain.

A use-case slice doesn't need to contain an entire flow and all its test cases - the first slice might just be the basic flow and one test case. Additional slices can then be added to complete the flow and address all the test cases.

The slicing mechanism is very flexible enabling you to create slices as big or small as you need to drive the incremental development of your system.

Principle 10: Build the system in slices

A system can be developed in slices where each slice is one or more paths through one of the system's use cases plus the relevant design, code and tests used to implement and verify them.

Most systems require a lot of work before they are usable and ready for operational use. They have many requirements, most of which are dependent on other requirements being implemented before they can be fulfilled, and value delivered. It is always a mistake to try to build such a system in one go. The system should be built in slices, each of which has clear value to the users.

The recipe is quite simple. First, identify the most useful thing that the system must do and focus on that. Then take that one thing and slice it into thinner slices. Decide on the test cases that represent acceptance of those slices. Some of the slices will have questions that can't be answered. Put those aside for the moment. Choose the most central slice that travels through the entire concept from end to end, or as close to that as possible. Estimate it as a team (estimates don't have to be "right", they're just estimates), and start building it.

This is the approach taken by Use-Case 3.0, where the use cases are sliced up to provide direction to the team, and where the system itself is evolved slice by slice.

The slices are more than just requirements and test cases

When we build the system in slices it is not enough to just slice up the requirements. Although use cases have traditionally been used to help understand and capture requirements, they have always been about more than this. As shown in [FIGURE 11](#), the use-case slices slice through more than just the requirements, they also slice through all the other aspects of the system and its documentation.

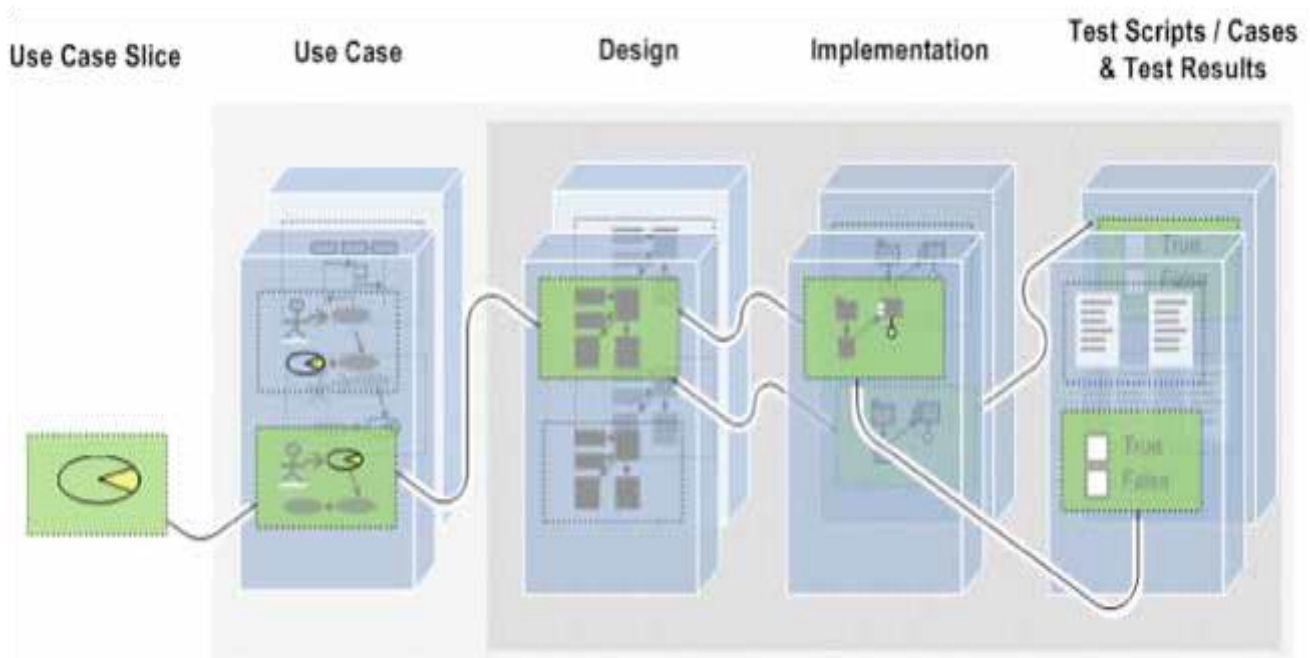


FIGURE 11: A USE-CASE SLICE IS MORE THAN JUST A SLICE OF THE USE CASE

On the left of [FIGURE 11](#) you can see the use-case slice, this is a slice taken from one of the use cases shown in the next column. The slice then continues through the design showing the design elements involved, and through the implementation where you can see which pieces of the code implement the slice. Finally, the slice cuts through the test assets, not just encompassing the test cases, but also the test scripts used to execute the test cases and the test results generated.

As well as providing traceability from the requirements to the code and tests, thinking of the slices in this way helps you develop the right system. When you come to implement a slice, you need to understand the impact that the slice will have on the design and implementation of the system. Does it need new system elements to be introduced? Can it be implemented by just making changes to the existing elements? If the impact is too great you may even decide not to implement the slice! If you have the basic design for the system this kind of analysis can be done easily and quickly and provides a great way to understand the impact of adding the slice to the system.

By addressing each aspect of the system slice by slice, use cases help with all the different areas of the system including user experience (user interface), architecture, testing, and planning. They provide a way to link the requirements to the parts of the system that implement them, the tests used to verify that the requirements have been implemented successfully, and the release and project plans that direct the development work. In Use-Case 3.0 there is a special construct, called the use-case realization, which is added to each use case to record its impact on the other aspects of the system.

Use-Case 3.0 Content

This section considers Use-Case 3.0 as a whole. The specific practices are available [here](#).

Use-Case 3.0 consists of a set of things to work with and a set of things to do.

Things to Work With

The subject of Use-Case 3.0 is the requirements, the system to be developed to meet the requirements, and the tests used to demonstrate that the system meets the requirements. At the heart of Use-Case 3.0 are the use case and the use-case slice. These capture the requirements and drive the development of the system. Figure 5 shows how these concepts are related to each other. It also shows how changes and defects impact on the use of Use-Case 3.0.

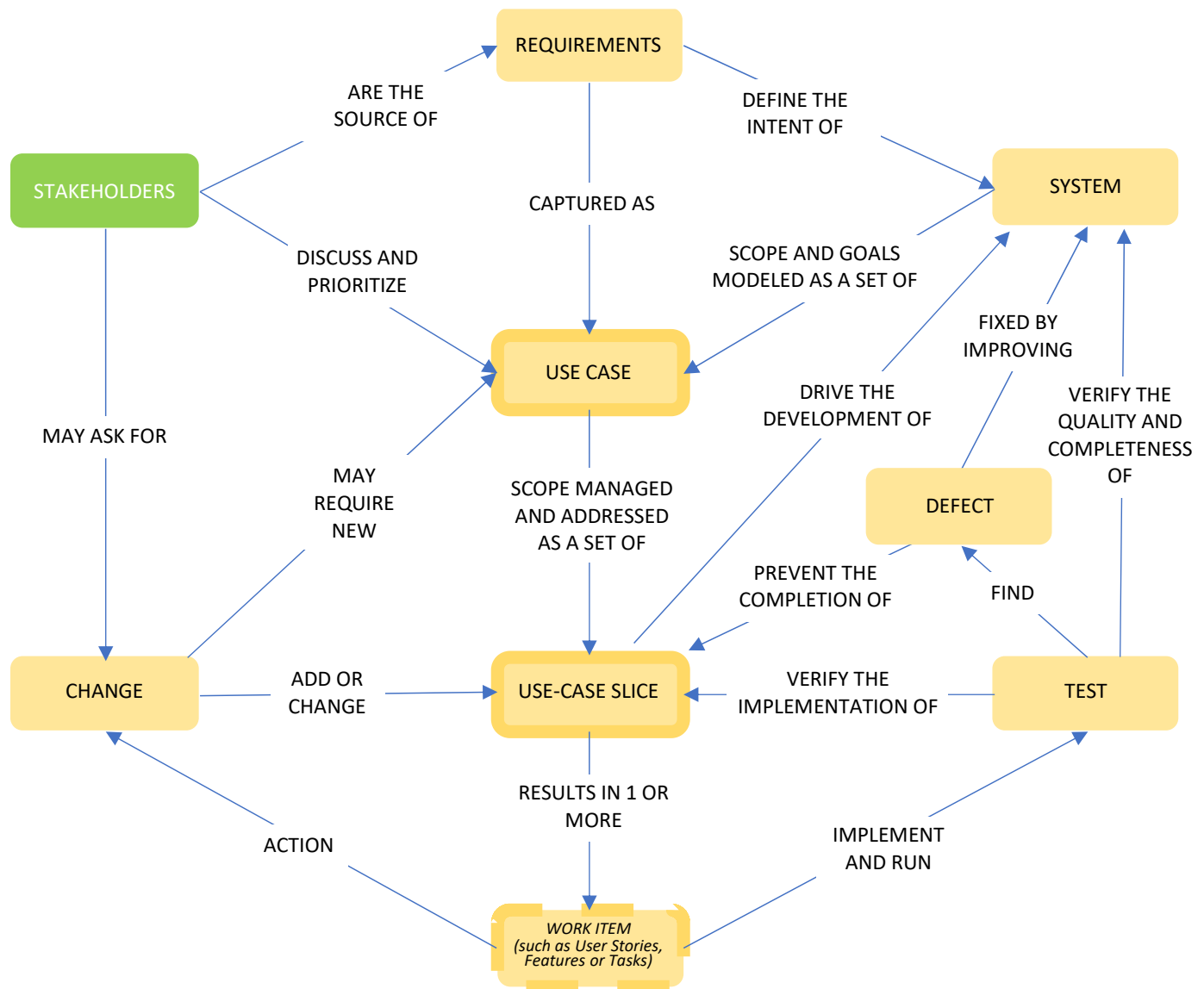


FIGURE 12: USE-CASE 3.0 CONCEPT MAP

The stakeholders are the people, groups, or organizations who affect or are affected by a software system. The requirements are what the system must do to satisfy the stakeholders. It is important to discover what is needed from the software system, share this understanding among the stakeholders and the team members, and use it to drive the development of the new system. In Use-Case 3.0 the requirements are captured as a set of use cases, which are scope managed and addressed as a set of use-case slices. Any changes requested by the stakeholders result in additions or changes to the set of use cases and use-case slices.

The system is the system to be built. It is typically a software system although Use-Case 3.0 can also be used in the development of new businesses (where you treat the business itself as a system), and combined hardware and software systems (embedded systems). It is the system that implements the requirements and is the subject of the use-case model. The quality and completeness of the system is verified by a set of tests. The tests also verify whether the implementation of the use-case slices has been a success. If defects are found during testing, then their presence will prevent the completion of the use-case slice until they have been fixed and the system improved.

Storytelling and on-going conversations bridge the gap between the stakeholders, the use cases, and the use-case slices. It is how the stakeholders communicate their requirements and explore the use cases.

To create an actionable plan to implement a use case it typically needs to be broken up into small, actionable pieces of work that can be given to an individual or team to complete.

Most of the use-case slices will require multiple work items to be completed to complete their development and testing. Common forms of work item used to implement a use-case slice include User Stories, Features and Tasks.

For example:

when working in an agile environment and implementing a slice that covers multiple steps in a flow of events then you might implement each step individually as a separate User Story.

when working in a more traditional environment you may wish to create a series of tasks - design, code, unit test, system test - for each slice to allocate to the various team members.

Note: if the use-case slice is simple and straight forward enough to be easily implemented and tested within a couple of days then there may only be the need to create a single work item.

This separation of concerns allows you to seamlessly use Use-Case 3.0 to complement the way that your development team currently works. Benefiting from the power of use cases with minimal disruption to your team.

Use Cases

A use case is all the ways of using a system to achieve a goal of a particular user. Taken together the set of all the use cases gives us all of the useful ways to use the system.

A use case is:

- A sequence of actions a system performs that yields an observable result of value to a particular user.
- The specific behavior of a system which participates in a collaboration with a user to deliver something of value for that user.
- A collection of all the successful, challenged and failure paths.
- The smallest unit of activity that provides a meaningful result to the user.
- The context for a set of related conversations and requirements.
- Independent of implementation, technology, and platform.
- Described textually and / or visually.

To understand a use case, we describe it as a network of flows. The flows cover both how to successfully achieve the goal and how to handle any problems that occur on the way. They help us to understand the use case and implement it slice by slice.

As **FIGURE 13** shows, a use case undergoes several state changes from its initial identification through to its fulfillment by the system. The states constitute important way points in the understanding and implementation of the use case indicating:

1. Goal Established: the goal of the use case has been established and its value is clear.
2. Flow Structure Understood: the structure of the use-case's flow of events has been understood enough for the team to start work identifying and implementing the first use-case slices.
3. Basic Flow Enabled: when the use case's basic flow can be successfully traversed (using the system itself) from beginning to end.
4. Sufficient Flows Fulfilled: the system fulfills enough of the use case's flows to provide a usable solution.
5. All Flows Fulfilled: when the system fulfills all the use case's flows and delivers all the ways of using the system to achieve the user's goals.

This will be achieved by implementing the use case slice-by-slice. The states provide a simple way assess the progress you are making in understanding and implementing the use case.

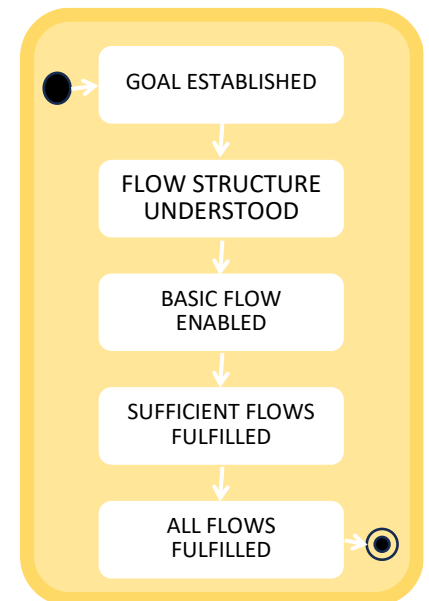


FIGURE 13: THE LIFECYCLE OF A USE CASE

Use-Case Slices

A slice of a use case that provides clear value to the user or other stakeholders. Typically, a use-case slice captures one of the ways of using a system to achieve a goal.

Use cases cover all the ways of achieving a goal of a particular actor. These will be of varying importance and priority. There are often too many flows to deliver in a single release and generally too many to work on in a single increment. Because of this we need a way to divide the use case into smaller slices of value that 1) allow us to deliver the use case in stages whilst ensuring that each stage is usable and valuable, 2) provide a suitable unit for development and testing by the development team, and 3) allow us to have small and similarly sized slices of value that flow quickly through development.

A use-case slice captures one or more paths through a use case that are of clear value to the customer. They act as a placeholder for all the work required to deliver the value implicit in the slice. As we saw earlier when we discussed how the use-case slices are more than just requirements and test cases, the use-case slice evolves to include the corresponding slices through design, implementation, and test.

The use-case slice is the most important element of Use-Case 3.0, as it is not only used to help with the requirements but to drive the development of a system to fulfill them.

Each use-case slice:

- Is always 'end-to-end'. It starts at the beginning of the use-case and ends at its end.
- Traverse one or more of the use-case's flows.
- Is always testable and complement the flow of events with explicit test cases.
- Enables use cases to be broken up into smaller, independently deliverable units of clear value.
- Enables the requirements contained in a set of use cases to be ordered, prioritized, and addressed in parallel.
- Links the different system models (requirements, analysis, design, implementation, and test) used in use-case driven development.

As **FIGURE 14** shows, a use-case slice undergoes several state changes from its initial identification through to its final acceptance. The states constitute important way points in the understanding, implementation and testing of the use-case slice indicating:

1. Identified: the slice has been scoped and the extent of the flows and other requirements covered is known.
2. Defined: the extent of the slice has been clarified by identifying the paths through the use case and how they would be verified. This is done by elaborating the flow of events and identifying a set of test cases to clearly define what it means to successfully implement the slice.
3. Analyzed: the slice has been analyzed so its impact on the components of the system is understood.
4. Prepared: the work items required to deliver an implementation of the slice have been quantified and added to a team's backlog and / or plans.
5. Implemented: the software system has been enhanced to implement ready for testing.
6. Verified: the slice has been verified as done and is ready for inclusion

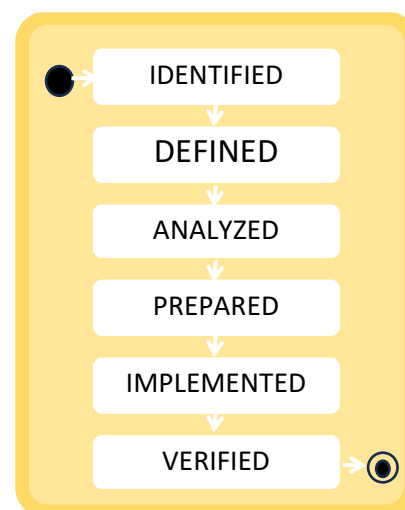


FIGURE 14: THE LIFECYCLE OF A USE-CASE SLICE

The states provide a simple way to assess the progress you are making in understanding and implementing the use-case slices. They also denote the points when the slice is at rest and could potentially be handed over from one person or team to another. To the casual observer glancing at the states, this might look like a waterfall process: identification > definition > analysis > implementation > verification. There's a big difference, though. In a waterfall approach, all the requirements are prepared before the analysis starts, and all the analysis is completed before the implementation starts, and all the implementation is completed before the verification starts. Here we are dealing with an individual use-case slice. Across the set of slices all the activities could be going on in parallel. While one use-case slice is being verified, another use-case slice is being implemented, a third is being prepared, and a fourth being analyzed. In the next chapter we will explore this more when we look at using Use-Case 3.0 with different development approaches.

Why Use Cases, Use-Case Slices and Flows?

A use case is all the ways of using a system to achieve a goal of a particular user.

All these ways, both successful and unsuccessful, are captured in the use case's flow of events. This provides a map of all the ways of using the system when conducting the use case, and just like a map there are many possible paths through the flow of events.

The use case always starts with the user doing something that the system can detect. This is the first step of the basic flow and the first step of all the paths through the use case. Regardless of the path through the use case we always end up at the final step of the basic flow where the use case ends.

FIGURE 15 illustrates how even for a simple use case with just a few flows there are an exponentially greater number of paths through the use case.

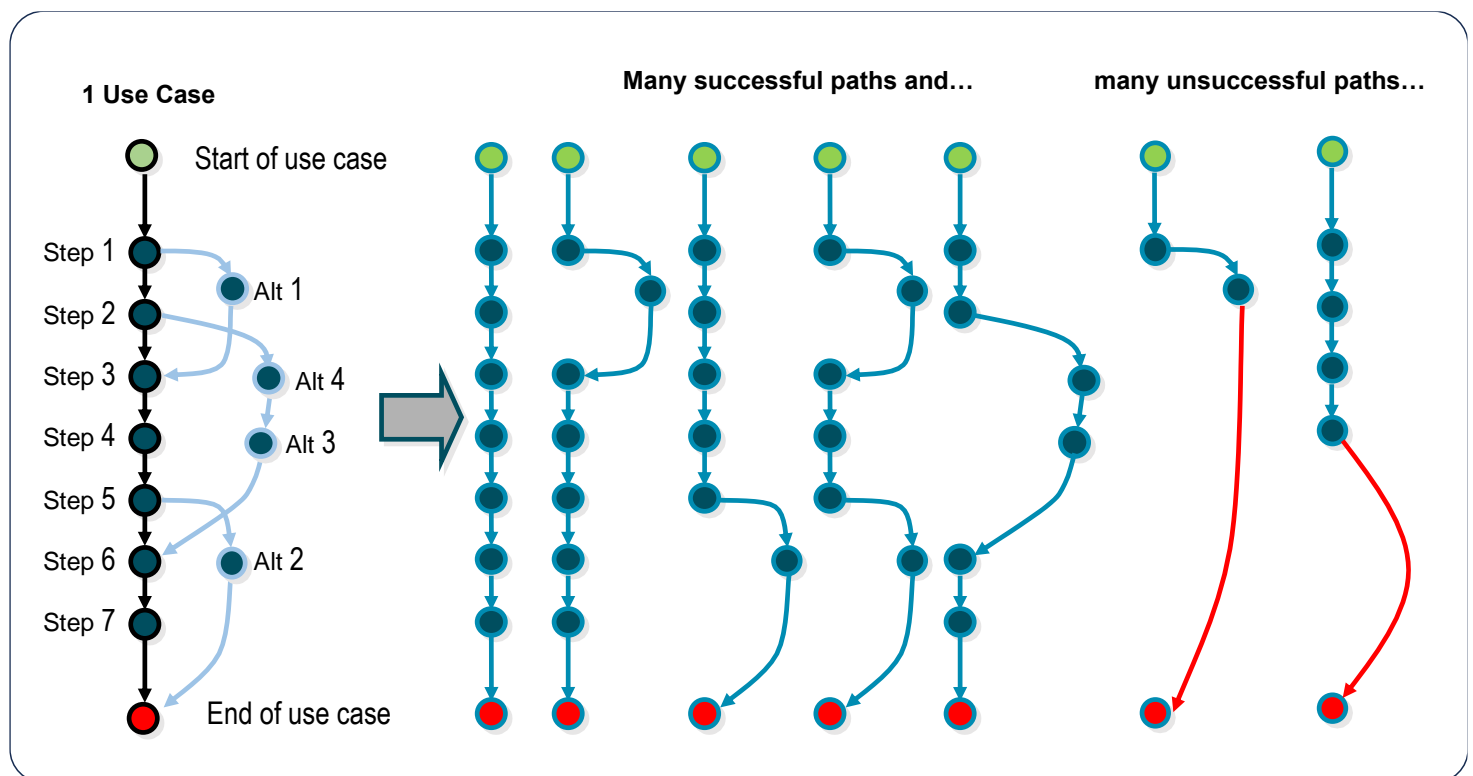


FIGURE 15: THERE ARE MANY PATHS THROUGH A USE CASE

On the left of the figure the basic flow is shown as a linear sequence of steps and the alternative flows are shown as detours from this set of steps. The alternative flows are always defined as variations on the basic flow. On the right of the diagram some of the paths through the use case are shown. Each path traverses one or more flows starting with the first step of the basic flow and terminating with the use case at the end of the basic

flow. This ensures that all the paths are related to the achievement of the same goal, are complete and meaningful, and are complementary as they all build upon the simple story described by the basic flow. For each path there will be one or more test cases.

The key to finding effective slices is to understand the structure of the use case and then to group the paths and their test cases into meaningful slices of value that can be used to 1) conduct the implementation on the use case in stages and 2) to scope manage the use case and focus on the quick delivery of value. The smallest a slice can be is a single test case traversing a single path through the use case. For example, it might be too much to start with the whole of the basic flow and you may choose to start with a single end-to-end test case and an experimental solution with hard coded values and no error handling.

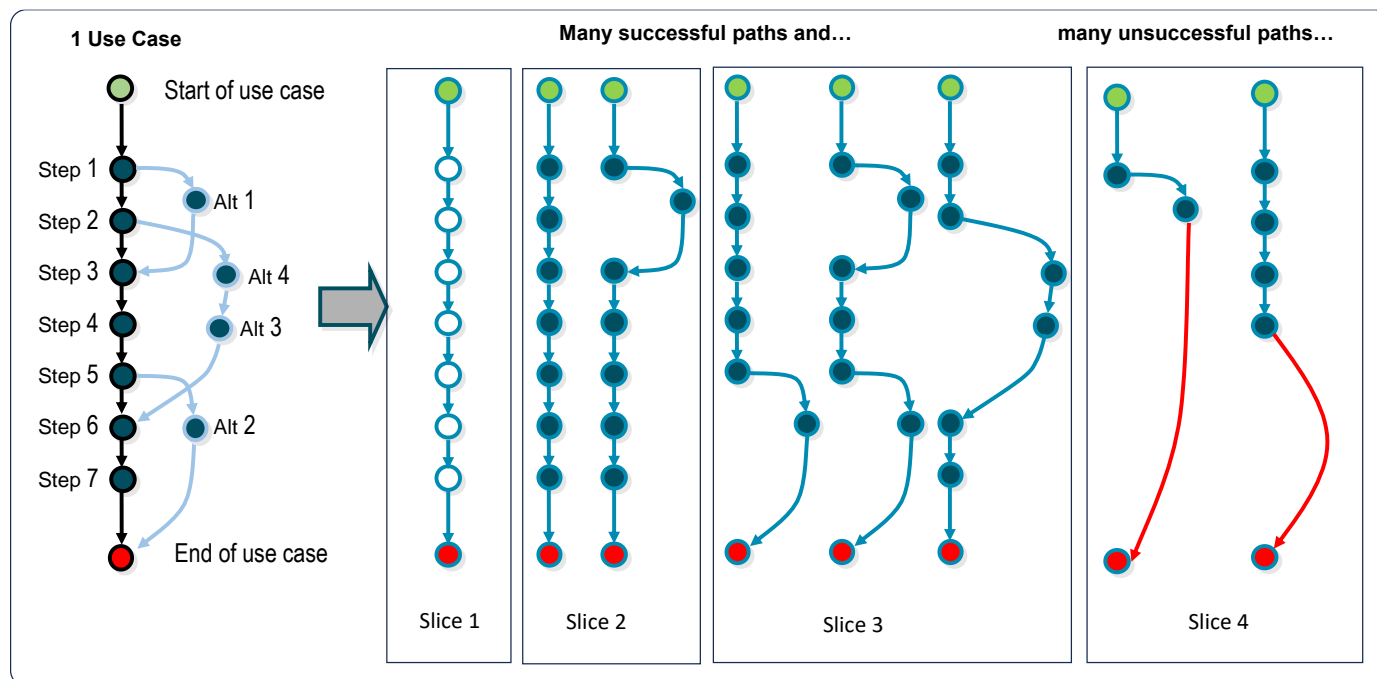


FIGURE 16: SLICING A USE CASE

FIGURE 16 illustrates one way of slicing up the use case with the first slice demonstrating the basic flow, the second slice completing the basic flow and alt1, the third slice addressing the other alternate flows and the fourth handling exceptions that can't be addressed in collaboration with the user.

Note: Slices can be functional or non-functional in nature. For example, we could create additional slices to address performance and / or loading requirements. These wouldn't require new flows but would be additional slices with additional test cases.

There are two common approaches to identifying the use-case's flow of events and create their use-case slices:

Top Down: Some people like to take a top-down approach where they 1) identify the use case, 2) outline the steps of the basic flow, and 3) brain-storm alternative flows based on the basic flow. This structures the use case and allows them to identify their slices.

Bottom Up: Using the bottom-up approach, we start by brainstorming some user goals and / or user stories and then grouping these by theme to identify our use cases. The input ideas are then examined to help us identify the basic, and some of the alternative, flows. The use-case structure then leads us to identify any flows or steps, and make sure that all the flows are well-formed and complementary.

You should pick the approach that works best for your stakeholders. You can of course combine the two

approaches and work both top-down, from your use cases, and bottom up from any suggested new user stories or change requests.

Defects and Changes

Although not a direct part of Use-Case 3.0, it is important to understand how defects and changes are related to the use cases and the use-case slices.

Changes requested by the stakeholders are analyzed against the current use-case model, use cases, and use-case slices. This enables the extent of the change to be quickly understood. For example, adding a new use case to the system is a major change as it changes the system's overall goals and purpose; whereas a change to an existing use case is typically much smaller, particularly if it is to a flow that has not been allocated to a slice and analyzed, prepared, implemented or verified.

Defects are handled by tracking which use-case slices, and by implication which test cases, resulted in their detection. If they are found during the implementation or verification of a use-case slice, then that slice cannot advance until the defect is addressed and the test can be passed. If they are found later during regression testing then the relationship between the failed test cases and the use cases allows you to quickly discern what impact the defect will have on the users and the usability of the system.

Work Products

The use cases and the use-case slices are supported by a number of work products that the team uses to help share, understand, and document them. **FIGURE 17** shows the five Use-Case 3.0 work products (in light yellow with gray borders) and their relationships to the requirements, use cases, use-case slices, tests, and the system (in dark yellow)

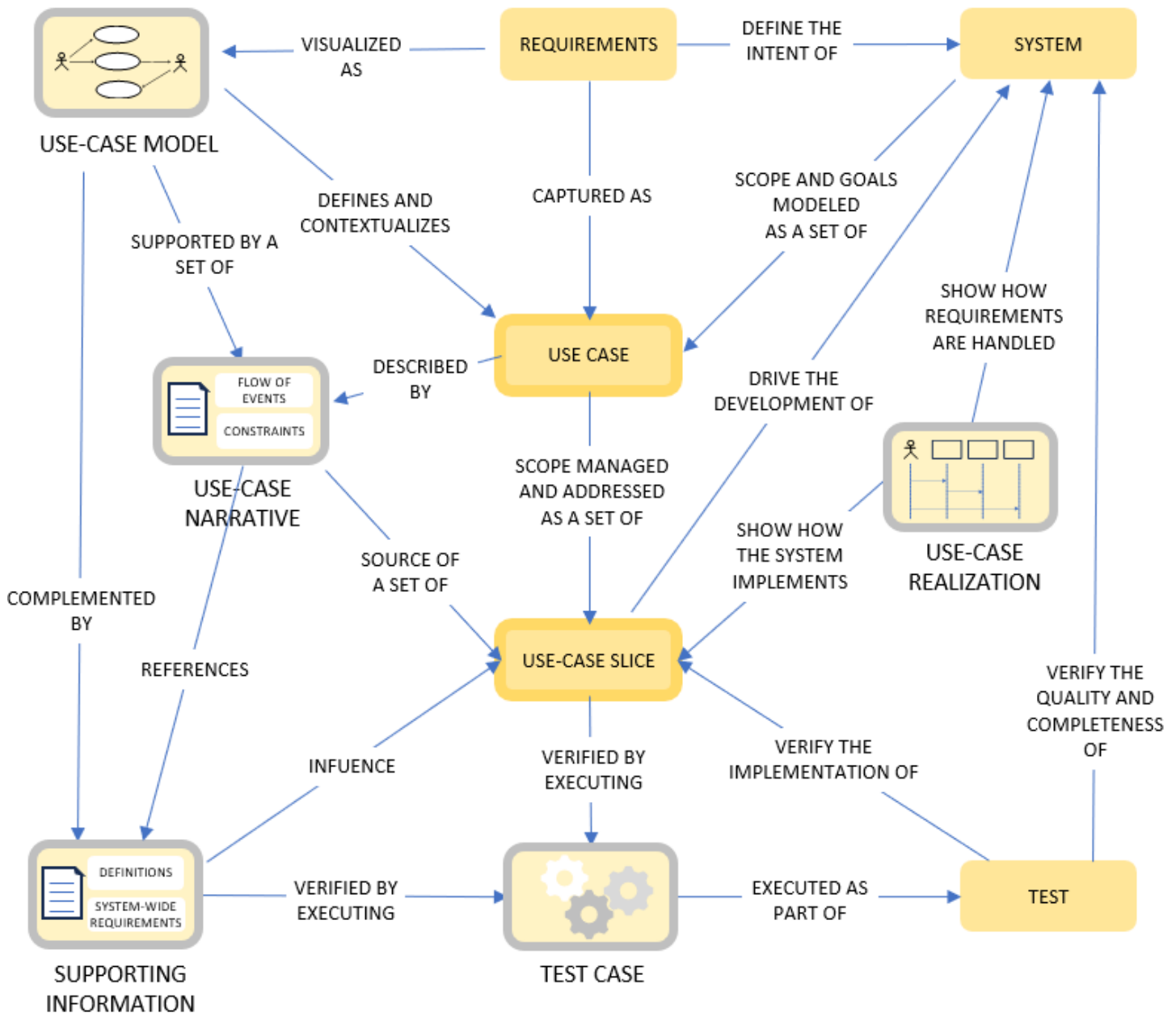


FIGURE 17: THE USE-CASE 3.0 WORK PRODUCTS

The use-case model visualizes the requirements as a set of use cases, providing an overall big picture of the system to be built. The model defines the use cases and provides the context for the elaboration of the individual use cases. The use cases are explored by outlining and discussing the flow of events and any constraints that apply to them. Each use case is described by 1) a use-case narrative that outlines its flows and constraints (such as standards, quality, or performance requirements) and 2) a set of test cases that will be used to verify the use-case and the requirements it captures. Use-Case Slices are used to stage the description, analysis, implementation and testing of the use case.

The use-case model is complemented by supporting information. This captures the definitions of the terms used in the use-case model and when outlining the flows and constraints in the use-case narratives. It also captures any system-wide requirements: those requirements that apply to all the use cases. Again, these will influence the slices selected from the use cases and be assigned to the use-case slices for implementation.

You may be disconcerted by the lack of any explicit work products to capture and document the use-case slices. These are not needed as they are fully documented by the other work products. If required, you can list the slices related to a use case as an extra section in the use-case narrative but this is not essential.

Working with the use cases and use-case slices

As well as creating and tracking the work products, you will also need to track the states and properties of the use cases and the use-case slices. This can be done in many ways and in many tools. The states can be tracked very simply using post-it notes or spreadsheets. If more formality is required one of the many commercially available application lifecycle management, work management, requirements management, change management or defect tracking tools could be used. [FIGURE 18](#) shows a use case and some of its slices captured on a set of sticky notes.

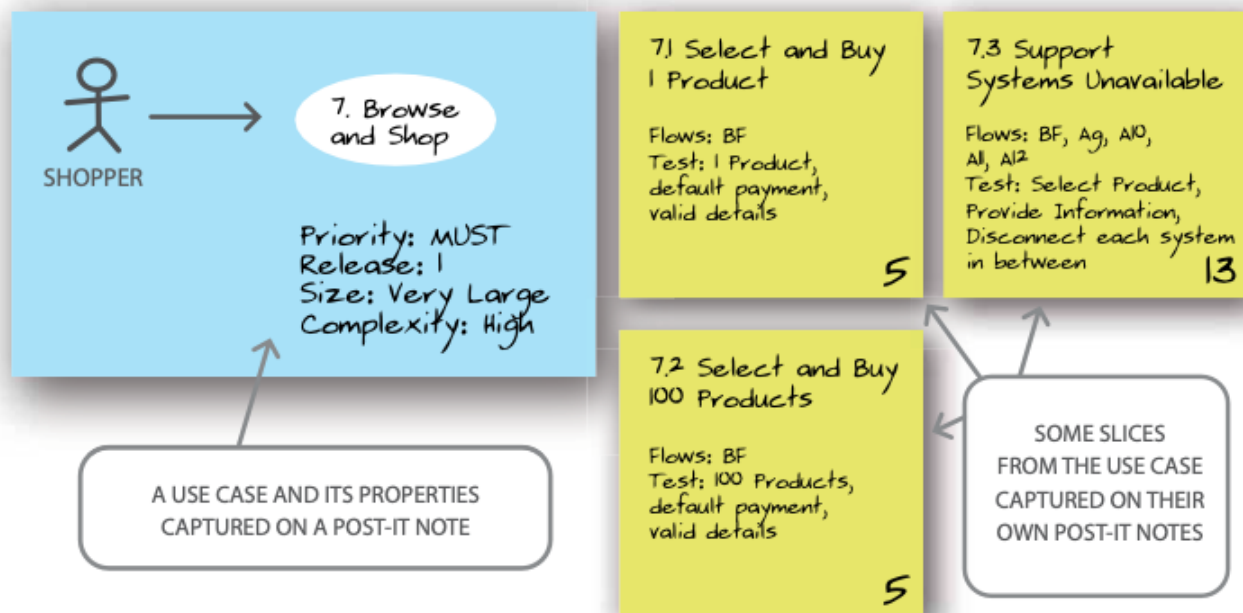


FIGURE 18: CAPTURING THE PROPERTIES OF A USE CASE AND ITS SLICES USING STICKY NOTES

The use case shown is use-case 'Browse and Shop' from an on-line shopping application. Slices 1 and 2 of the use case are derived from the basic flow: 'Select and buy 1 Product' and 'Select and buy 100 Products'. Slice 3 is based on multiple flows covering the availability of the various support systems involved in the use case.

The essential properties for a use case are its name, state, and priority. In this case the popular MoSCoW (Must, Should, Could, Would) prioritization scheme has been used. The use cases should also be estimated. In this case a simple scheme of assessing their relative size and complexity has been used.

The essential properties for a use-case slice are 1) a meaningful name, 2) references to the use case and the flows it covers, 3) references to the tests and test cases that will be used to verify its completion, and 4) an estimate of the work needed to implement and test the slice. In this example the reference to the use case is implicit in the slices number and list of flows. The estimates have been added later after consultation with the team. These are the large numbers towards the bottom right of each sticky note. In this case the team has played planning poker to create relative estimates using story points; 5 story points for slices 7.1 and 7.2, and 13 story points for slice 7.3 which the team believe will take more than twice the effort of the other slices. Alternatively ideal days, t-shirt sizing (XS, S, M, L, XL, XXL, XXXL), or any other popular estimating technique could be used.

The use cases and the use-case slices should also be ordered so that the most important ones are addressed first. **FIGURE 19** shows how these sticky notes can be used to build a simple product back log on a white board. Reading from left to right you can see 1) the big picture illustrated by use-case diagrams showing the scope of the complete system and the first release, 2) the use cases selected for the first release and some of their slices which have been identified but not yet detailed and ordered, 3) the ordered list of slices ready to be developed in the release and finally 4) those slices that the team have successfully implemented and verified.



FIGURE 19: USING USE CASES AND USE-CASE SLICES TO BUILD A PRODUCT BACKLOG

FIGURE 19 is included for illustrative purposes only, there are many other ways to organize and work with your requirements. For example, many teams worry about their sticky notes falling off the whiteboard. These teams often track the state of their use cases and use-case slices using a simple spreadsheet including worksheets such as those shown in **FIGURE 20** and **FIGURE 21**.

USE CASE	NAME	RELEASE	PRIORITY	STATE	SIZE	COMPLEXITY
7	BROWSE AND SHOP	1	1 - MUST	STORY STRUCTURE UNDERSTOOD	VERY LARGE	HIGH
14	GET NEW AND SPECIAL OFFERS	1	1 - MUST	STORY STRUCTURE UNDERSTOOD	MEDIUM	MEDIUM
17	MAINTAIN PRODUCTS & AVAILABILITY	1	1 - MUST	STORY STRUCTURE UNDERSTOOD	LARGE	HIGH
12	TRACK ORDERS	1	2 - SHOULD	STORY STRUCTURE UNDERSTOOD	LARGE	LOW
13	LOCATE STORE	1	2 - SHOULD	STORY STRUCTURE UNDERSTOOD	SMALL	LOW
16	SET PRODUCT OFFERS	1	2 - SHOULD	STORY STRUCTURE UNDERSTOOD	MEDIUM	HIGH
11	GET SHOPPING HISTORY	1	3 - COULD	STORY STRUCTURE UNDERSTOOD	SMALL	MEDIUM
1	BUILD HOUSE			GOAL ESTABLISHED		
2	DESIGN INTERIOR			GOAL ESTABLISHED		
3	BUILD GARDEN			GOAL ESTABLISHED		
4	FILL GARDEN			GOAL ESTABLISHED		

FIGURE 20: THE USE-CASE WORKSHEET FROM A SIMPLE USE-CASE TRACKER

USE CASE	SLICE	STORIES	FLOW	TEST CASES	STATE	ORDER	ESTIMATE (STORY POINTS)	TARGET RELEASE
7 BROWSE AND SHOP	7.1	SELECT AND BUY 1 PRODUCT	BF	7.1.1	PREPARED	1	13	1
7 BROWSE AND SHOP	7.2	SELECT AND BUY MANY PRODUCTS	BF	7.2.1, 7.2.3	PREPARED	2	13	1
7 BROWSE AND SHOP	7.4	HANDLE PAYMENT AND DELIVERY DETAILS	A4, A5, A6	7.3.1, 7.3.2	SCOPED	3	3	1
17 MAINTAIN PRODUCTS AND AVAILABILITY	17.1	CREATE NEW PRODUCT	BF	17.1.1, 17.1.2	SCOPED	4	20	1
1 BUILD HOUSE	1.1	ADD FIRST HOUSE	BF	1.1.1, 1.1.2, 1.1.3	SCOPED	5	5	1
2 DESIGN INTERIOR	2.1	DESIGN ROOM	BF, A3		VERIFIED	6	5	1
2 DESIGN INTERIOR	2.4	PURCHASE CONTENTS	A6		SCOPED	7	3	1
7 BROWSE AND SHOP	7.5	HANDLE LOSS OF KEY SUPPORT SYSTEMS	A9, A11, A12		SCOPED	8	5	1
7 BROWSE AND SHOP	7.6	PRODUCT OUT OF STOCK	A7		SCOPED	9	13	1
17 MAINTAIN PRODUCTS AND AVAILABILITY	17.2	HANDLE PRODUCT DETAIL ERRORS	A2, A3		SCOPED	10	5	1
7 BROWSE AND SHOP	7.7	QUIT SHOPPING	A14		SCOPED	11	20	1
5 WALK-THROUGH DESIGN	5.1	NAVIGATE DESIGN	BF, A1		SCOPED	12	8	1
5 WALK-THROUGH DESIGN	5.2	HANDLE NAVIGATION ERRORS	A2		SCOPED	13	2	1
1 BUILD HOUSE	1.2	ADD OR REMOVE FLOOR	A2, A5	1.2.1, 1.2.2, 1.2.3	SCOPED	14	1	1

FIGURE 21: THE USE-CASE SLICE WORKSHEET FROM A SIMPLE USE CASE TRACKER

These illustrations are included to help you get started. The use cases and the use-case slices are central to everything the team does, so use whatever techniques you need to make them tangible and visible to the team. Feel free to add other attributes as and when you need them, for example to record the source of the use case or its owner, or to target the use-case slices on a particular increment within the release.

Completing the Work Products

As well as tracking the use cases and the use-case slices you will need to, at least, sketch out and share the supporting work products.

All the work products are defined with a number of levels of detail. The most important level of detail defines the essentials, the minimal amount of information that is required for the practice to work. Further levels of detail help the team cope with any special circumstances they might encounter. For example, this allows small, collaborative teams to have very lightweight use-case narratives defined on simple index cards and large distributed teams to have more detailed use-case narratives presented as documents. The teams can then grow the narratives as needed to help with communication, or thoroughly define the important or safety critical requirements. It is up to the team to decide whether they need to go beyond the essentials, adding detail in a natural fashion as they encounter problems that the bare essentials cannot cope with.

In some cases there will be an additional level of detail that is used to establish a sketch of the work product prior to the completion of the essentials. For example a use case narrative may be briefly described when the use case is found and then further evolved to create the bulleted outline once it is decided that the use case is in scope.

FIGURE 22 shows the levels of detail defined for the set of Use-Case 3.0 work products. The lightest level of detail is shown at the top of the table. The amount of detail in the work product increases as you go down the columns enhancing and expanding the content.

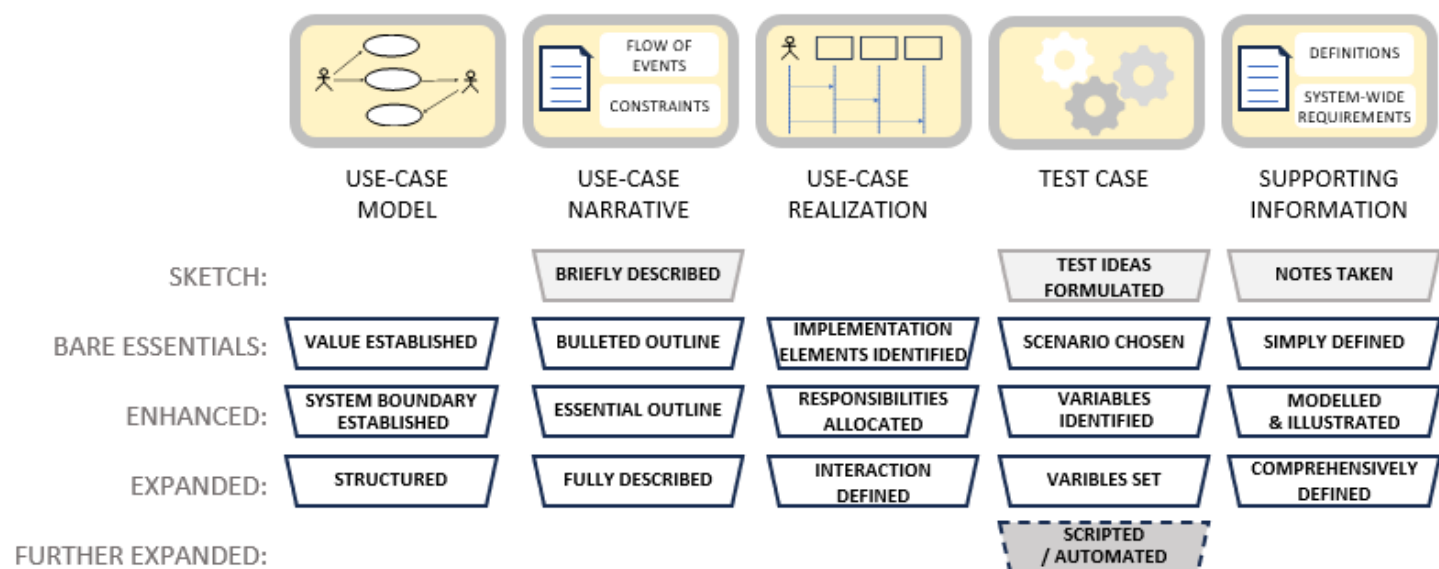


FIGURE 22: WORK PRODUCT LEVELS OF DETAIL

The good news is that you always start in the same way, with a sketch that evolves cover the essentials. The team can then continually adapt the level of detail in their use-case narratives to meet their emerging needs. The level of detail can also be adjusted to reduce waste; anything beyond the essentials should have a clear reason for existing or be eliminated. As Einstein (is attributed to have) said “Everything should be made as simple as possible, but not simpler”.

For more information on the work products and their levels of detail see Appendix 1: Work Products.

Things to do

Use-Case 3.0 breaks the work up into a number of essential activities that need to be done if the use cases are to provide real value to the team. These activities are shown in [FIGURE 23](#) where they are grouped into those activities used to discover, order and verify the requirements, and those used to shape, implement and test the system. The solid chevrons indicate activities that are explicitly defined by Use-Case 3.0. The dashed chevrons are placeholders for other activities that the practice depends on to be successful. Use-Case 3.0 does not care how these activities are performed, it just needs them to be done.

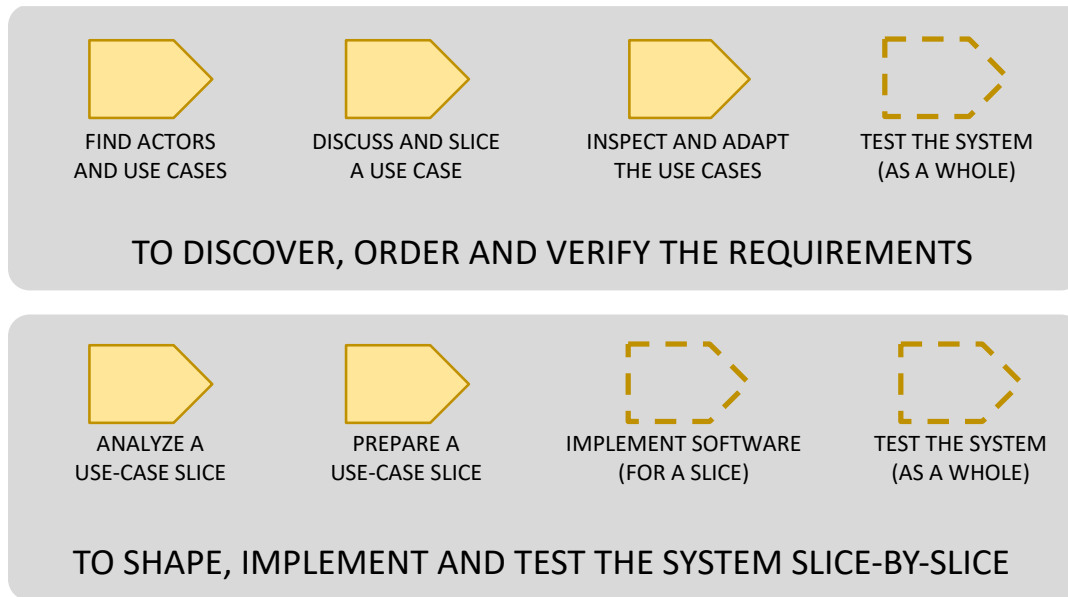


FIGURE 23: THE ACTIVITIES IN USE-CASE 3.0

Read [FIGURE 23](#) from left to right to get an impression of the order in which the activities are first performed. The activities themselves will all be performed many times in the course of your work. Even a simple activity such as ‘Find Actors and Use Cases’ may need to be performed many times to find all the use cases, and may be conducted in parallel with, or after, the other activities. For example, whilst continuing to ‘Find Actors and Use Cases’ you may also be implementing some of the slices from those use cases found earlier.

The rest of this chapter introduces each of the activities, following the journey of a use-case slice from the initial identification of its parent use case through to its final testing and inspection. The next chapter includes a brief discussion on how to organize the activities to support different development approaches such as Scrum, Kanban, Iterative and Waterfall.

Find Actors and Use Cases

First you need to find some actors and use cases to help you to:

- Agree on the goals of the system.
- Agree on new system behavior.
- Scope releases of the system.
- Agree on the value the system provides.
- Identify ways of using and testing the system.

The best way to do this is to hold a use-case modeling workshop with your stakeholders. There is no need to find all the system's use cases, just focus on those that are going to provide the stakeholders with the value they are looking for. Other actors and use cases will be found as you inspect and adapt the use cases.

As the use cases are discovered they should be ordered to support the team's release plans. One of the great things about use cases is that they enable high-level scope management without the need to discover or author all the use-cases' flows or slices. If a use case isn't needed, then there is no need to discuss or document its flows or slices. If the use case is in scope, it should be outlined so that there is enough information to start the slicing process.

Repeat this activity as necessary to evolve your model and find any missing actors or use cases.

TIP: MODEL STORM TO KICK START YOUR USE-CASE MODEL

The formal nature of the use-case model, and its use of the Unified Modeling Language, can be a barrier to involving stakeholders in the modelling effort.

A great way to overcome this is to simply get the stakeholders together to brainstorm some different users and their goals using sticky-notes (vertical for users and horizontal for goals.) Then facilitate the grouping of these into actors and use cases, which the stakeholders will then find very easy to quantify, outline, and order.

Discuss and Slice a Use Case

Next you need to build your understanding of the use case and create your first use-case slices. You do this to:

- Align on the scope of the use case and its flow of events.
- Create suitably sized items for the team to work on.
- Fit within the time and budget available.
- Deliver the highest value to the users, and other stakeholders.
- Demonstrate critical project progress or understanding of needs.

Even the simplest use case will cover many flows and many paths to value. You need to slice the use case to select the flows to be implemented. You should do the slicing with your stakeholders to make sure that all the slices created are of value and worth implementing. Don't slice up all the use cases at once. Just focus on the use cases to be worked in immediately.

You don't need to completely slice up the use case, just pull out the slices that are needed to progress the work and leave the rest of the use case for slicing when and if it is needed. You can even adopt a pull model where the developers ask for new slices to be identified as and when they have the capacity to implement them.

The slices created should be sequenced for delivery to make sure the development team tackles them in the right order. Again, you should do this with your stakeholders and other team members to make sure that the ordering defines the smallest usable system possible. The best way to do this is to consider the combination of priority, value, risk, and necessity.

Repeat this activity whenever new slices are needed.

TIP: YOU ONLY NEED THE BASIC FLOW OF THE MOST IMPORTANT USE CASE TO CREATE YOUR FIRST SLICE

Many people think that they need to have outlined all the use cases before they can start to create their slices. This leads to the adoption of a waterfall approach that delays the creation of valuable, working software.

One slice from one use case is enough to get the team started on the development and testing of the system.

The first slice of a use case should always be based on its basic flow. For some complex systems this slice may not even cover the whole of the flow. You may just take a subset of the basic flow, skipping the detail of some steps and stubbing up the solution for others, to attack the biggest challenges in developing the use case and learn if it can be implemented successfully.

Analyze a Use-Case Slice

Before you commit to doing the work and start coding you should analyze the slice to:

- Understand its impact on the system elements that will be used to implement it.
- Define the responsibilities of the affected system elements.
- Define how the system elements interact to perform the use case.

When a team is presented with a new slice to implement the first thing to do is to work out how it will affect the system. How many bits of the system are going to need to be changed and in what way? How many new things are needed and where do they fit?

Analyzing the target slices is often a pre-cursor to the planning of the development tasks. It allows the team to plan the implementation of the slice as a set of smaller code changes rather than as one, large, indivisible piece of work. Alternatively, the slice itself can be used as the unit of work and analyzing the slice is just the final thing to be undertaken by the developer before the coding starts.

As the team build their understanding of the system and its architecture, they will find it easier and easier to analyze the slices and will often be able to do it in their heads. It is still worth sketching the analysis out with some colleagues before committing to the coding. This will validate a lot of the design decisions, check that nothing has been misunderstood, and provide traceability for use later when investigating the impact of defects and changes. The result of this kind of analysis is known as a use-case realization as it shows how the use case is realized by the elements of the implementing system.

Perform this activity at least once for each slice. Repeat this activity whenever there are changes applied to the slice.

TIP: KEEP THE ANALYSIS COLLABORATIVE AND LIGHTWEIGHT

The easiest way to analyze the use-case slice is to get the team together to discuss how it would affect the various elements of the system.

As the team walks-through the design they picture it on a white board, typically in the form of a simple sequence or collaboration diagram, which can then be captured as a photograph or in the team's chosen modelling tool.

Prepare a Use-Case Slice

Once a slice is selected for development more work is required to:

- Get it ready for implementation.
- Clearly define what it means to successfully implement the slice.
- Define required characteristics (i.e. non-functional requirements).
- Focus the development of software towards the tests it must meet.
- Split it up into a set of smaller work items (for example, user stories, features, or tasks) that can be integrated into the team's backlog and plans.

Preparing a use-case slice is a collaborative activity, you need input from the stakeholders, developers, and testers to clarify the use-case narrative and define the test cases. When preparing a use-case slice you should focus on the needs of the developers and testers who will implement and verify the slice.

Think about how they will access the information they need. Will they be able to talk to subject matter experts to flesh out the requirements, or will they need everything to be documented for them? You also need to balance the work between the detailing of the use-case narrative and the detailing of the test cases. The more detail you put in the use-case narrative the easier it will be to create the test cases. On the other hand the lighter the use-case narrative the less duplication and repetition there will be between it and the test cases. You should create the use-case narrative and the test cases at the same time, so that the authors can balance their own needs, and those of their stakeholders.

Think about how to split the work up into a set of smaller work items (for example, User Stories, Features, or Tasks) that can be integrated into the team's backlog and plans. For example, the slice will probably touch on many of the system's components or span many steps. In both cases it may be possible for each of these changes to be done independently or in parallel. The end-to-end testing of the slice will make sure that everything is pulled together and integrated before the slice is considered done.

Perform this activity at least once for each slice. Repeat this activity whenever there are changes applied to the slice.

TIP: IF THE SLICE HAS NO TEST CASES, THEN IT HAS NOT BEEN PROPERLY PREPARED

When you prepare a use-case slice do not forget to define the test cases that will be used to verify it. It is by looking at the test cases that we know what really needs to be achieved.

The test cases provide the developers with an empirical statement of what the system needs to do. They will know that the development of the slice is not completed until the system successfully passes all the test cases.

TIP: FOR AGILE TEAMS SPLIT THE SLICE UP INTO A SET OF SMALL, ACTIONABLE USER STORIES

Use-Case Slices generally involve many steps and changes to many different parts of the system because of this they usually require more than a few days effort to complete. This makes them too large to meet the definition of ready used by many Agile teams particularly those used to using User Stories.

The good news is that Use-Case Slices can easily be split up into a set of User Stories - for example each step can be presented as a User Story. For example Step 1: As a customer I want to withdraw a standard amount of cash so that I can go shopping, Step 2: As a Bank I want to check that the funds are available so that the customer does not go overdrawn. The same logic can also be applied to each alternative flow; for example Alt 1: As a Bank I want to offer Customers in good standing an overdraft if they have insufficient funds to fulfill their requested withdrawal so that they are happy with our service.

The Use-Case Slice can then act as the team goal.

Implement Software (for a Slice)

You are now ready to design, code, unit test, and integrate the software components needed to implement a use-case slice.

The software will be developed slice-by-slice, with different team members working in parallel on different components or related work items. Each slice will require changes to one or more pieces of the system. To complete the implementation of a slice the resulting pieces of software will have to be unit tested and then integrated with the rest of the system.

Test the System (for a Slice)

Next, independently test the software to verify that the use-case slice has been implemented successfully. Each use-case slice needs to be tested before it can be considered complete and verified. This is done by successfully executing the slice's test cases. The independence of the use-case slices enables you to test it as soon as it is implemented and provide immediate feedback to the developers.

Use-case 3.0 works with most popular testing practices. It can be considered a form of test-driven development as it creates the test cases for each slice before the slice is given to the developers for implementation.

Test the System as a Whole

Each increment of the software system needs to be tested to verify that it correctly implements all the new use-case slices without breaking any other parts of the system. It is not sufficient to just test each slice as it is completed. The team must also test the system as a whole to make sure that all of the implemented slices are compatible, and that the changes to the system haven't resulted in the system failing to support any previously verified slices.

The test cases produced using Use-Case 3.0 are robust and resilient. This is because the structure of the use-case narratives results in independently executable, scenario-based test cases.

Inspect and Adapt the Use Cases

You also need to continuously tune and evaluate the use-case model, use cases, and use-case slices to:

- Handle changes.
- Track progress.
- Fit your work within the time and budget available.
- Keep the use-case model up to date.
- Tune the size of the slices to increase throughput.

As the work progresses it is essential that you continually evolve your use-case model, use cases and use-case slices. Priorities change, lessons are learnt, and changes are requested. These can all have an impact on the use cases and use-case slices that have already been implemented, as well as those still waiting to be progressed. This activity will often lead to the discovery of new use cases and the refactoring of the existing use cases and use-case slices.

The varying demands of the project may need you to tune your use of Use-Case 3.0 and adjust the size of the slices or the level of detail in your use-case narratives, supporting information and test cases. It is important that you continually inspect and adapt your way-of-working as well as the use cases and use-case slices you are working with.

Perform this activity as needed to maintain your use cases and handle changes.

TIP: DON'T FORGET TO MAINTAIN YOUR BACKLOG OF USE-CASE SLICES

By ordering your slices and tracking their state (defined, analyzed, prepared, implemented, verified) you create a backlog of the requirements left to implement. This list should be continually monitored and adjusted to reflect the progress of the team and the desires of the stakeholders.

As the work progresses you should monitor and adjust the slice size to eliminate waste and improve the team's effectiveness.

Practices

Introducing the Use-Case 3.0 Practice Family

Over the years use-case practitioners have found many ways to benefit from the use of use cases in many different domains and problem spaces.

This widespread adoption of use cases has led to many different use-cases for use cases themselves including:

- Focus on delivering end-to-end value by focusing on the most critical use cases.
- Visually summarize what a system does for its users.
- Provide a permanent record of what a system does.
- Formally drive the development of a system.

In the first two cases people have typically been using either the occasional use-case narratives or a stand-alone use-case models to provide context for, and help, identify User Stories.

To help facilitate the different ways that use cases can be used this e-book is now supported by a set of Essence practices. **FIGURE 24** shows the full set of practices. These provide you with a selection of lightweight starting points that can then be extended to add structure and formality as and when it is needed. They have been designed to support these common use cases and adapt to any others that may emerge.

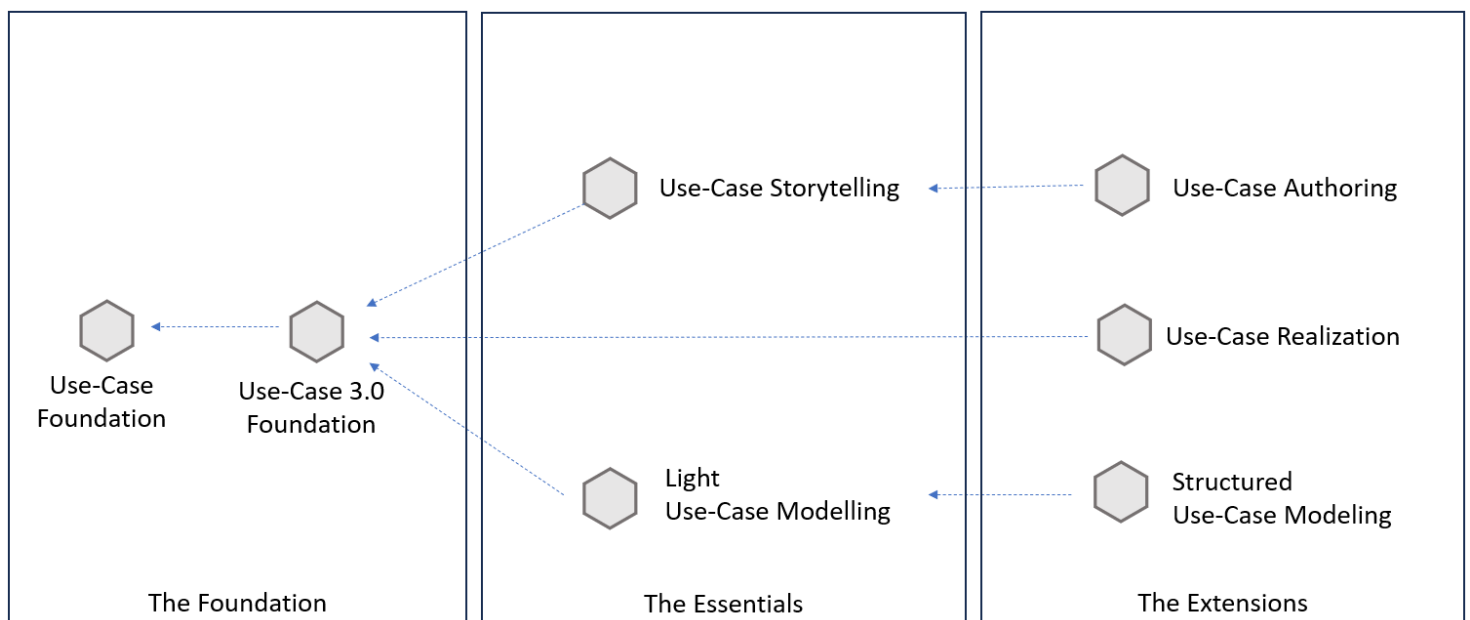


FIGURE 24: THE USE-CASE 3.0 PRACTICE FAMILY

With Use Case 3.0 you can start your use case journey from any combination of the two essential use-case practices:

- Use-Case Storytelling - the use of simple use-case narratives to help teams focus on delivering a usable system of clear value to its users.
- Light Use-Case Modelling - the creation of a use-case model that visualizes the value of a system and how it helps the users achieve their goals.
- Use-Case Essentials: A combination of Use-Case Storytelling and Light Use-Case Modeling. Combining the two essential use-case practices gives you both a use-case model and its supporting narratives to drive the development of a system and act as its permanent record.

You can then add additional content as you continue your use-case journey by selecting additional use case

3.0 practices. For example - if needed, you can apply the detailed Use-Case Authoring practice to provide more detail to your use-case narratives and supporting information. You can add Use-Case Realization to create design artefacts such as sequence and collaboration diagrams. You can add Structured Use-Case Modeling with supporting use cases and additional modeling conventions to allow you to cover the full scope of your intended system.

You can of course use Use-Case 3.0 in its entirety, as shown in [FIGURE 25](#) and discussed in the previous section, but as you can see that's quite a lot to take on in one go particularly if you have never used use cases before.

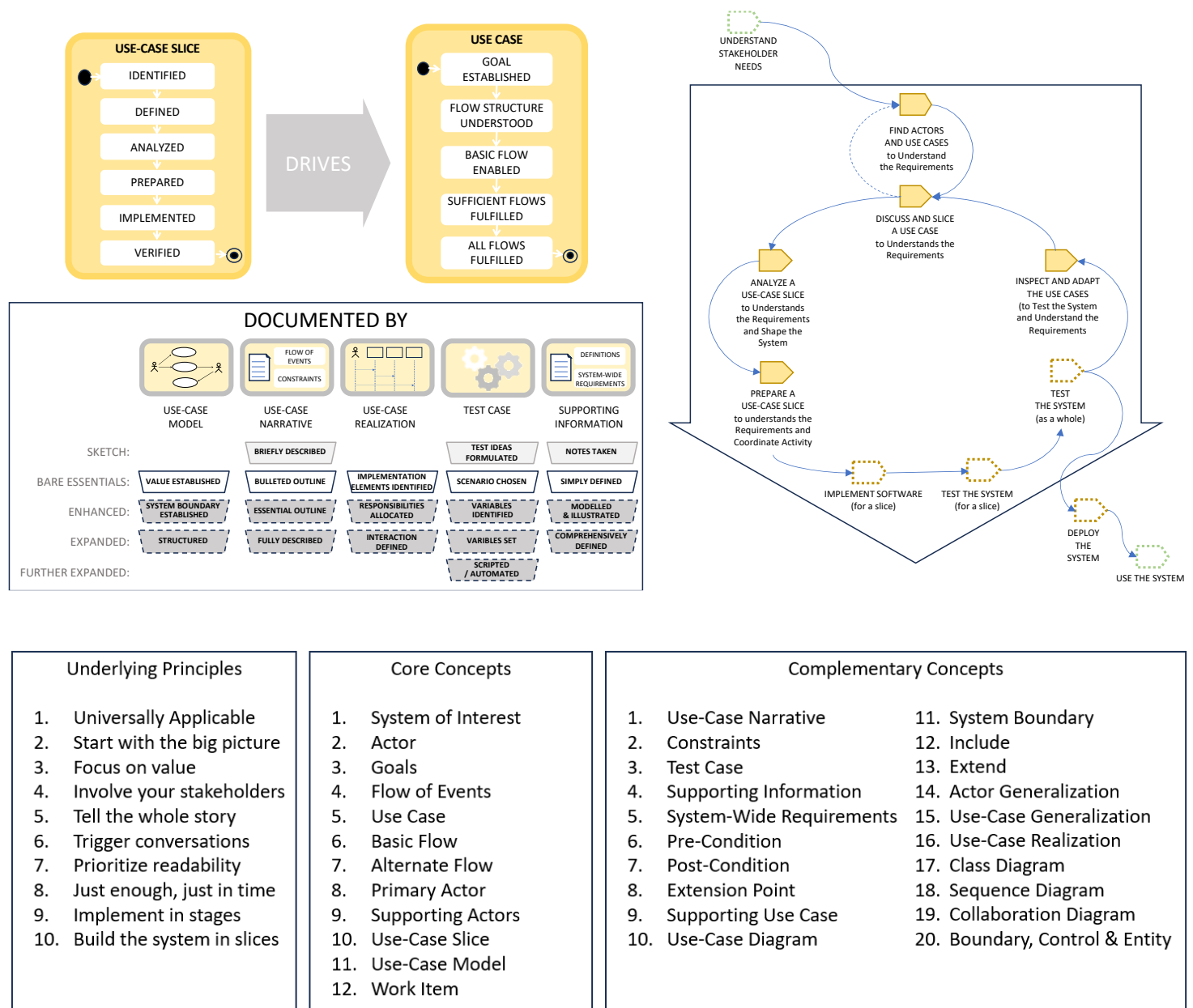


FIGURE 25: AN OVERVIEW OF USE-CASE 3.0

In the following sections we will look at each practice in turn and some of the more common combinations of these and other practices.

Use-Case Foundation

The foundational concepts and principles that underlie all successful applications of Use Cases.

Underlying Principles	Core Concepts
1. Universally Applicable	1. System of Interest
2. Start with the big picture	2. Actor
3. Focus on value	3. Goals
4. Involve your stakeholders	4. Flow of Events
5. Tell the whole story	5. Use Case
6. Trigger conversations	6. Basic Flow
7. Prioritize readability	7. Alternate Flow
8. Just enough, just in time	8. Primary Actor
9. Implement in stages	9. Supporting Actors

FIGURE 26: AN OVERVIEW OF THE USE-CASE FOUNDATION

This practice is based on the Use-Case Foundation published by Ivar Jacobson and Alistair Cockburn in 2024.

Use-Case 3.0 Foundation

The foundational concepts and principles that underlie all successful applications of Use Case 3.0. The Use-Case 3.0 Foundation extends the Use-Case Foundation to add the key Use-Case 3.0 concepts of the Use-Case Slice and to elaborate on the states of the Use Case Alpha whilst adding an additional principle ‘build the system in slices’.

It also binds the foundation to the Essence Kernel making it executable and adds the concept of Work Item to enable the composition of all the Use-Case 3.0 Practices with other popular practices such as Use Stories, Scrum, Kanban, integration with larger agile frameworks such as SAFe®, Scrum@Scale, Nexus and also integration with more traditional task-based planning approaches.

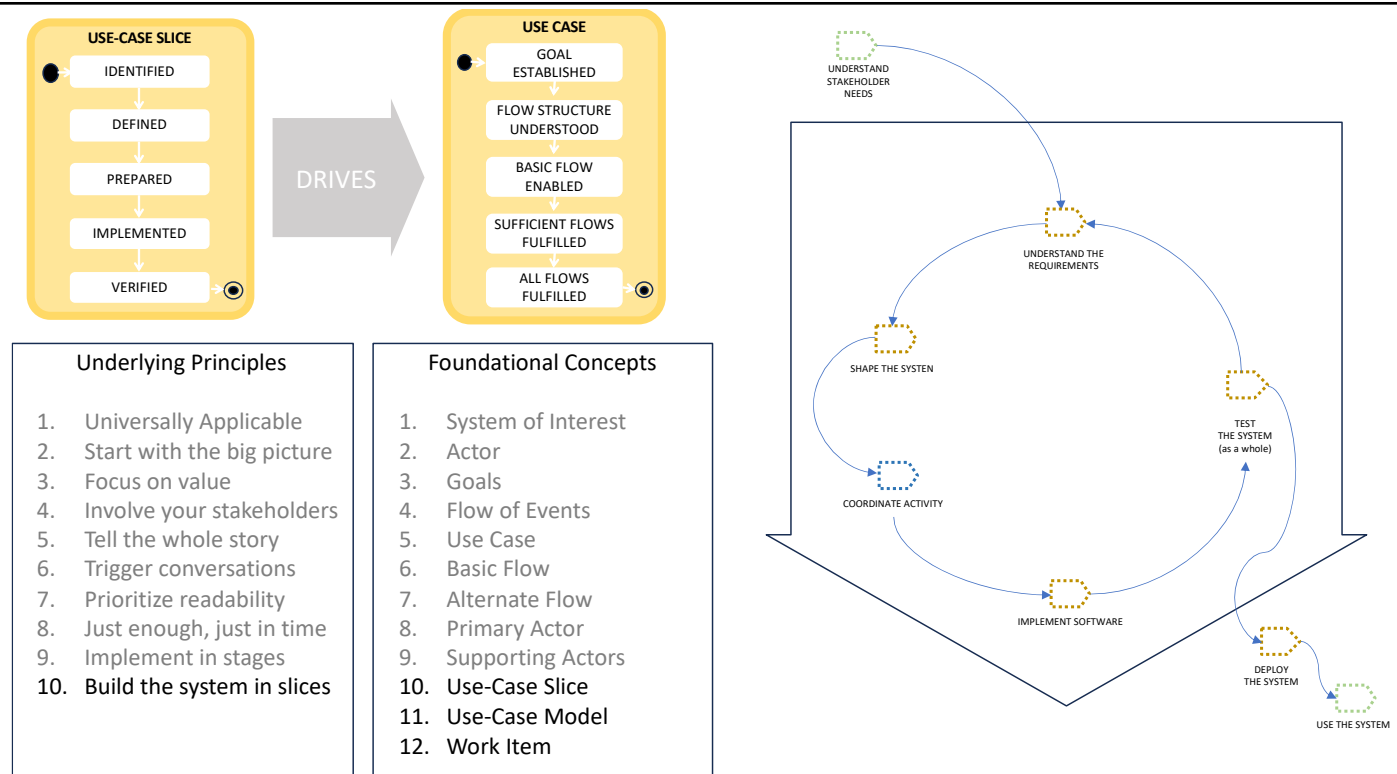


FIGURE 27: AN OVERVIEW OF THE USE-CASE 3.0 FOUNDATION

Use-Case Storytelling

Apply simple Use-Case Narratives to capture the most valuable aspects of your system and incrementally deliver a system that helps the users to achieve their real goals.

You can use this practice stand-alone when:

- You want to understand one or more critical aspects of your system.
- You need to create some valuable short-term goals for your team.
- You'd like to use use cases to add some structure and context to your User Story backlog.

Or in conjunction with the Light Use-Case Modeling practice:

- To flesh out your use cases

USE –CASE STORYTELLING

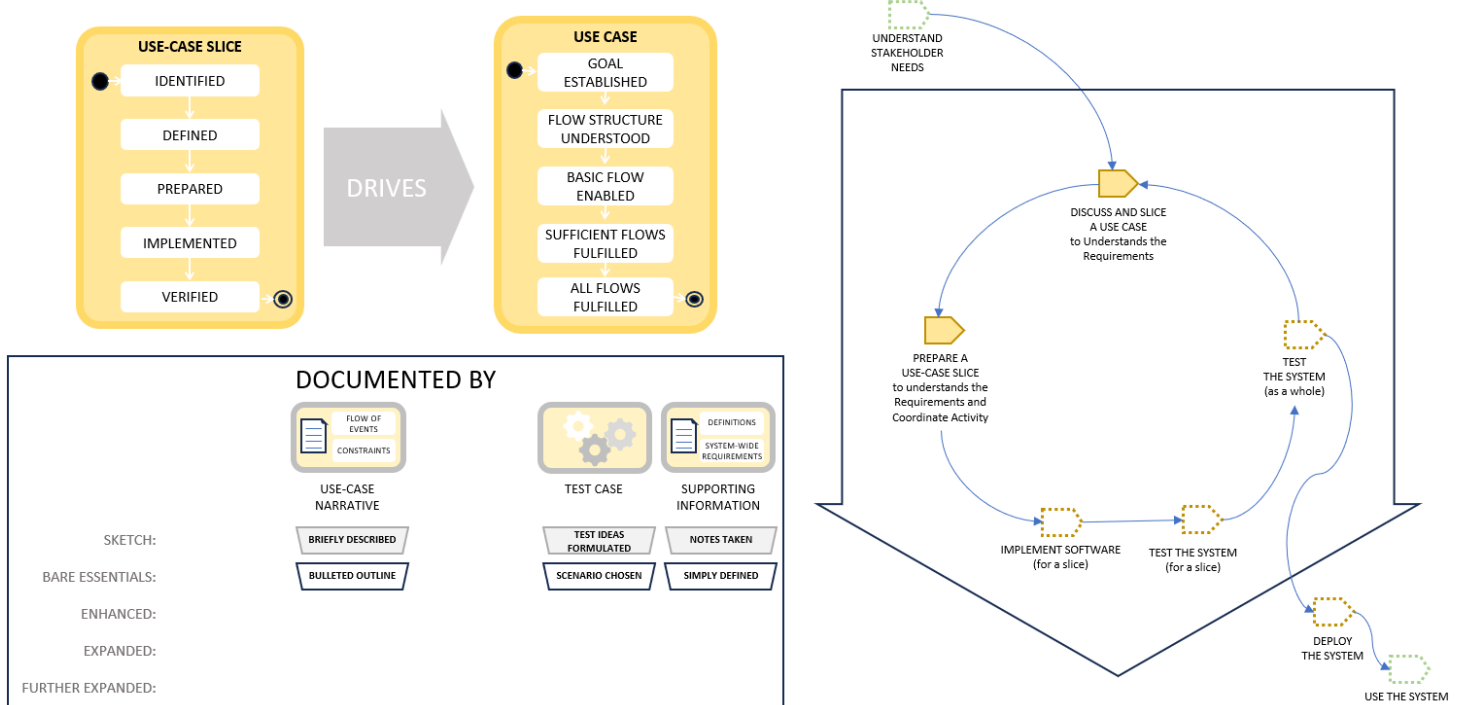


FIGURE 28: AN OVERVIEW OF THE USE-CASE STORYTELLING PRACTICE

Note: This practice does not include any use-case modeling. The team suggests a use case to focus on as part of the Discuss and Slice a Use Case activity.

This practice adds an additional concept that can be used to capture and document your Use-Cases: Constraints.

Once a development team has the agreed the flow of events with stakeholders, they can easily identify one or more user stories to implement the basic flow, identify additional user stories to implement each alternate flow and to handle exceptions. The development team can then discuss with stakeholders how to group those user stories together into coherent use-case slices - and so identify and define exactly what value should be tested and delivered by each release or by each increment.

Light Use-Case Modeling

Create a simple, visual model that identifies and focuses on the most valuable aspects of your system and succinctly communicates this value to all your stakeholders.

You can use this practice stand-alone when:

- You'd like to create a simple overview of the value your system provides.
- You'd like to have more context for your User Stories and Story Maps

Or in conjunction with the Use-Case Storytelling practice:

- To create use-case narratives and test cases that support your development and testing activities.

LIGHT USE-CASE MODELING

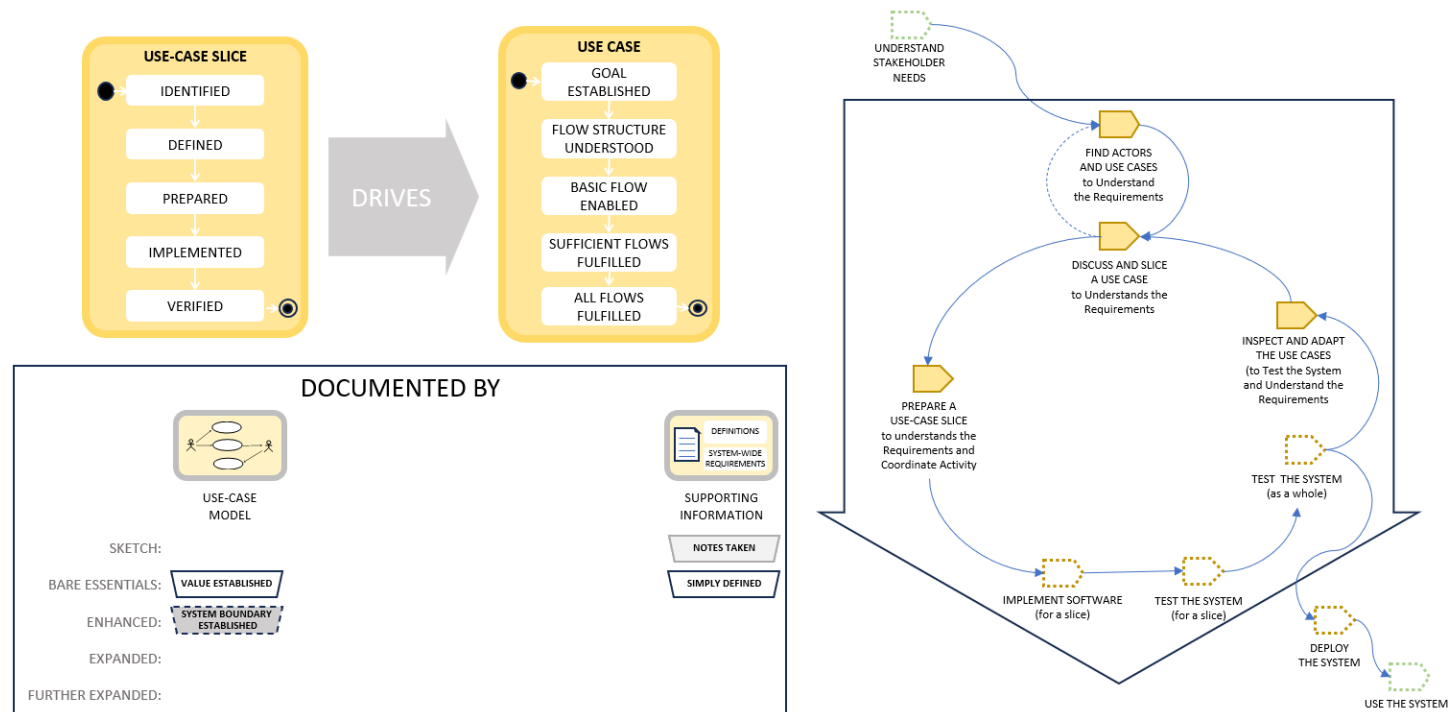


FIGURE 29: AN OVERVIEW OF LIGHT USE-CASE MODELING

This practice adds additional concepts that can be used to capture and document your Use-Case Model: Supporting Use Cases and Use-Case Diagrams.

A system boundary is established implicitly by using this lightweight practice, but the model may not yet capture a complete description of all actors and all use cases.

The light use-case modeling practice does not require the authoring of any formal use-case narratives; each use case can be used to provide more context for the identification of your User Stories or Features. A development team can brainstorm with their stakeholders what sequence of steps are required to fulfill the use case goal, as well as any failure conditions or alternate ways to achieve the same goal. These steps can then be used to identify a series of candidate user stories. The development team can then discuss with stakeholders how to group those user stories together into coherent use-case slices - and so identify and define exactly what value should be tested and delivered by each release or by each increment.

Use-Case Essentials

A combination of Use-Case Storytelling and Light Use-Case Modeling to create a lightweight Use-Case Essentials Practice.

Use this combination of practices when you want to place use-cases at the heart of your development processes.

USE-CASE ESSENTIALS

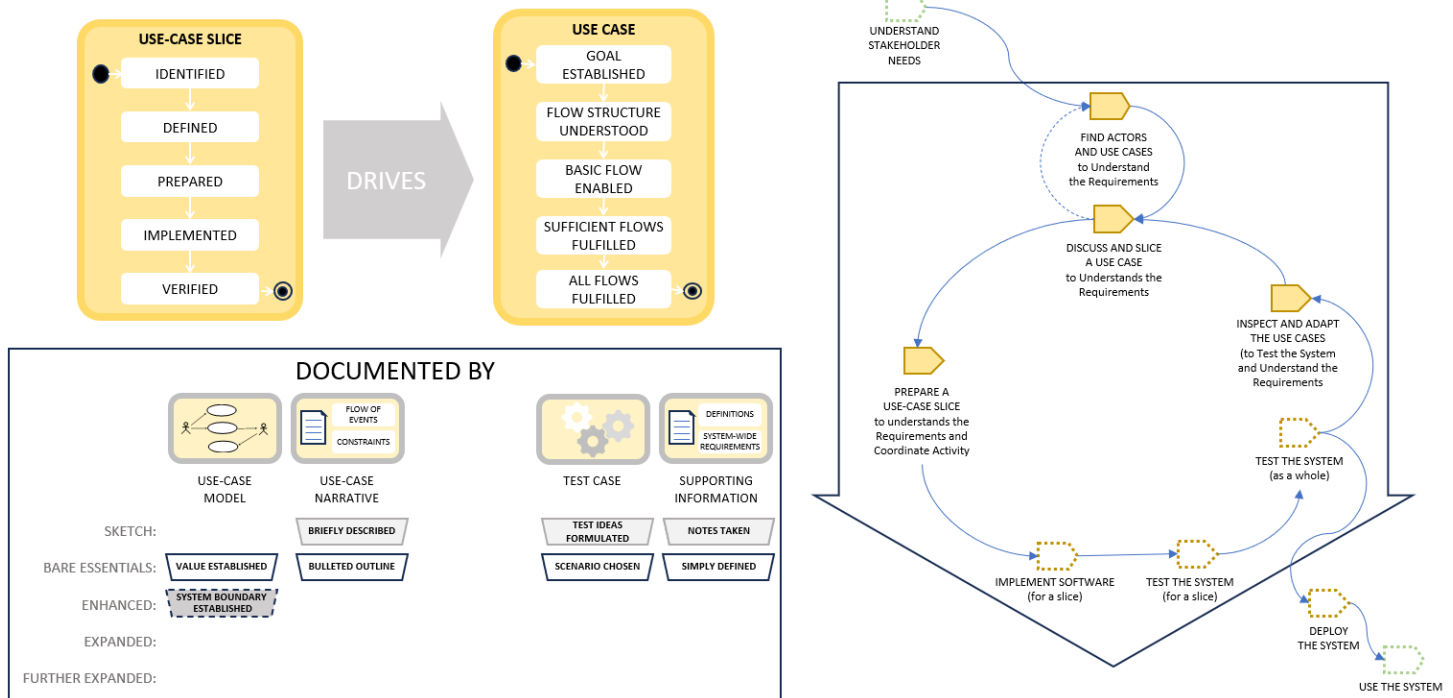


FIGURE 30: AN OVERVIEW OF THE USE-CASE DRIVEN ESSENTIALS PRACTICE

This is the most common starting point for people new to use cases as it combines the power of light use-case modeling to find your use cases with the efficiency of use-case storytelling to bring the use cases to life.

Use-Case Authoring

Evolve your Use-Case Narratives to provide a formal, detailed specification of the most critical aspects of your system.

This practice extends the Use-Case Storytelling practice to support the use of use cases when working with more complex business domains and safety critical systems, or when you need to support formal contracting mechanisms.

Use this practice when:

1. A complete, verifiable requirements specification is required for contractual or other reasons.
2. When, for whatever reason, the subject matter experts and the developers will not be able to have frequent, regular discussions with the developers about the use cases and their flows.
3. When an area of a system is complex and / or safety critical and needs a more rigorous up-front specification before development and testing can commence.

This practice can, of course, be used to add detail to your use-cases when starting with the Light Use-Case Essentials practice.

USE –CASE AUTHORING

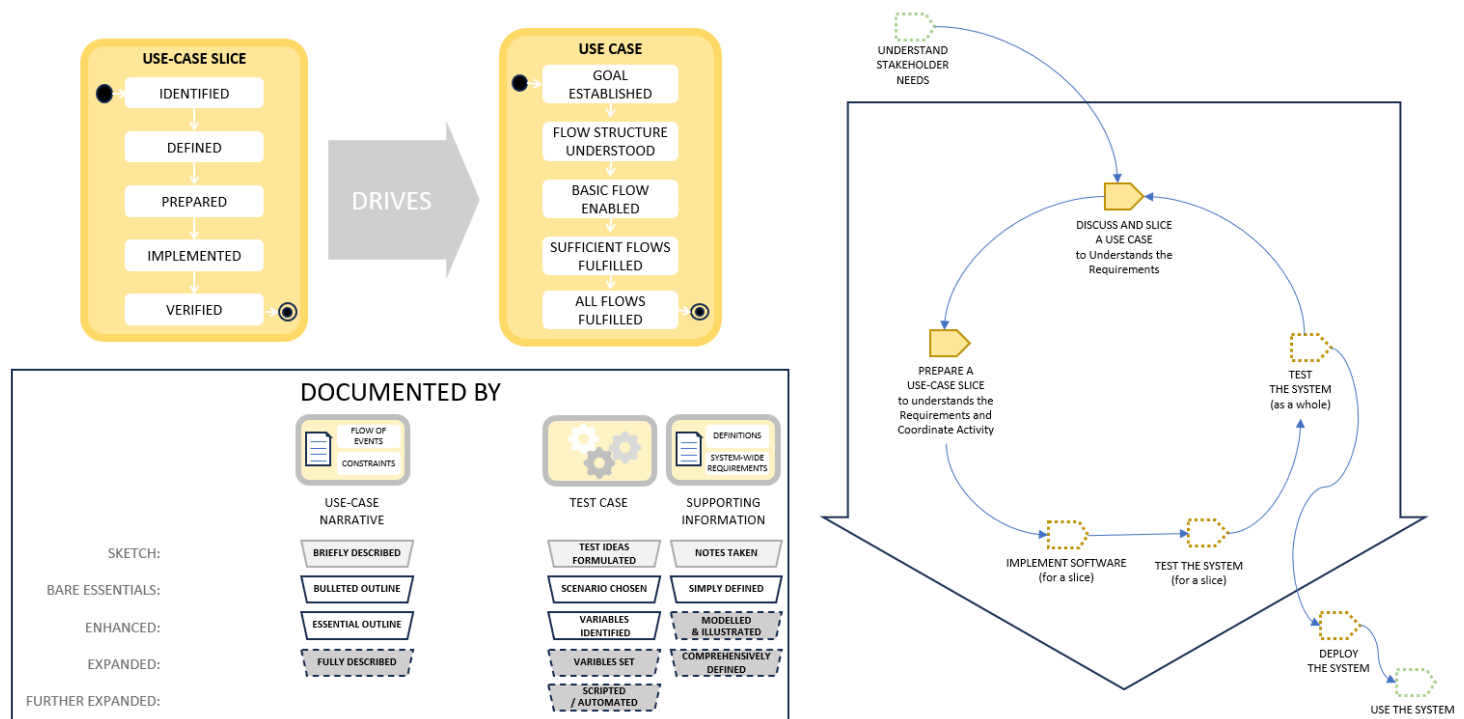


FIGURE 31: AN OVERVIEW OF THE USE-CASE AUTHORING PRACTICE

This practice adds additional concepts that can be used to add precision and accuracy to your Use-Case Narratives: Supporting Information, System-Wide Requirements, Pre-Conditions, Post-Conditions, and Extension Points

Once again, this practice does not require the creation and or maintenance of a Use-Case Model but, particularly where contracts are involved, it is usually combined with some form of Use-Case Modeling (light or structured) to provide a more complete specification of a system's requirements.

Structured Use-Case Modeling

Evolve your Use-Case Model to fully define the scope of your intended system and reflect its evolution over time.

This practice extends the Light Use-Case Modeling practice.

Use this practice when:

- You want to capture the full system boundary (i.e. all use cases and all actors are captured).
- You want to be able to better visualize the contents of your releases.
- You want to use your use-case model as the heart of the system's permanent record.
- You want to define different configurations and pricing options.

This practice can, of course, be used to add more rigor and completeness to your Use-Case Model when starting with the lightweight Use-Case Essentials practice.

STRUCTURED USE-CASE MODELING

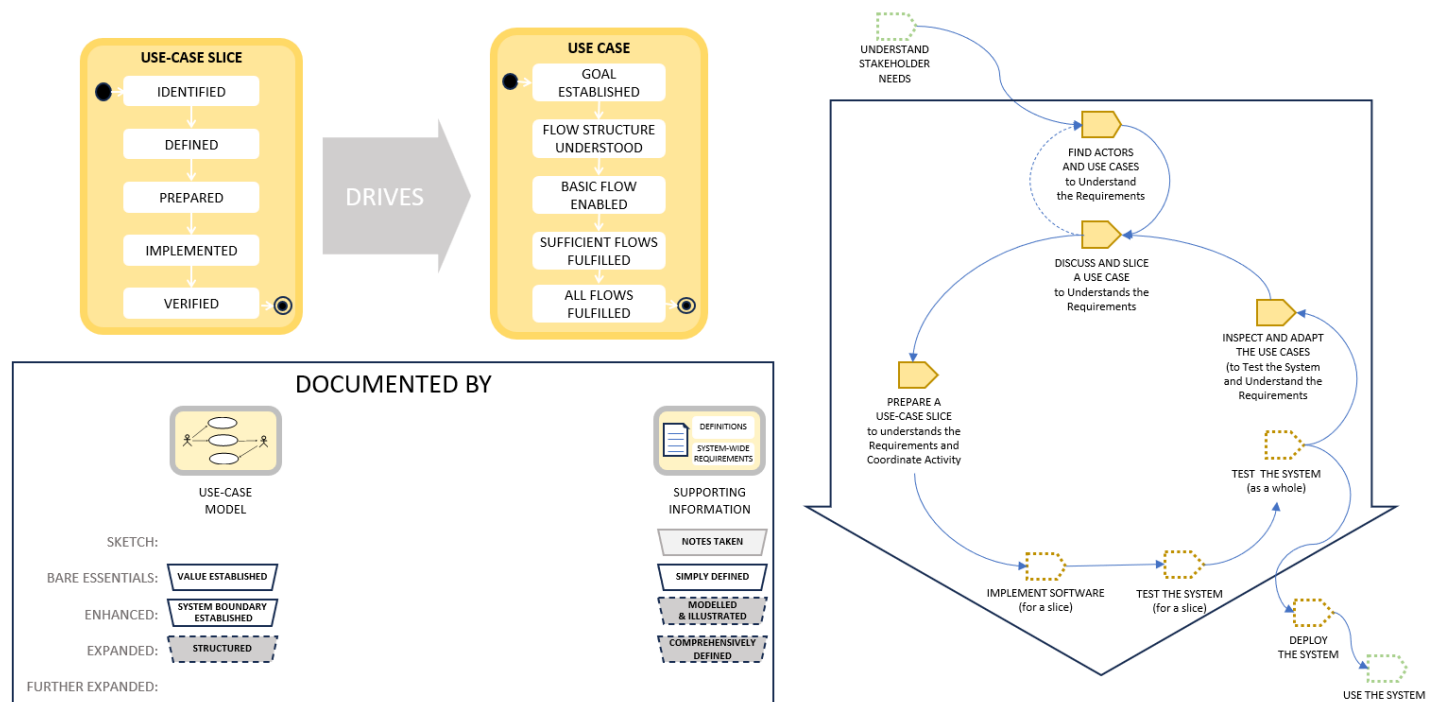


FIGURE 32: AN OVERVIEW OF THE STRUCTURED USE-CASE MODELING PRACTICE

This practice adds additional concepts that can be used to add flexibility, precision, and accuracy to your Use-Case Model: System Boundary, Include, Extend, Actor Generalization and Use-Case Generalization. Extend may be used to describe different behaviors of the system - for example to represent different options offered to customers, the scope of different releases, or different versions of the same system offered to customers at different price points. Extend may also be used to represent complex error handling or exception handling - where you do not wish for that behavior to obscure the understandability of the use case model. These additional concepts are extremely valuable, but they should only be used when they will help stakeholders understand the big picture, and where they improve understanding and communication.

Use-Case Realization

Analyze the impact that a use case and / or use-case slice has on the design and implementation of your system using the Unified Modeling Language (UML).

Use this practice when:

- You want to understand the subsystems and components impacted by a use case or use-case slice.
- You want to understand the architectural impact of a use case.
- You want to develop the different components of the system in parallel.

The practice can be used in conjunction with the Use-Case Storytelling Practice, the Light Use-Case Modeling Practice and any form of use-case driven development. Add this practice to analyze a use-case slice to produce a use-case realization work item.

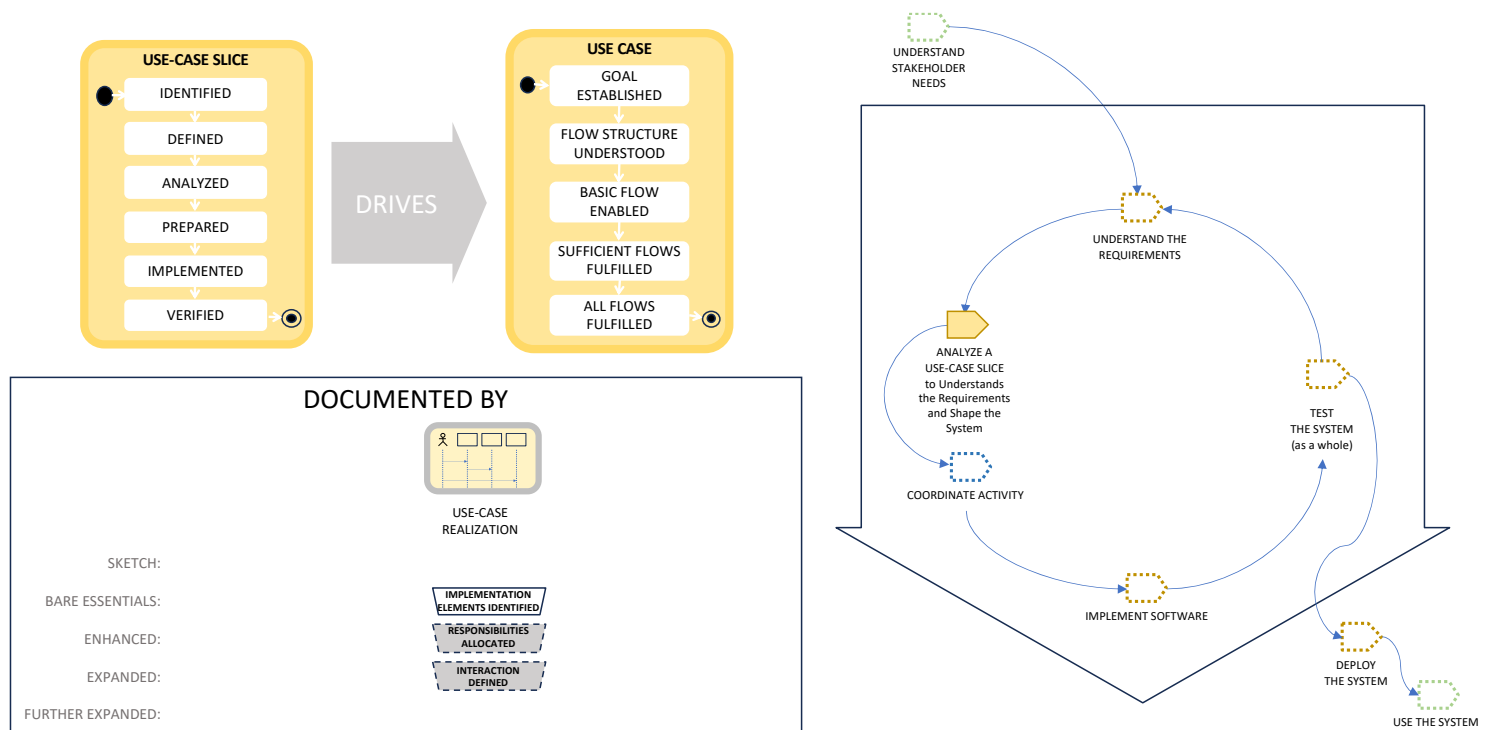


FIGURE 33: AN OVERVIEW OF THE USE-CASE REALIZATION PRACTICE

This practice adds additional concepts to help you realize your use cases: Use-Case Realization, Class Diagram, Sequence Diagram, Collaboration Diagram and Boundary, Control and Entity classes.

Note: There are many different approaches to realizing your use cases that you can use in conjunction with this practice such as UI Mockups, Wire Frames, textual scenarios and many, many more. So start with this practice and experiment with your preferred analysis and design techniques to get the maximum benefit from your use cases.

Using Use-Case 3.0

Use-Case 3.0 can be used in many different contexts to help produce many different kinds of system. In this chapter we look at using use cases for different kinds of system, different kinds of requirements and different development lifecycles.

It can also be used to complement and improve the results that team's get from using complementary product development, requirements, and work management practices such as User Stories, Features, and Task-based planning.

Use-Case 3.0: Applicable for all types of system

Many people think that use cases are only applicable to user-intensive systems where there is a lot of interaction between the human users and the system. This is strange because the original idea for use cases came from telecom switching systems, which have both human users (subscribers, operators) and machine users, in the form of other interconnected systems. Use cases are of course applicable for all systems that are used - and that means of course all systems.

Use-Case 3.0: It's not just for user-intensive applications

Use cases are just as useful for embedded systems with little or no human interaction as they are for user intensive ones. Nowadays, people are using use cases in the development of all kinds of embedded software in domains as diverse as the motor, consumer electronics, military, aerospace, and medical industries. Even real-time process control systems used for chemical plants can be described by use cases where each use case focuses on a specific part of the plant's process behavior and automation needs.

All that is needed for use cases to be appropriate is for the system to collaborate with the outside world, regardless of whether the users are humans or other systems. Their applicability is far broader than most people think.

Use-Case 3.0: It's not just for software development

The application of use cases is not limited to software development. They can also help you to understand your business requirements, analyze your existing business, design new and better business processes, and exploit the power of IT to transform your business. By using use cases recursively to 1) model the business and its interactions with the outside world and 2) model the systems needed to support and improve the business you can seamlessly identify where the systems will impact on the business and which systems you need to support the business.

The use cases used to model the business are often referred to as business use cases. They provide the context for your IT systems development work, allowing the business development and the IT development to be carried out in perfect synchronization. Not only can you develop the IT systems slice-by-slice, but you can also develop your business model slice-by-slice. This is very powerful as it allows you to evolve your business and its supporting systems in tandem with one another, enabling incremental business development as well as incremental systems development.

In the modern world the business and the IT systems that support it can, and should, be developed in synch (one won't work without the other). The use of use cases and use-case slices at both the business and IT boundaries can close the gap between the business and the IT enabling them to work as truly collaborative partners.

Use-Case 3.0: handling all types of requirement

Although they are one of the most popular techniques for describing a systems' functionality, use cases are also used to explore their non-functional characteristics. The simplest way of doing this is to capture them as part of the use cases themselves. For example, relate performance requirements to the time taken between specific steps of a use case or list the expected service levels for a use case as part of the use case itself.

Some non-functional characteristics are more subtle than this and apply to many, if not all, of the use cases. This is particularly true when building layered architectures including infrastructure components such as security, transaction management, messaging services, and data management. The requirements in these areas can still be expressed as use cases - separate use cases focused on the technical usage of the system. We call these additional use cases infrastructure use cases as the requirements they contain will drive the creation of the infrastructure that the application will run on. These use cases and their slices can be considered as cross-cutting concerns that will affect the behavior of the system when the more traditional functional use cases are performed. For example, a use case could be created to explore how the system will manage database transactions including all the different usage scenarios such as the schemes for data locking, data caching, commit and roll-back. This use case would apply every time another use case retrieves or stores data in the system.

Combining these infrastructure use cases with other techniques such as separation of concerns and aspect-oriented programming allows these common requirements to be addressed without having to change the implementation of the existing functional use cases.

Use-Case 3.0: Applicable for all development lifecycles

Use-Case 3.0 works with all popular software development lifecycles including:

- Iterative, backlog-driven approaches such as Scrum, EssUP and openUP and their scaled equivalents such as SAFe, Scrum@Scale, LeSS and Nexus.
- One-piece flow-based lifecycles such as Kanban
- All-in-one go lifecycles such as Waterfall

In the following 3 short sections we will illustrate how Use-Case 3.0 and, in particular, use-case slices can help with each of these. These sections are not as self-contained as the rest of the document and rely upon the reader having a basic understanding of the approach being discussed. We recommend that you only read the sections for the approaches you are familiar with.

Use-Case 3.0 and backlog-driven iterations

Before adopting any backlog-driven approach you must understand what items will go in the backlog. There are various forms of backlog that teams use to drive their work including product backlogs, release backlogs, team backlogs and project backlogs. Regardless of the terminology used they all follow the same principles. The backlog itself is an ordered list of everything that might be needed and is the single source of requirements for any changes to be made. The basic concept of a backlog is illustrated by FIGURE 34.

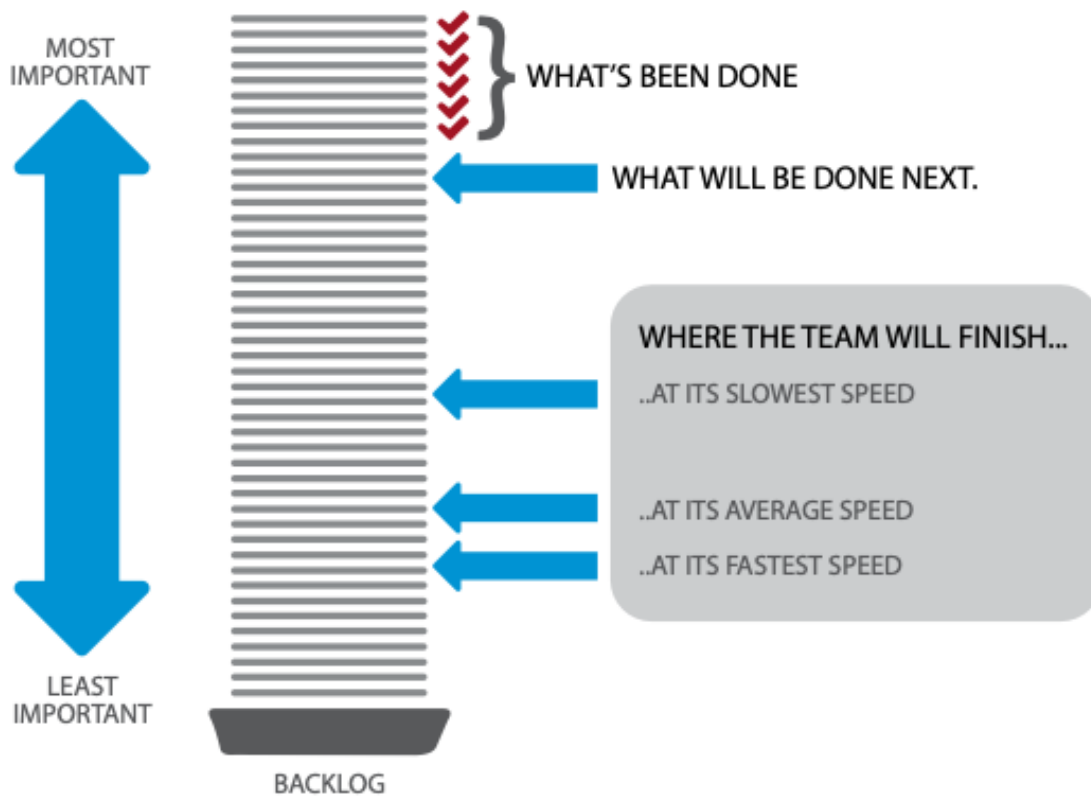


FIGURE 34: A BASIC BACKLOG

When you use Use-Case 3.0 the use-case slices can be used as the team's the primary backlog item type. The use of use-case slices ensures that your backlog items are well-formed, as they are naturally independent, valuable, and testable. The structuring of the use-case narrative that defines them makes sure that they are estimable and negotiable, and the use-case slicing mechanism enables you to slice them as small as you need to support your development team.

The use cases are not put into the ordered list themselves as it is not clear what this would mean. Does it mean that this is where the first slice from the use case would appear or where the last slice from the use case would appear? If you want to place a use case into the list before slicing just create a dummy slice to represent the whole use case and insert it into the list.

When you adopt a backlog-driven approach it is important to realize that the backlog is not built and completed up-front but is continually worked on and refined, something that is often referred to as refining or maintaining the backlog. The typical sequence of activities for a backlog-driven, iterative approach is shown in [FIGURE 35](#).

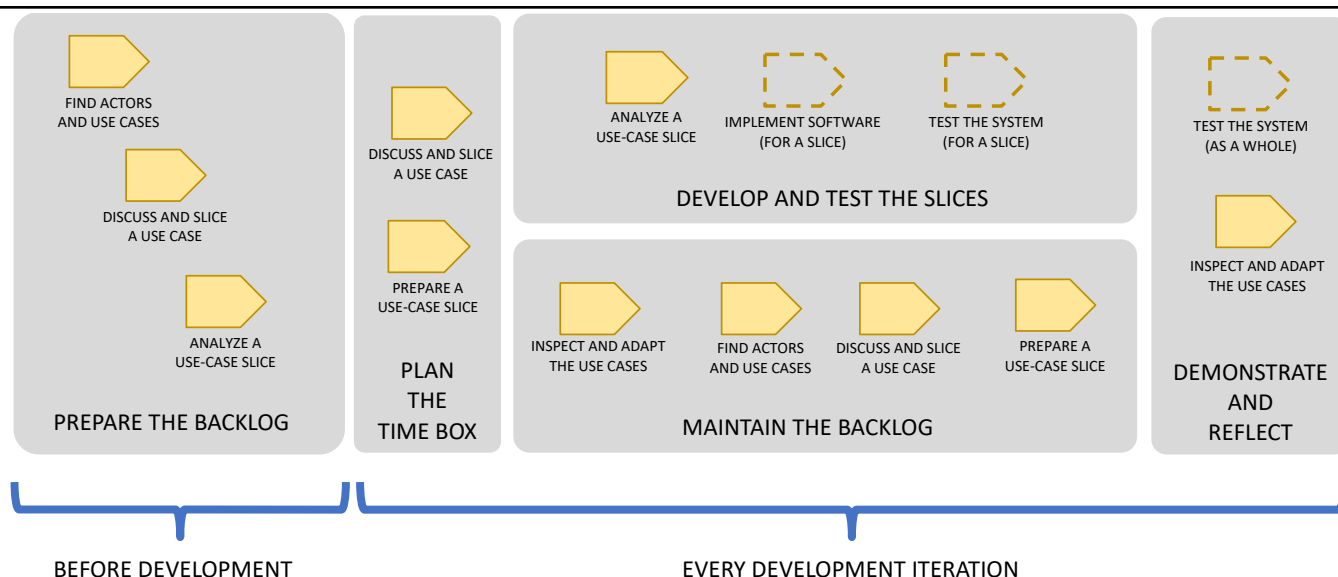


FIGURE 35: USE-CASE 3.0 ACTIVITIES FOR ITERATIVE DEVELOPMENT APPROACHES

Before the development starts the initial backlog is prepared. ‘Find Actors and Use Cases’ is used to build the initial use-case model and scope the system, ‘Discuss and Slice a Use Case’ is used to create the initial set of most important use-case slices to seed the backlog, and ‘Analyze a Use-Case Slice’ is used to assess the impact of the selected use-case slices.

Once the backlog is up and running you can start the first development iteration. Every iteration starts with some planning. During this planning you need to ‘Discuss and Slice the Use Cases’ to further slice the selected use-case slices to make sure they are small enough to complete in the iteration and ‘Prepare a Use-Case Slice’ to create the work items to be populate the iteration’s plan. The development team then uses ‘Analyze a Use-Case Slice’, ‘Implement Software (for a slice)’, and ‘Test the System (for a slice)’ to develop the identified slices and add them to the system.

While the development is on-going the team also uses ‘Inspect and Adapt the Use Cases’, ‘Discuss and Slice a Use Case’ and ‘Prepare a Use-Case Slice’ to maintain the backlog, handle change and make sure there are enough backlog items ready to drive the next iteration. The team may even need to use ‘Find Actors and Use Cases’ to handle major changes or discover more use cases for the team. In Scrum it is recommended that teams spend 5 to 10 per cent of their time maintaining their backlog. This is not an inconsiderable overhead for the team, and Use-Case 3.0 provides the work products and activities needed to do this easily and efficiently.

Finally at the end of the iteration the team needs to demonstrate the system and reflect on their performance during the iteration. The team should use ‘Test the System (as a whole)’ to understand where they are, and ‘Inspect and Adapt Use Cases’ to reflect on the quality and effectiveness of their use cases and use-case slices.

Complementing a Story Backlog with Use-Cases and Use-Case Slices

As discussed throughout this document, primarily in The Use-Case Foundation and Prepare a Use-Case Slice Sections, use-case slices generally involve many steps and changes to many different parts of the system and need to be split into a series of smaller work items for insertion into a team’s plans or story backlog.

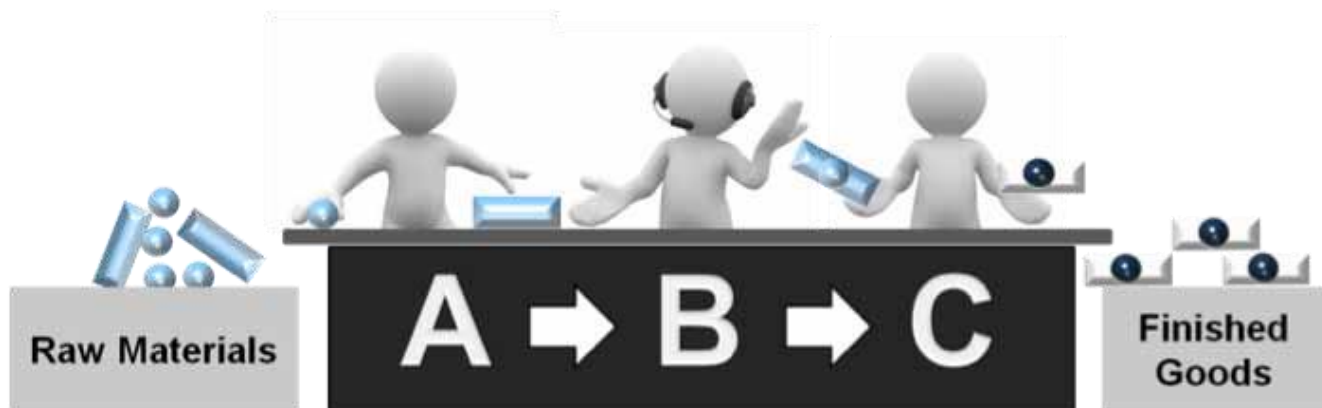
This relationship between the slice representing an end-to-end slice of value and the one or more work items needed to implement the slice makes Use-Case 3.0 100% compatible with teams that are already using User Stories.

For example:

- For teams using Scrum with a User Story Backlog the Use Cases and Use-Case Slices will help them find the right User Stories to populate their backlogs and focus on delivering true user value. They will also help them form more effective Product and Sprint Goals.
- For teams using Epics (big stories) to complement their User Stories (the smaller stories that they take into their Sprints or Iterations) then the slices work perfectly as Epics.

Use-Case 3.0 and one-piece flow

One-piece flow is an approach that avoids the batching of the requirements seen in the iterative and waterfall approaches. In a one-piece flow approach each requirements item flows through the development process. One-piece flow is a technique taken from lean manufacturing. **FIGURE 36: BASIC ONE-PIECE FLOW** shows a small team engaging in one-piece flow passing each item directly from workstation A to B to C.



People icons © ioannis kounadeas / Fotolia

FIGURE 36: BASIC ONE-PIECE FLOW

For this to work effectively you need small, regularly sized items that will flow quickly through the system. For software development the requirements are the raw materials and working software is the finished goods. Use cases would be too irregularly sized and too big to flow through the system. The time at stations A, B and C would be too unpredictable and things would start to get stuck. Use-case slices though can be sized appropriately and tuned to meet the needs of the team. **FIGURE 37** illustrates one-piece flow for software development with use-case slices.



People icons © ioannis kounadeas / Fotolia

FIGURE 37: ONE-PIECE FLOW FOR SOFTWARE DEVELOPMENT WITH USE-CASE SLICES

As well as flowing quickly through the system, there needs to be enough items in the system to keep the team busy. one-piece flow doesn't mean that there is only one requirements item being worked on at a time or that there is only one piece of work between one workstation and the next. Work in progress limits are used to level the flow and prevent any wasteful backlogs from building up.

One-piece flow doesn't mean that individuals only do one thing and only work at one workstation. For example there could be more people working in Development than there are in Test, and if things start to get stuck then the people should move around to do whatever they can to get things moving again. If there are no use-case slices waiting to be tested but there are slices stuck in preparation, then the testers can show some initiative and help to do the preparation work. In the same way you are not limited to one person at each workstation, or even only one instance of each workstation.

Kanban boards are a technique for visualizing the flow through a production line. A Kanban is a sign, flag, or signal within the production process to trigger the production and supply of product as part of just-in-time and lean manufacturing. On a Kanban board Kanban cards are used to represent the items in the system. Figure 37 shows a simple Kanban board for a development team which first analyses each slice to understand its impact, then develops and unit tests the software, and finally independently tests the resulting software before putting it live.

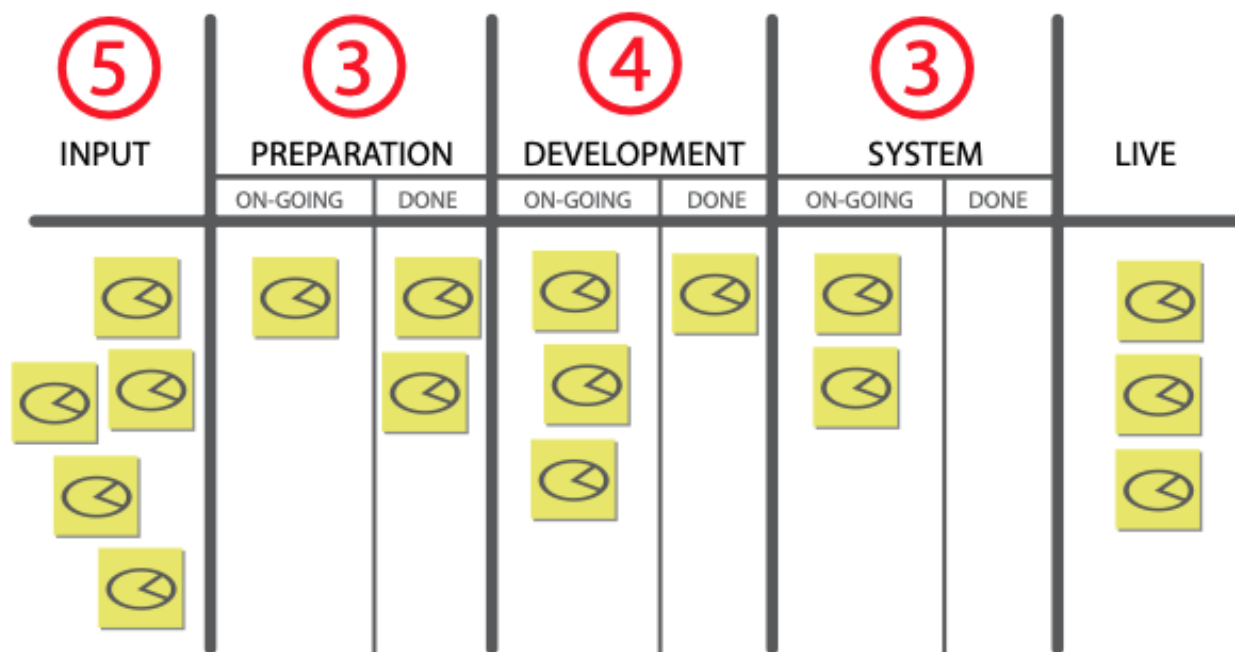


FIGURE 38: USE-CASE SLICES ON A KANBAN BOARD

The work in progress limits are shown in red. Reading from left to right you can see that slices have to be identified and scoped before they are input to the team. Here there is a work in progress limit of 5, and the customers, product owner or business requirements team that are the source of the requirements try to keep 5 use-case slices ready for implementation at all times.

Slices are pulled from the input queue into the preparation area where impact analysis is undertaken, slices are clarified, and the test cases finalized. Here there is a work in progress limit of 3 items. Items in the on-going column are currently being worked on. The items in the done column have had their preparation completed and are waiting to be picked up by a developer. In this way the slices work their way through the development team and after successfully passing the independent system testing go live. A work in progress limit covers all the work at the station, including both the on-going and done items. There is no work in progress limit on the output or the number of items that can go live.

An important thing to note about Kanban is that there is no definitive Kanban board or set of work in progress limits; the structure of the board is dependent on your team structure and working practices. You should tune the board and the work in progress limits as you tune your practices. The states for the use-case slices are a great aid to this kind of work design. [FIGURE 39](#) shows the alignment between the states and the Kanban board shown in [FIGURE 38](#). The states are very powerful as they clearly define what state the slice should be in when it is to be handed on to the next part of the chain.

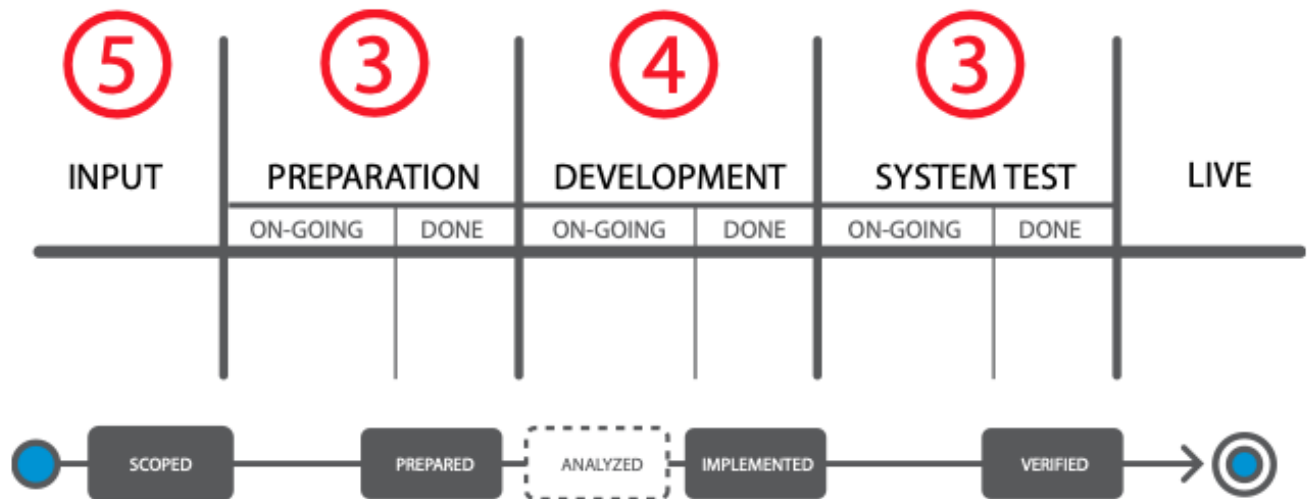


FIGURE 39: ALIGNING THE STATES OF THE USE-CASE SLICE TO THE KANBAN

[FIGURE 40](#) shows where the different Use-Case 3.0 activities are applied. The interesting thing here is that “Inspect and Adapt Use Cases” is not the responsibility of any particular workstation but is conducted as part of the regular quality control done by the team. This activity will help the team to tune the number and type of workstations they have as well as their work in progress limits.

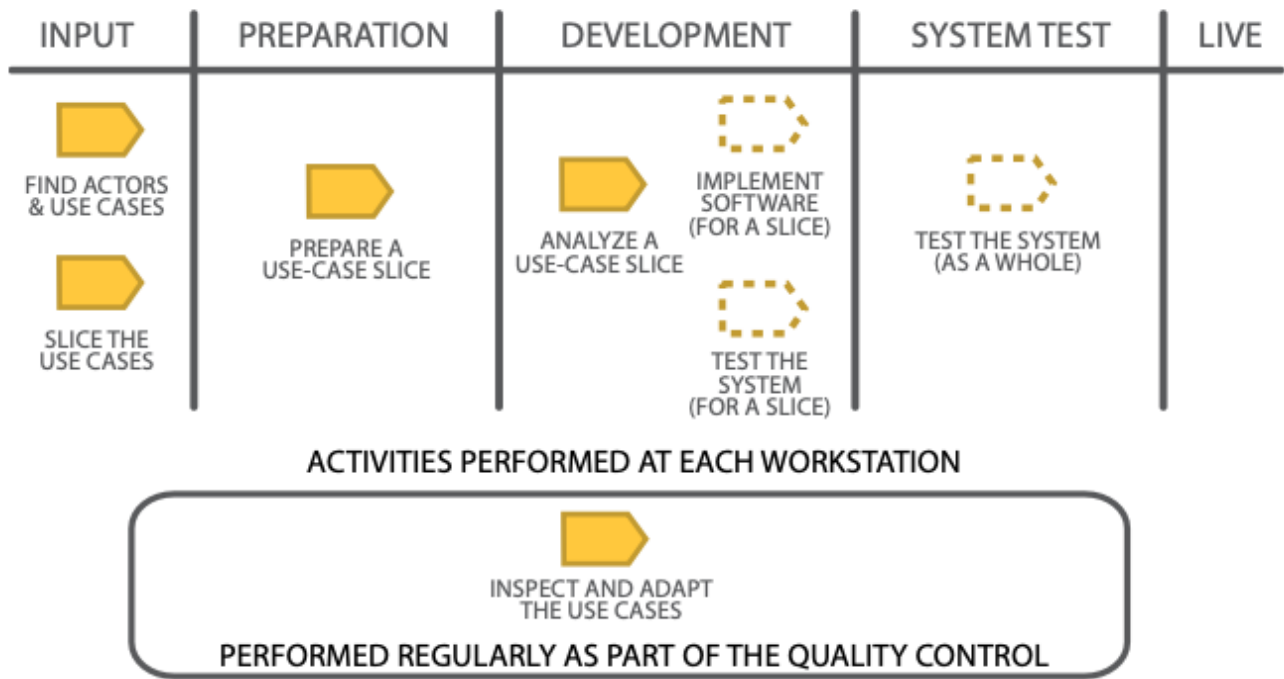


FIGURE 40: USE-CASE 3.0 ACTIVITIES FOR ONE-PIECE FLOW

For example, as a result of reviewing the team’s effectiveness you might decide to eliminate the preparation workstation and increase the work in progress limits for development and system test. Again, you exploit the states of the use-case slice to define what it means for each workstation to have finished their work resulting in the Kanban board shown in [FIGURE 41](#).

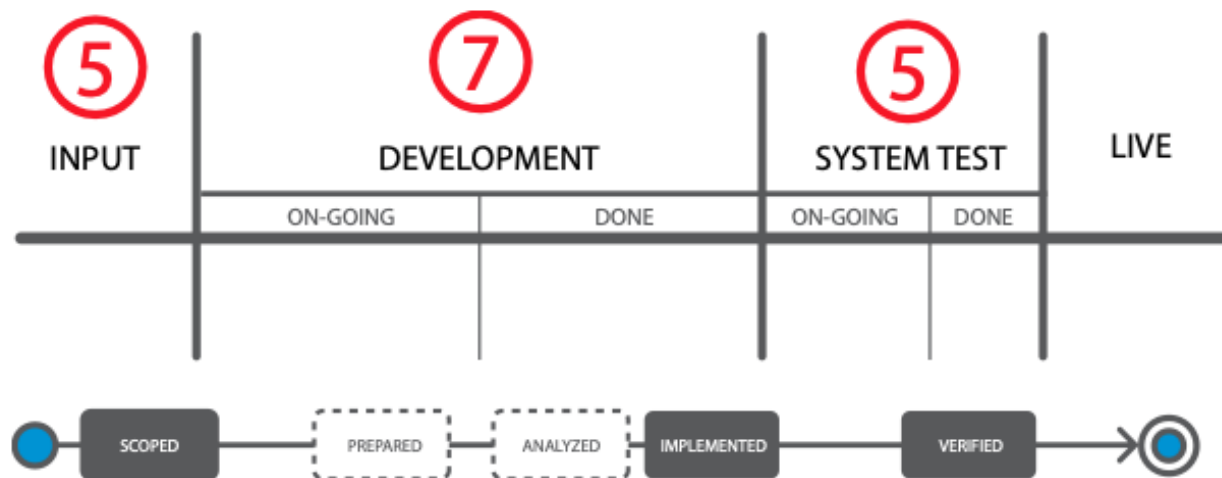


FIGURE 41: THE TEAM'S REVISED KANBAN BOARD SHOWING COMPLETION STATES

Use-Case 3.0 and waterfall

For various reasons you may find that you need to develop your software within the constraints of some form of waterfall governance model. This typically means that some attempt will be made to capture all the requirements up-front before they are handed over to a third-party for development.

When you adopt a waterfall approach the use cases are not continually worked on and refined to allow the final system to emerge but are all defined in one go at the start of the work. They then proceed in perfect synchronization through the other development phases, all of which focus on one type of activity at a time. The typical sequence of activities for a waterfall approach is shown in [FIGURE 42](#).

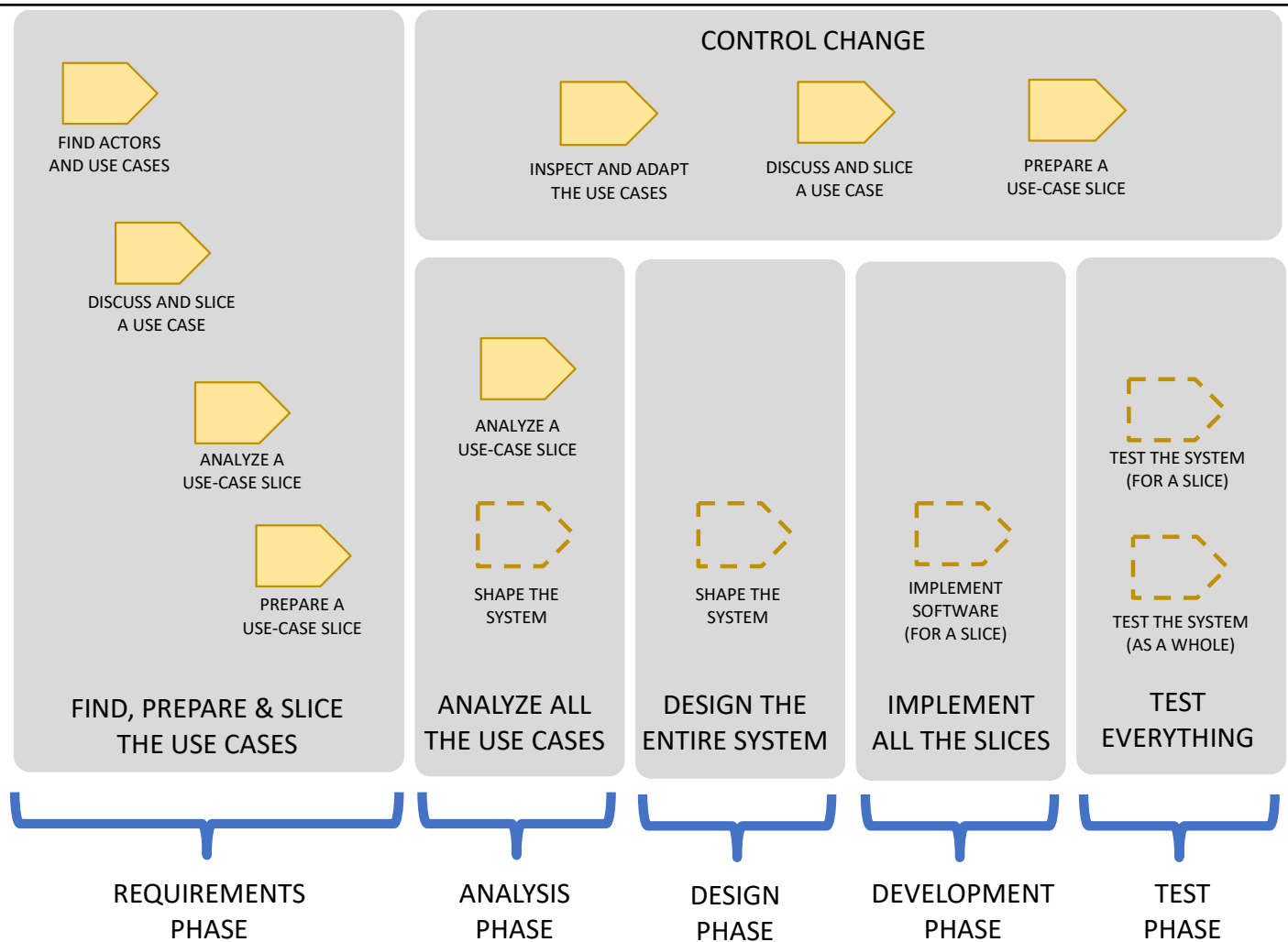


FIGURE 42: USE CASE 3.0 ACTIVITIES FOR WATERFALL APPROACHES

Even within the strictest waterfall environment there are still changes happening during the development of the software itself. Rather than embrace and encourage change, waterfall projects try to control change. They will occasionally ‘Inspect and Adapt the Use Cases’ when there is a change request that cannot be deferred, and they will prepare additional use-case slices to handle any changes that are accepted. They are unlikely to find any further use cases after the requirements phase as this would be considered too large a change in scope.

The ‘one activity at a time’ nature of the waterfall approach means that the make-up of the team is continually changing over time, and so the ability to use face-to-face communication to share the stories is very limited. To cope with this you need to turn up the level of detail on the work products, going way beyond the bare essentials. [FIGURE 43](#) shows the level of detail typically used on waterfall projects.

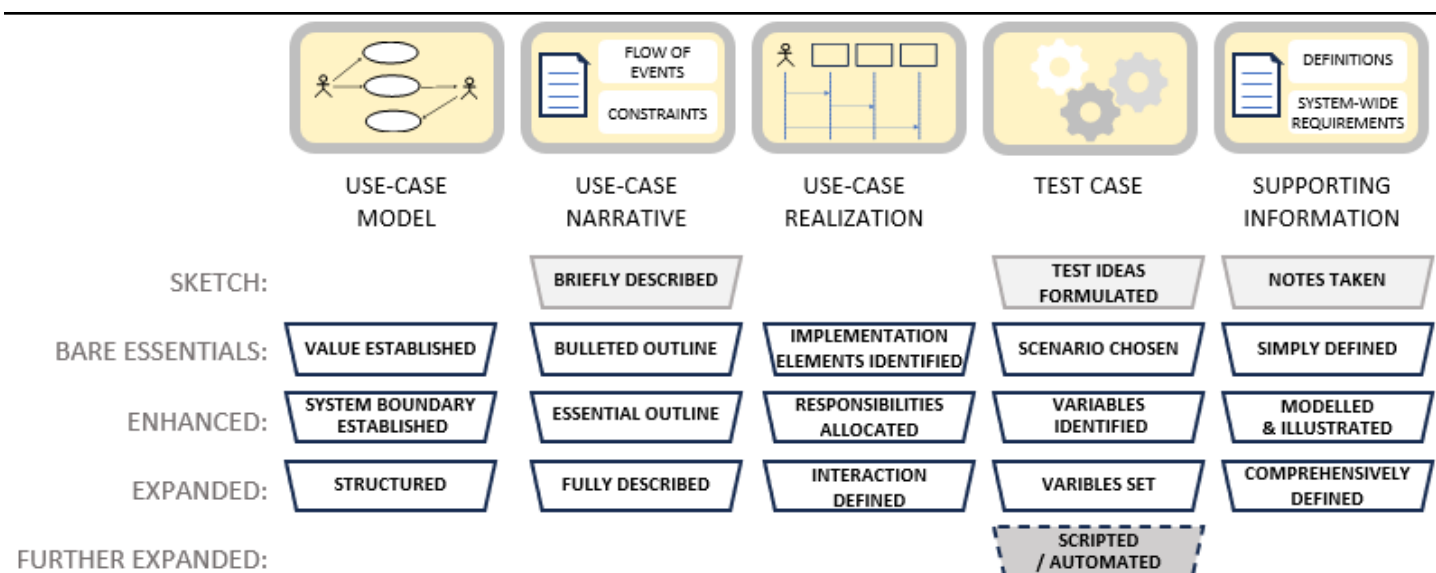


FIGURE 43: LEVELS OF DETAIL FOR THE WORK PRODUCTS WHEN USING A WATERFALL APPROACH

Within each of the development phases one or more of the work products are progressed to a very high-level of detail to ensure that they are 1) complete and 2) answer any and all questions that might arise during the later phases. In the requirements phase the use-case model is worked and re-worked to make sure that all the use cases have been discovered, all the use-case narratives are fully described and the supporting information is comprehensively defined. At this stage some thought will be put into testing and the test ideas formulated. The test cases are then put to one side until the test phase is reached.

The use cases and their supporting information are handed over to the analysis and design team who will flesh out the use-case realizations first to assign responsibilities to the system elements and then to define all the interactions. Eventually coding will start and all the use cases and use-case slices will be implemented. Finally, the testers will get involved and all the test cases will be defined in detail and testing will commence.

The sequential nature of this way-of-working may lead you to think that there is no role for use-case slices to play, and that just handling the entire use cases would be enough. This is not true as the finer grained control provided by the use-case slices allows the requirements team to be much more specific about the actual scope of the system to be built. Even in waterfall projects it is unlikely that you will need all the slices from all of the use cases. They will also help you to handle any last-minute changes in scope caused by schedule or quality problems.

Use-Case 3.0 – It's not just for one type of team

Another important aspect of Use-Case 3.0 is its ability to adapt to existing team structures and job functions whilst encouraging teams to eliminate waste and increase efficiency. To this end Use-Case 3.0 does not pre-define any particular roles or team structures, but it does define a set of states for each of the central elements (the use case and the use-case slice).

As illustrated by the discussion on Use-Case 3.0 and one-piece flow, the states indicate when the items are at rest and could be handed-over from one person or team to another. This allows the practice to be used with teams of all shapes and sizes from small cross-functional teams with little or no handovers to large networks of specialist teams where each state change is the responsibility of a different specialist. Tracking the states and handovers of these elements allows the flow of work through the team (or teams) to be monitored, and teams to adapt their way-of-work to continuously improve their performance.

Use-Case 3.0: Scaling to meet your needs - scaling in, scaling out and scaling up

No one, predefined approach fits everyone, so we need to be able to scale our use of Use-Case 3.0 in several different dimensions:

1. Use cases scale in to provide more guidance to less experienced practitioners (developers, analysts, testers, etc.) or to practitioners who want or need more guidance.
2. They scale out to cover the entire lifecycle, covering not only analysis, design, coding and test but also operational usage and maintenance.
3. They scale up to support large and very large systems such as systems of systems: enterprise systems, product lines, and layered systems. Such systems are complex and are typically developed by many teams working in parallel, at different sites, possibly for different companies, and reusing many legacy systems or packaged solutions.

Regardless of the complexity of the system you are developing you always start in the same way by identifying the most important use cases and creating a big picture summarizing what needs to be built. You can then adapt Use-Case 3.0 to meet the emerging needs of the team. In fact, Use-Case 3.0 insists that you continuously inspect and adapt its usage to eliminate waste, increase throughput and keep pace with the ever-changing demands of the team.

Conclusion

Use-Case 3.0 exists as a set of proven and well-defined practices that are compatible with many other software development practices such as Continuous Integration, Intentional Architecture, and Test-Driven Development. It also works with all popular management practices. In particular, it has the lightness and flexibility to support teams that work in an agile fashion. It also has the completeness and rigor required to support teams that are required to work in a more formal or waterfall environment.

Use-Case 3.0 is:

- **lightweight** - in both its definition and application.
- **scalable** - and suitable for teams and systems of all sizes.
- **versatile** - and suitable for all types of systems and development approaches
- **easy to use** - use-case models can be quickly put in place and the slices created to meet the teams' needs.

Use-Case 3.0 is 100% compatible with teams that are already using user stories.

Use-Case 3.0 is free and offered to the public in this guide and as a set of supporting Essence Practices available from www.ivarjacobson.com. This is offered as a stand-alone set of cards or as part of the Essence Workbench environment.

This is the first of many publications on Use-Case 3.0, you can expect to see many other articles, white papers and blogs on the subject published on www.ivarjacobson.com

Appendix 1: Work Products

This appendix provides definitions and further information of the work products used by Use-Case 3.0. The work products covered are:

- Supporting Information
- Test Case
- Use-Case Model
- Use-Case Narrative
- Use-Case Realization

Supporting Information

The purpose of the supporting information is to capture important terms used to describe the system, and any and all requirements that don't fit inside the use-case model.

The supporting information:

- Helps ensure a common understanding of the specified solution.
- Focuses on concepts and terms that need to be understood by everyone involved in the work, and in particular those terms referenced by the use cases.
- Captures those important global requirements and quality attributes that don't relate to any single use case such as supported platforms and system availability.
- Details any standards that need to be followed. For example, coding, presentation, language, safety, and any other industry standards that apply to the system.
- Helps to identify additional work items not readily identifiable directly from the use-cases, such as those that will be used to demonstrate the different platforms supported or the desired levels of availability.

The role of the supporting information is to support the evolution of the use cases and the implementation of the use-case slices. Capture it to complement your use-case model and avoid miscommunication between the team members. The information can come from many sources, such as other requirements documents, specifications, and discussions with stakeholders and domain experts. You can also include domain, process, and other business models if they are a useful aid to understanding the use-case model and the system it describes.

The supporting information can be documented at varying levels of detail ranging from a simple set of basic definitions through to a comprehensive and fully described set of definitions, standards, and quality attributes. The supporting information can be presented at the following levels of detail:



Notes Taken: A basic level of detail that indicates what is included is just an outline of the most obvious terms and areas to be addressed.

More detail will need to be added if the information is to support the successful identification and preparation of the right use-case slices.



Simply Defined: All terms referenced by the use-case narratives must be defined and the system's global quality attributes clearly specified. At this level of detail these are captured as simple lists of declarative statements such as those used in the glossary that accompanies this e-book.

This is the lightest level of detail which provides support for the development of the use-case slices. It also clarifies the global requirements of the system, enabling the team to tell if the system implementing the slices is truly usable and not just demonstrable. It is suitable for most teams, particularly those that collaborate closely with their users and are able to fill in any missing detail by talking with them.



Modeled and Illustrated: More detail can be added to the supporting information by transforming the basic definitions into models that precisely capture the definitions, their attributes and their relationships, and providing real world examples to clarify things. At this level of detail, we go beyond simple definitions and start to use complementary techniques such as business rule catalogues, information modeling and domain modeling. It is particularly useful for supporting those use-case models where a misunderstanding of the requirements could have severe safety, financial or legal consequences.



Comprehensively Defined: Sometimes it is necessary to clarify the information by providing more detailed explanations and support materials such as comprehensive examples, derivations and cross- references.

At this level of detail, the supporting information becomes more complicated, with more precision, cross-referencing and use of formal specification techniques.

The supporting information provides a central location to look for terms and abbreviations that may be new to the team and to find the global quality attributes that everyone in the team should understand. It is a valuable complement to the use-case model itself. Without the supporting information it can be difficult to understand what it means for the system to be usable and ready for use.

The supporting information is usually represented as a simple list of terms and their definitions. The list is often split up into sections such as definitions of terms, business rules, operational constraints, design constraints, standards, and system-wide requirements. The list may be published as part of a Wiki site to simplify access and maintenance.

Test Case

The purpose of a test case is to provide a clear definition of what it means to complete a slice of the requirements. A test case defines a set of test inputs and expected results for the purpose of evaluating whether or not a system works correctly.

Test cases:

- Provide the building blocks for designing and implementing tests.
- Provide a mechanism to complete and verify the requirements.
- Allow tests to be specified before implementation starts.
- Provide a way to assess system quality.

Test cases are an important, but often neglected, part of a use case. The test cases provide the true definition of what it is that the system is supposed to do to support a use case. The test cases are particularly important when we start to slice up the use cases as they provide the developers with a clear definition of what it means to successfully implement a use-case slice.

Test cases can be used with many forms of requirements practice including use cases, user stories and declarative requirements. In all cases the tester must be presented with a slice of requirements to test, one with a clear beginning and end from which they can derive an executable test scenario.

Test cases can be presented at the following levels of details:



Test Ideas Formulated: The lightest level of detail just captures the initial idea that will inform the test case. When defining a test case, it needs to be clear what the idea behind the test case is and what it is that it is verifying.

More detail will need to be added if the test case is to be executable.



Scenario Chosen: To be able to run a test case a tester must be presented with a test scenario to execute. The structure of the use-case narrative ensures that every use-case slice will present the tester with one or more candidate test scenarios. The art of creating effective test cases is to choose the right subset of the potential test scenarios to fulfill the test idea and clearly define done for the slice.

This is the lightest level of detail that provides an executable test case. Once the scenario has been chosen the test case is defined enough to support exploratory and investigative testing. This can be very useful early in the project lifecycle when the insight provided by testing the system is invaluable, but the specification (and solution) may not be stable enough to support formal, scripted testing.



Variables Identified: A test case takes some inputs, manipulates system states, and produces some results. These variables appear as inputs, internal states, and outputs in the requirements. At this level of detail, the acceptable ranges for the key variables involved in the scenario are explicitly identified.

This level of detail is suitable for those test cases where soliciting the opinion of the tester is an essential part of the test, for example when undertaking usability testing. It can also be used when more structure is needed for exploratory and investigative testing.



Variables Set: The test case can be further elaborated by explicitly providing specific values for all the variables involved in the test case.

This level of detail is suitable for manual test cases, as all the information needed by an intelligent tester to repeatedly and consistently execute the test case is in place.



Scripted / Automated: If a test case is to be used many times or to support many different tests then it is worth making the effort to fully script or automate it.

At this level of detail, the test case can be executed without any intervention or additional decision making.

The test cases are the most important work product associated with a use case; remember it is the test cases that define what it means to complete the development of a use case, not the use-case narrative. In a way the test cases are the best form of requirements you can have.

The test cases will be used throughout the lifetime of the system - they are not just used during the implementation of the use cases but are also used as the basis for regression testing and other quality checks. The good news is that the structure of the use cases and use-case narratives naturally leads to well-formed, robust, and resilient test cases; ones that will last as long as the system continues to support the use cases.

The use-case narratives are collections of flows that the system must support, and for each flow described in the use-case narrative there will have to be at least one test case. You create the test cases at the same time as the use-case narratives as part of preparing a use-case slice for development.

Use-Case Model


A use-case model is a model of all of the useful ways to use a system, and the value that they will provide. The purpose of a use-case model is to capture all the useful ways to use a system in an accessible format that captures a system's requirements and can be used to drive its development and testing.

A use-case model:


- Allows teams to agree on the required functionality and characteristics of a system.
- Clearly establishes the boundary and scope of the system by providing a complete picture of its actors (being outside the system) and use cases (being inside the system).
- Enables agile requirements management.

A use-case model is primarily made up of a set of actors and use cases, and diagrams illustrating their relationships. Use-case models can be captured in many different ways including as part of a Wiki, on a white board or flipchart, as a set of PowerPoint slides, in a MS Word document, or in a modeling tool.


The use-case model can be prepared at different levels of detail:

 **Value Established:** The first step towards a complete use-case model is to identify the most important actors and use cases - the primary ones. These are the ones that provide the value of the system.

This is the lightest level of detail. It is suitable for most projects, particularly those adding new functionality to existing systems where there is little or no value in modeling all the things the system already does.

 **System Boundary Established:** The primary actors and use cases capture the essence of why the system is built. They show how the users will get value from the system. They may not provide enough value to set it up and keep it running. In these cases, supporting actors and operational use cases are necessary to enable and support the effective operation of the system.

This level of detail is useful when modeling brand new systems or new generations of existing systems. At this level of detail all the systems actors and use cases are identified and modeled.

 **Structured:** The use-case model often contains redundant information, such as common sequences or patterns of interaction. Structuring the use-case model is the way to deal with these redundancies.

For large and complex systems, especially those that are used to provide similar functionality within many different contexts, structuring the use-case model can aid understanding, eliminate waste, and help you find reusable elements.

As long as your use-case model clearly shows the value that the stakeholders will receive from your new or updated system then it is doing its job. Care should be taken when adding detail to the model. Only advance to System Boundary Established or Structured if these levels of detail are clearly going to add value and help you deliver the new system more efficiently.

Use-Case Narrative

The purpose of a use-case narrative is to tell the story of how the system and its actors work together to achieve a particular goal.

Use-case narratives:

- Identify the flows used to explore the requirements and identify the use-case slices
- Describe a sequence of actions, including variants that a system and its actors can perform to achieve a goal.
- Are presented as a set of flows that describe how an actor uses a system to achieve a goal, and what the system does for the actor to help achieve that goal.
- Capture the requirements information needed to support the other development activities.

Use-case narratives can be captured in many ways including as part of a Wiki, on index cards, as MS Word documents, or inside one of the many commercially available work management, requirements management or modeling tools.

Use-case narratives can be developed at different levels of detail ranging from a bulleted outline, identifying the basic flow and the most important variants, through to a comprehensive, highly detailed specification that defines all the actions, inputs and outputs involved in performing the use case. Use-Case narratives can be prepared at the following levels of detail:



Briefly Described: The lightest level of detail that just captures the goal of the use case and which actor starts it.

This level of detail is suitable for those use cases you decide not to implement. More detail will be needed if the use case is to be sliced up for implementation.



Bulleted Outline: The use case must be outlined in order to understand its size and complexity. This level of detail also enables effective scope management as the outline allows the different parts of the use case to be prioritized against one another and, if necessary, targeted onto different releases.

This is the lightest level of detail that enables the use case to be sliced up and development to progress. It is suitable for those teams that are in close collaboration with their users and are able to fill in any missing detail via conversations and the completion of the accompanying test cases.



Essential Outline: Sometimes it is necessary to clarify the responsibilities of the system and its actors whilst undertaking the use case. A bulleted outline captures their responsibilities but does not clearly define which parts of the use case are undertaken by the system and which are undertaken by the actor(s).

At this level of detail, the narrative becomes a description of the dialog between the system and its actors. It is particularly useful when establishing the architecture of a new system or trying to establish a new user experience.



Fully Described: Use-case narratives can be used to provide a highly detailed requirements specification by evolving them to their most comprehensive level of detail, fully described. The extra detail may be needed to cover for the absence of expertise within the team, a lack of access to the stakeholders or to effectively communicate complex requirements.

This level of detail is particularly useful for those use cases where a misunderstanding of the contents could have severe safety, financial or legal consequences. It can also be useful when off-shoring or out-sourcing software development.

The use-case narrative is a very flexible work product that can be expanded to capture the amount of detail you need to be successful whatever your circumstances. If you are part of a small team working collaboratively with the customer on an exploratory project then bulleted outlines will provide a very lightweight way of discovering the requirements. If you are working in a more rigid environment where there is little access to the real experts, then essential outlines or fully described narratives can be used to plug the gaps in the team's knowledge.

Not every use-case narrative needs to be taken to the same level of detail - it is not uncommon for the most important and risky use cases to be more detailed than the others. The same goes for the sections of the use-case narrative - the most important, complex, or risky parts of a use case are often described in more detail than the others.

Use-Case Realization

The purpose of a use-case realization is to show how the system's elements, such as components, programs, stored procedures, configuration files and database tables, collaborate to perform a use case.

Use-case realizations:

- Identify the system elements involved in the use cases.
- Capture the responsibilities of the system elements when performing the use case.
- Describe how the system elements interact to perform the use case.
- Translate the business language used in the use-case narratives into the developer language used to describe the system's implementation.

Use-case realizations are incredibly useful and can be used to drive the creation and validation of many of the different views teams use to design and build their systems. For example, user interface designers use use-case realizations (in the form of storyboards) to explore the impact of the use cases on the user interface. Architects use use-case realizations to analyze the architecturally significant use cases and assess whether or not the architecture is fit for purpose.

Use-case realizations can be presented in many different formats - the format of the realization is completely dependent on the team's development practices. Common ways of expressing use-case realizations include simple tables, storyboards, sequence diagrams, collaboration diagrams, and data-flow diagrams. The important thing is that the team creates a realization to identify which system elements are involved in the implementation of the use case and how they will change.

Create a use-case realization for each use case to identify the system elements involved in performing it and, most importantly, assess how much they will have to be changed. You can think of the use-case realizations as providing the 'how' to complement the use-case narratives 'what'.

Use-case realizations can be presented at the following levels of detail:



Implementation Elements Identified: The lightest level of detail that just captures the elements of the system, both new and existing, that will participate in the use case.

This level of detail is suitable for small teams, working in close collaboration and developing simple systems with a known architecture. You may need to add more detail if your system is complex, or your team is large or distributed.



Responsibilities Allocated: To allow the team to be able to update the affected system elements in parallel, or in support of multiple slices, the developers need to understand the responsibilities of the individual elements. The responsibilities provide a high-level definition of what each element needs to do, store and track.

This level of detail is suitable for situations where each use-case slice touches on multiple system elements, or where the slices will be developed by multiple developers working in parallel. It should also be used when the architecture of the system is immature, and the overall responsibilities of the system elements have yet to be understood.



Interaction Defined: To provide a complete, unambiguous definition of the changes required to each system element involved in the use case, the use-case realization must include details of all the interfaces and interactions involved in performing the use case.

This level of detail is particularly useful for those use cases where the system design is complex or challenging. It can also be useful when the system elements are to be developed by developers with

little or no knowledge of the design of the system, no access to experienced designers, and no remit to re-factor or alter the design. It is also useful when dealing with inexperienced developers who are still learning their trade.

The use-case realization is a very flexible work product, teams can expand their realization to add more detail as and when they need it. If a small team is doing all their analysis and design collaboratively then simple, lightweight use-case realizations will be sufficient. If a large team, that is unable to have lots of collaborative sessions, is developing a complex system then more detailed realizations will aid communication, and make sure that all the developers have all the information they need to successfully deliver their system elements and implement their use-case slices.

Glossary of Terms

Actor: An actor defines a role that a user can play when interacting with the system. A user can either be an individual or another system. Actors have a name and a brief description, and they are associated to the use cases with which they interact.

Alternative Flow: Description of variant or optional behavior as part of a use-case narrative. Alternative flows are defined relative to the use case's basic flow.

Application: Computer software designed to help the actors in performing specific tasks.

Aspect-Oriented Programming: A programming technique that aims to increase modularity by allowing the separation of cross-cutting concerns (see http://en.wikipedia.org/wiki/Aspect-oriented_programming).

Basic Flow: The description of the normal, expected path through the use case. This is the path taken by most of the users most of the time; it is the most important part of the use-case narrative.

Customer: The stakeholder who is paying for the development of the system or who is expected to purchase the system once it is complete.

Flow: A description of some full or partial path through a use-case narrative. There is always at least a basic flow, and there may be alternative flows.

Requirements: What the software system must do to satisfy the stakeholders.

Separation of Concerns: The process of splitting up a system to minimize the overlap in functionality (see http://en.wikipedia.org/wiki/Separation_of_concerns).

Software System: A system made up of software, hardware, and digital information, and that provides its primary value by the execution of the software.

A software system can be part of a larger software, hardware, business or social solution.

Stakeholder: A person, group or organization who affects or is affected by the software system.

System: A group of things or parts working together or connected in some way to form a whole. Typically used to refer to the subject of the use-case model: the product to be built.

System Element: Member of a set of elements that constitutes a system (ISO/IEC 15288:2008)

Test Case: A test case defines a set of test inputs and expected results for the purpose of evaluating whether or not a system works correctly.

Use case: A use case is all the ways of using a system to achieve a particular goal for a particular user.

Use-Case 3.0: A scalable set of agile practices that uses use-cases to capture a set of requirements and drive the incremental development of a system to fulfill them.

Use-Case Diagram: A diagram showing a number of actors and use cases, and their relationships.

Use-Case Model: A model of all of the useful ways to use a system, and the value that it will provide. A use-case model is primarily made up of a set of actors and a set of use cases, and diagrams illustrating their relationships.

Use-Case Narrative: A description of a use case that tells the story of how the system and its actors work together to achieve a particular goal. It includes a sequence of actions (including variants) that a system and its actors can perform to achieve a goal.

Use-Case Slice: A use-case slice is one or more stories selected from a use case to form a work item that is of clear value to the customer.

User: A stakeholder who interacts with the system to achieve its goals.

User Story: A short, simple description of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

Acknowledgements

General

Use-Case 3.0 is based on industry-accepted best practices, used and proven for decades. We would like to thank the tens of thousands of people who use use cases every day on their projects and in particular those who share their experiences inside and outside their own organizations. Without all of their hard work and enthusiasm we wouldn't have the motivation or knowledge to attempt this evolution of the technique. We hope you find this e-book useful, and continue to inspect and adapt the way that you apply use cases.

People

We would also like to thank everyone who has directly contributed to the creation of this or previous versions of this e-book including, in no particular order, Kurt Bittner, Paul MacMahon, Richard Schaff, Eric Lopes Cardozo, Svante Lidman, Craig Lucia, Tony Ludwig, Ron Garton, Burkhard Perken-Golomb, Arran Hartgroves, James Gamble, Brian Hooper, Stefan Bylund, and Pan-Wei Ng.

Bibliography

Object-Oriented Software engineering: A Use Case Driven Approach Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard The original book that introduced use cases to the world.

- Publisher: Addison-Wesley Professional; revised edition (July 10, 1992)
- ISBN-10: 0201544350 ISBN-13: 978-0201544350

The Object Advantage: Business Process Reengineering With object Technology Ivar Jacobson, Maria Ericsson, Agneta Jacobson

The definitive guide to using use cases for business process reengineering.

- Publisher: Addison-Wesley Professional (September 30, 1994)
- ISBN-10: 0201422891 ISBN-13: 978-0201422894

Software Reuse: Architecture, Process and organization for Business Success Ivar Jacobson, Martin Griss, Patrik Jonsson

A comprehensive guide to software reuse, including in-depth guidance on using use cases for the development of product lines and systems-of-interconnected systems.

- Publisher: Addison-Wesley Professional (June 1, 1997)
- language: English
- ISBN-10: 0201924765 ISBN-13: 978-0201924763

Use-Case Modeling

Kurt Bittner and Ian Spence

The definitive guide to creating use-case models and writing good use cases.

- Publisher: Addison-Wesley Professional; 1 edition (August 30, 2002)
- ISBN-10: 0201709139 ISBN-13: 978-0201709131

Aspect-oriented Software Development with Use Cases Ivar Jacobson and Pan-Wei Ng

The book that introduced the world to use-case slices in their previous guise as use-case modules.

- Publisher: Addison-Wesley Professional; 1 edition (January 9, 2005)
- ISBN-10: 0321268881 ISBN-13: 978-0321268884

Use Cases are Essential Ivar Jacobson and Alistair Cockburn

The co-authored paper that launched a new wave of interest in Use Cases

- acmqueue November 11, 2023. Volume 21, issue 5.

Use Cases Foundation Ivar Jacobson and Alistair Cockburn

[A concise paper](#) published in 2024 which defines the common ground that lies between the two most widely practiced ways of working with use cases.

Unifying User Stories, Use Cases, Story Maps: The power of verbs Alistair Cockburn

Alistair Cockburn's new guide to what makes use cases, user stories and story maps all work, and what makes them work well together.

- Publisher: Humans and Technology Press; 1 edition (May, 2024)
- ISBN 978-1-7375197-6-8

About the Authors

Ivar Jacobson

Dr. Ivar Jacobson is a father of components and component architecture, use cases, aspect-oriented software development, modern business engineering, the Unified Modeling Language and the Rational Unified Process. His latest contribution to the software industry is a formal practice concept that promotes practices as the ‘first-class citizens’ of software development and views process simply as a composition of practices. He is the principal author of six influential and best-selling books. He is a keynote speaker at many large conferences around the world and has trained several process improvement consultants.

Ian Spence

As Chief Technology officer at Ivar Jacobson International, Ian Spence specialized in the agile application of the Unified Process. He is a certified RUP practitioner, ScrumMaster, SAFe Fellow and an experience coach having worked with 100s of projects to introduce iterative and agile techniques. He has over 30 years experience in the software industry, covering the complete development lifecycle, including requirements capture, architecture, analysis, design, implementation and project management. His specialty subjects are iterative project management, agile team working and requirements management with use cases. In his role as CTO, Ian contributed to the technical direction of Ivar Jacobson International and he continues to work with the company’s Technology office to define the next generation of smart, active, software development practices. He was the project lead and process architect for the development of the Essential Unified Process and the practices it contains. When he is not working on researching, capturing and defining practices he spends his time assisting companies in the creation and execution of change programs to improve their software development capability. He is co-author of the Addison Wesley books “Use Case Modeling” and “Managing Iterative Software Development Projects”.

Keith de Mendonca

Dr Keith de Mendonca is a principal consultant at Ivar Jacobson International. He has worked with a number of industry luminaries to essentialize software practices using the Essence language. As a Chief Technology Architect and Chief Site Technologist at Symbian Ltd, he worked with software engineers in the UK, India and China to develop an operating system for the first wave of smartphones - used by Nokia, Sony Ericsson, Fujitsu and many other phone manufacturers. He returned to the UK to join Ivar Jacobson International in 2015. Keith has helped many medium-sized and large enterprises improve productivity and become more innovative by adopting lean-agile development practices at the team and at the enterprise level.



About Ivar Jacobson International

IJI is a global services company specializing in improving the performance of software development teams by removing barriers to the adoption of new practices. Through the provision of high caliber people, innovative practices, and proven solutions, we ensure that our customers achieve strong business/IT alignment, high performing teams, and projects that deliver.

www.ivarjacobson.com

Switzerland

+41 (0) 79 330 16 57

United Kingdom

+44 (0) 20 3934 0278

Sweden

+46 (0) 8 515 10 174