USE-CASE 2.0

The Hub of Software Development

IVAR JACOBSON, IAN SPENCE, AND BRIAN KERR

Use cases have been around for almost 30 years as a requirements approach and have been part of the inspiration for more-recent techniques such as user stories. Now the inspiration has flown in the other direction. Use-Case 2.0 is the new generation of use-case-driven development—light, agile, and lean—inspired by user stories and the agile methodologies Scrum and Kanban.

Use-Case 2.0 has all the popular values from the past—not just supporting requirements, but also architecture, design, test, and user experience—and it is instrumental in business modeling and software reuse.

Use-Case 2.0 has been inspired by user stories to assist with backlogs à la Scrum and one-piece flow with Kanban, with the introduction of an important new concept, the use-case slice.

This article makes the argument that use cases essentially include the techniques that are provided by user stories but offer significantly more for larger systems, larger teams, and more complex and demanding developments. They are as lightweight as user stories, but can also scale in a smooth and structured way to incorporate as much detail

as needed. Most importantly, they drive and connect many other aspects of software development.

USE CASES—WHY STILL SUCCESSFUL AND POPULAR?

Use cases were introduced at OOPSLA 87 (Object-Oriented Programming Systems, Languages, and Applications),[6] although they were not widely adopted until the publication of the 1992 book *Object-Oriented Software Engineering—a Use-Case–Driven Approach*.[7] Since then many other authors have adopted parts of the idea, notably Alistair Cockburn[2] concerning requirements and Larry Constantine[4] regarding designing for better user experiences. Use cases were adopted as a part of the standard UML (Unified Modeling Language[1]), and its diagrams (the use case and the actor icons) are among the most widely used parts of the language. Many other books and papers have been written about use cases for all kinds of systems—not just for software, but also business, systems (such as embedded systems), and systems of systems. Focusing on today and the future, the latest macro-trends, IoT (Internet of Things) and Industrial Internet, have made use cases their choice.[9]

The use-case practice has evolved over the years, inspired by ideas from many different people, with the newer ideas incorporated into Use-Case 2.0. One new idea is slicing a use case into use-case slices.[5,8] The idea of sizing these slices to become suitable backlog items and relating them to user stories is at the core of Use-Case 2.0.

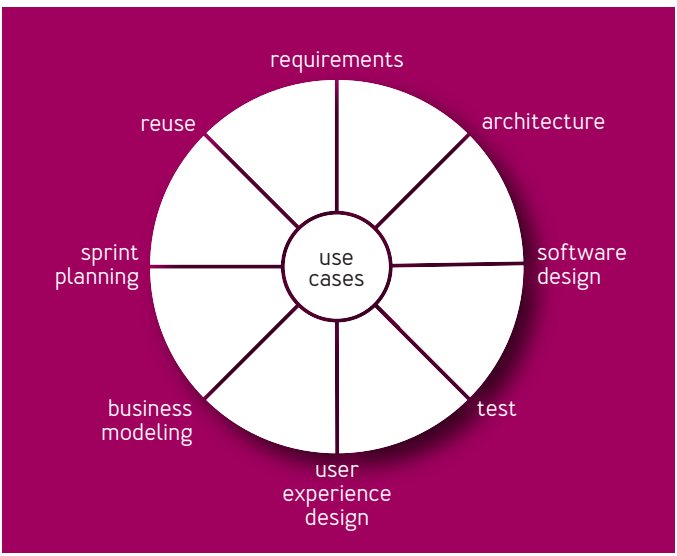
Use cases can and should be used to drive software development. They do not prescribe how you should plan or

manage your development work, or how you should design, develop, or test your system. They do, however, provide a structure for the successful adoption of your selected management and development practices.

The reason for the success of the use-case approach is not just that it is a very practical technique to capture *requirements* from a usage perspective or to design practical *user experiences*, but it impacts the whole development lifecycle. The key use cases—or to be more precise, the key use-case slices (a slice being a carefully selected part of a use case)—assist systematically in finding the application *architecture*. They drive the identification of components or other software elements in *software design*. They are the elements that have to go through *testing*—and truly support test-driven design. They are the elements to put in the backlog when planning *sprints* or to put on the canvas using Kanban. The use cases of a business are the processes of the business; thus, the advantage of doing *business modeling* with use cases is that it leads directly to finding the use cases of the system to be developed to support the business. Moreover, use cases help in finding commonalities, which directs the architecture work to achieve software *reuse*.

There are many more similar values in applying use cases, but most important is that the idea of use cases is intuitively graspable. This is a lightweight, lean and agile, scalable, versatile, and easy-to-use approach. Many people who hear about use cases for the first time takes them to heart; many start using the term in everyday-life situations without thinking about all the details that help with so many aspects

FIGURE 1: **USE CASES ARE THE HUB OF SOFTWARE DEVELOPMENT**



of software development—the aspects that are the spokes of the software-development wheel in which use cases are the hub (see figure 1).

Thus, it seems clear that use cases have stood the test of time and have a very healthy future.

PRINCIPLES FOR USE-CASE ADOPTION

There are six basic principles at the heart of any successful application of use cases:

1. Keep it simple by telling stories.
2. Understand the big picture.
3. Focus on value.
4. Build the system in slices.

5. Deliver the system in increments.
6. Adapt to meet the team's needs.

Principle 1: Keep it simple by telling stories

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.

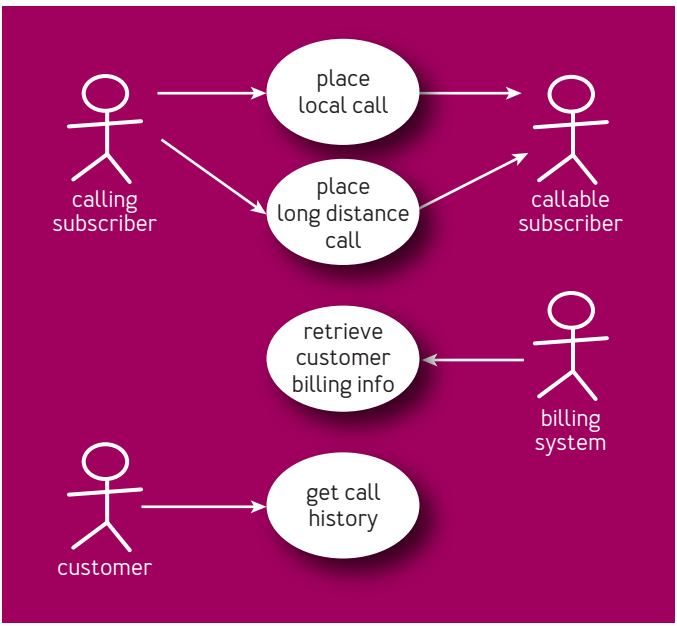
Use cases capture the goals of the system. To understand a use case, we tell stories. The stories cover how to achieve the goal and how to handle problems that occur on the way. Use cases provide a way to identify and capture all the different but related stories in a simple but comprehensive way. This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

Principle 2: Understand the big picture

Whether the system you are developing is large or small, whether it is a software system, a hardware system, or a business system, understanding the big picture is essential. 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, what it will cost, and what benefit it will provide.

A use-case diagram is a simple way of presenting an overview of a system's requirements. Figure 2 is the use-case diagram for a simple telephone system. This picture shows all the ways the system can be used, who starts the

FIGURE 2: **USE-CASE DIAGRAM FOR A SIMPLE TELEPHONE SYSTEM**



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 be other systems and, in some cases, both (for example, the role of the called subscriber might be an answering machine and not a person).

Principle 3: Focus on value

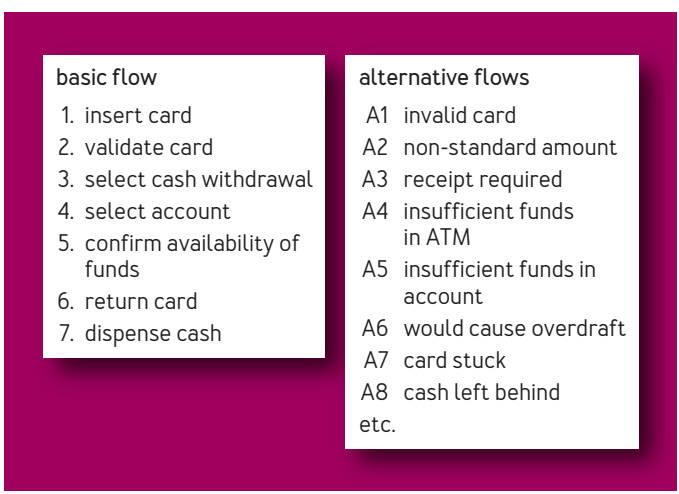
When trying to understand how a system will be used, it is always important to focus on the value it will provide to its users and other stakeholders. Value is generated only if the

system is actually used, so it is much better to focus on how the system will be used than on 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.

Figure 3 shows a use-case narrative structured in this way for the cash-withdrawal use case of a cash machine. 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 stories, helping to find the test cases that complete their definition.

FIGURE 3: **STRUCTURE OF A USE-CASE NARRATIVE**



This kind of bulleted outline may be enough to capture the stories and drive the development, or it may need to be elaborated on as the team explores the details of what the system needs to do.

Principle 4: Build the system in slices

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 has to 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. 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 and start building it.

This is the approach taken by Use-Case 2.0, where the use cases are sliced up to provide suitably sized work items, and where the system itself evolves slice by slice.

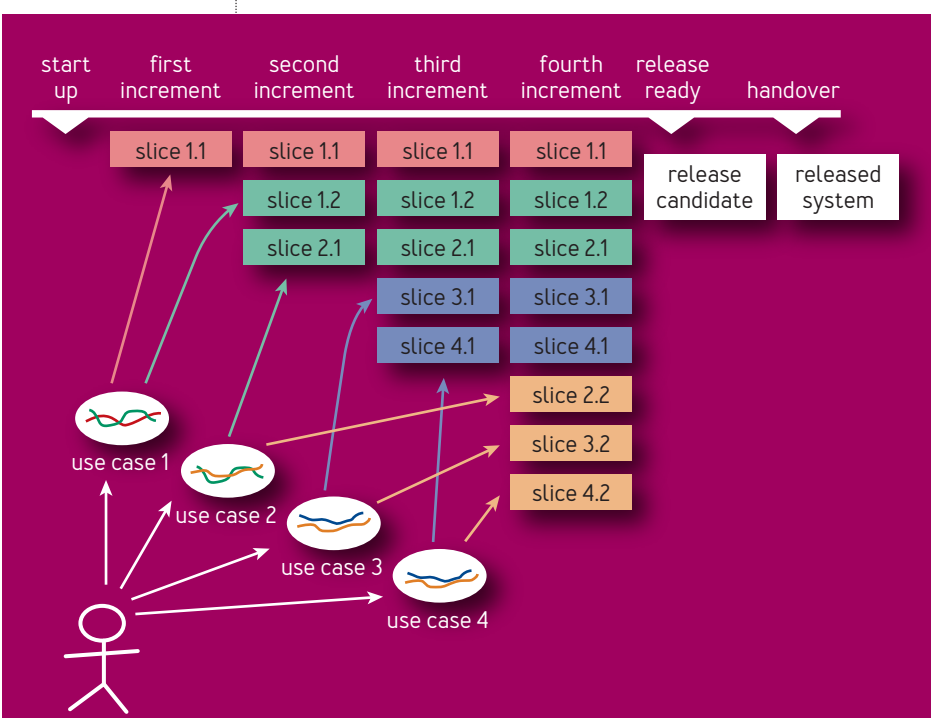
Although use cases have traditionally been used to help understand and capture requirements, they have always been about more than this. The use-case slices slice through more than just the requirements; they also slice through all the other aspects of the system and its documentation, including the design, implementation, test cases, and test results.

Principle 5: Deliver the system in increments

Most software 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 evolve through a series of increments. Each increment provides a demonstrable or usable version of the system. This is the way that all systems should be produced.

Figure 4 shows the incremental development of a system

FIGURE 4: **USE CASES, USE-CASE SLICES, INCREMENTS, AND RELEASES**



release. The first increment contains only 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.

Principle 6: Adapt to meet the team's needs

Unfortunately, there is no one-size-fits-all solution to the challenges of software 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.

Use-Case 2.0 is designed with this in mind and can be as light as desired. Small, collaborative teams can have very lightweight use-case narratives that capture the bare essentials of the stories. 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 or not 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 USE-CASE 2.0 PRACTICE

The Use-Case 2.0 practice describes the key concepts to work with, the work products used to describe them, and a set of activities.

Concepts to work with

Use-Case 2.0 encompasses 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 2.0 are the *use case*, the *use-case slice*, and the *story*.

Use cases capture the requirements, and each use case is scope managed by slicing it up into a set of use-case slices that can be worked on independently. Telling stories bridges the gaps between the stakeholders, the use cases, and the use-case slices. This is how the stakeholders communicate their requirements and explore the use cases. Understanding the stories is also the mechanism for finding the right use-case slices to drive the implementation of the system.

Use Cases

A use case is:

- A sequence of actions a system performs that yields an observable result of value to a particular user.
- That specific behavior of a system, which participates in collaboration with a user to deliver something of value for that user.
- The smallest unit of activity that provides a meaningful result to the user.
- The context for a set of related requirements.

Taken together, the set of all use cases gives us all the functional requirements of the system.

The way to understand a use case is to tell stories. These stories cover both how to achieve a goal and how to handle

any problems that occur on the way. They help developers understand the use case and implement it slice by slice.

A use case undergoes several defined state changes, beginning with just having its *goal established*, through *story structure understood*, *simplest story fulfilled*, *sufficient stories fulfilled*, to *all stories fulfilled*. The states constitute important waypoints in the understanding and implementation of the use case.

Use-Case Slices

Use cases cover many related stories of varying importance and priority. There are often too many stories to deliver in a single release and generally too many to work on in a single increment. Hence, there is a need for dividing use cases into smaller pieces.

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. It acts as a placeholder for all the work required to complete the implementation of the selected stories. 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 2.0, as it is used not only to help with the requirements, but also to drive the development of a system to fulfill them.

A use-case slice undergoes several state changes, from its initial identification where it is *scoped*, through being *prepared*, *analyzed*, *implemented*, and, finally, *verified*. These states allow for planning and tracking the understanding, implementation, and testing of the use-case slice.

To the casual observer glancing at the states, this might look like a waterfall process. There's a big difference, though, as this involves 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 is being analyzed.

Stories

Telling stories is how developers explore use cases with stakeholders. Each story of value to the users and other stakeholders is a thread through one of the use cases. The stories can be functional or non-functional in nature.

A story is described by part of the use-case narrative, one or more flows and special requirements, and one or more test cases. The key to finding effective stories is to understand the structure of the use-case narrative. The network of flows can be thought of as a map that summarizes all the stories needed to describe the use case. In the previous cash-machine example in figure 3, you could identify specific stories such as "Withdraw a standard amount of $100," "Withdraw a nonstandard amount of $75 and get a receipt," or "Respond to an invalid card."

Each story traverses one or more flows beginning with the use case at the start of the basic flow and terminating with the use case at the end of the basic flow. This ensures that all the stories 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.

Work Products

Use cases and use-case slices are supported by a number of work products that the team uses to help share, understand, and document them.

A *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 individual use cases.

Use cases are explored by telling stories. Each use case is described by (1) a *use-case narrative* that outlines its stories as a set of flows; and (2) a set of *test cases* that complete the stories. These can be complemented with a set of special requirements that apply to the whole use case and are often non-functional. These will influence the stories, help assign the right stories to the use-case slices for implementation, and, most importantly, define the right test cases.

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 stories in the use-case narratives. It also captures any systemwide requirements that apply to all of the use cases.

A *use-case realization* can be created to show how the system's elements collaborate to perform a use case. Think of the use-case realization as providing the "how" to complement the use-case narrative's "what." Common ways of expressing use-case realizations include simple tables, storyboards, or sequence diagrams.

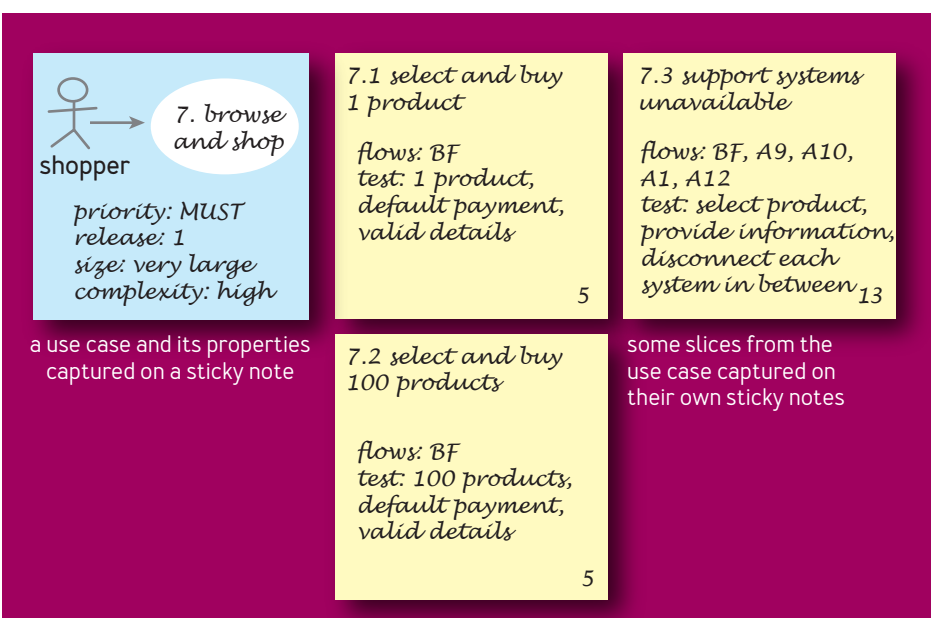
Working with the use cases and use-case slices

In addition to creating and tracking the work products, developers need to track the states and properties of use cases and use-case slices. This can be done in many ways and with many tools. The states can be tracked very simply using sticky notes or spreadsheets. If more formality is required, then one of the many commercially available requirements-management, change-management, or defect-tracking tools can be used.

Figure 5 shows a use case and some of its slices captured on a set of sticky notes.

The use case shown is "7. Browse and Shop" from an online

FIGURE 5: **CAPTURING THE PROPERTIES OF A USE CASE AND ITS SLICES USING STICKY NOTES**



shopping application. Slices 1 and 2 of the use case are based on individual stories derived from the basic flow: "Select and Buy 1 Product" and "Select and Buy 100 Products." Slice 3 is based on multiple stories 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. Here a simple scheme of assessing relative size and complexity has been used.

The essential properties for a use-case slice are: (1) a list of its stories; (2) references to the use case and the flows that define the stories; (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 stories are used to name the slice, and the references to the use case are 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 toward the bottom right of each sticky note. In this case the team has played Planning Poker to create relative estimates using story points.

The use cases and the use-case slices should also be ordered so that the most important ones are addressed first.

Keeping work products as lightweight as appropriate

All of the work products are defined with a number of levels of detail. The first level defines the bare essentials, the minimal amount of information that is required for the

practice to work. Further levels of detail are defined to help the team cope with any special circumstances they might encounter. 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.

The good news is that you always start in the same way, with the bare essentials. The team can then continually adapt the level of detail in their use-case narratives to meet their emerging needs.

Things to do

Use-Case 2.0 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.

The *Find Actors and Use Cases* activity produces a use-case model that identifies the use cases, which will be subsequently *sliced*. These use-case slices will then be *prepared* by describing the related stories in the use-case narrative and defining the test cases. The slice is *analyzed* to work out how the system elements will interact to perform the use case, then *implemented* and *tested* as a slice. Use-Case 2.0 can be considered a form of test-driven development, as it creates the test cases for each slice upfront. Finally, the whole system is *tested* to ensure that all the slices work together when combined.

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, while continuing to *Find Actors and Use Cases*, you may also be implementing some of the slices from those use cases found earlier.

As the project progresses, priorities change, lessons are learned, 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 progress. This means there will be an ongoing *Inspect and Adapt* activity for the use cases. This will also adapt the way of working with the Use-Case 2.0 practice to adjust the size of slices or the level of details in work products to meet the varying demands of the project and team.

USING USE-CASE 2.0

Many people think that use cases are applicable only to user-intensive systems that have 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 applicable to all systems that are used—and that means *all* systems.

It's not just for user-intensive applications

In fact, use cases are just as useful for embedded

systems with little or no human interaction as they are for user-intensive ones. People are using use cases in the development of all kinds of embedded software in domains as diverse as motor, consumer electronics, military, aerospace, and medical industries. Even realtime 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.

It's not just for software development

The application of use cases is not limited to software development. They can also help understand business requirements, analyze existing business, design new and better business processes, and exploit the power of IT to transform 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, developers can seamlessly identify where the systems will impact the business and which systems are needed to support the business.

Handling all types of requirements

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

as part of the use case itself.

Some non-functional characteristics are subtler 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. These additional use cases are called *infrastructure use cases*,[8] as the requirements they contain will drive the creation of the infrastructure on which the application will run.

Applicable for all development approaches

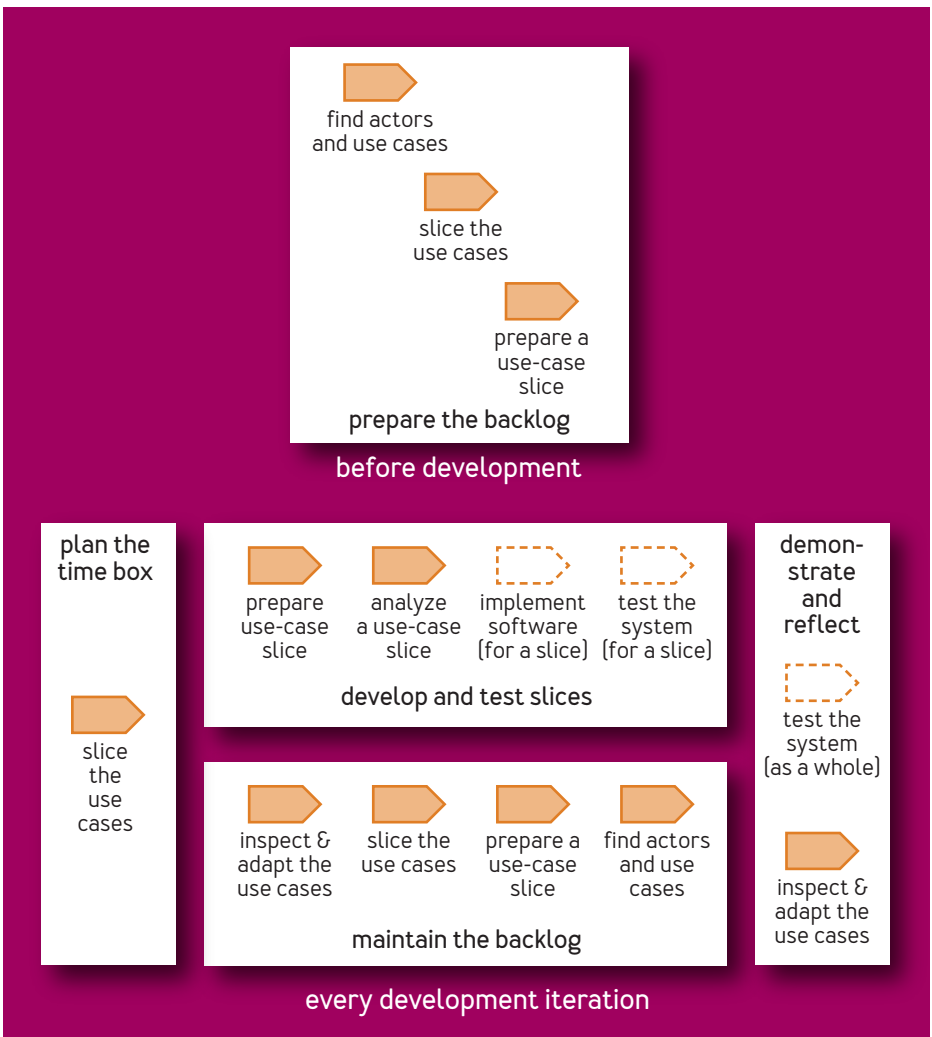
Use-Case 2.0 works with all popular software-development approaches, including:

- Backlog-driven iterative approaches such as Scrum, EssUP, and OpenUP.
- One-piece flow-based approaches such as Kanban.
- All-in-one-go approaches such as the traditional waterfall.

Use-Case 2.0 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 backlogs that teams use to drive their work, including product, release, 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.

FIGURE 6: **USE-CASE 2.0 ACTIVITIES FOR ITERATIVE DEVELOPMENT APPROACHES**



In Use-Case 2.0, the use-case slices are the primary backlog items. The use of use-case slices ensures that 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 them to be sliced as small as needed to support the development team.

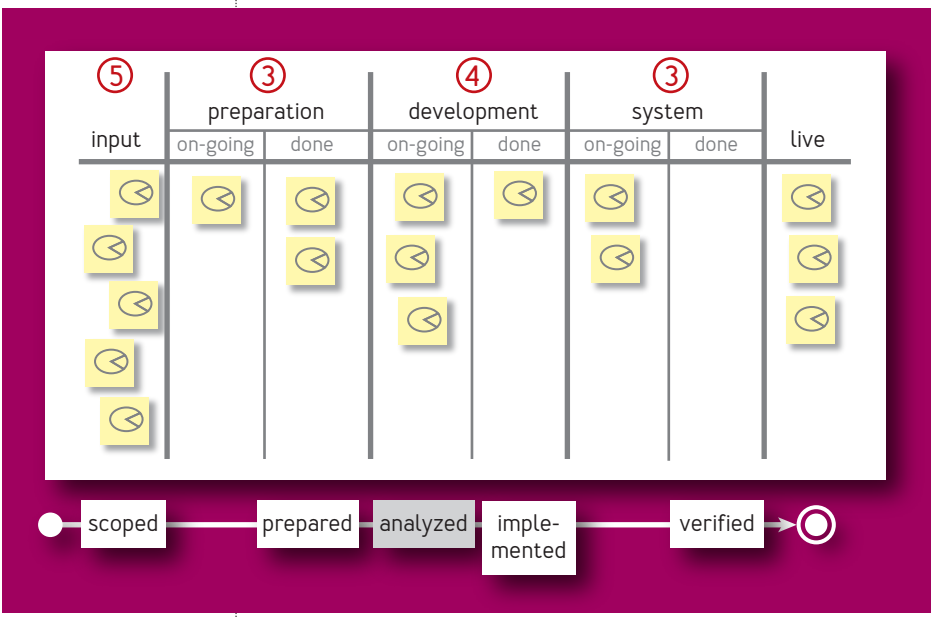
When a backlog-driven approach is adopted, it is important to realize that the backlog is not built and completed upfront but is continually worked on and refined. The typical sequence of activities for a backlog-driven, iterative approach is shown in figure 6.

Use-Case 2.0 and one-piece flow

One-piece flow is a technique taken from lean manufacturing that avoids the batching of the requirements seen in the iterative and waterfall approaches. Each requirements item flows quickly through the development process, but to work effectively, this technique needs small, regularly sized items. Use cases would be too irregularly sized and too big to flow through the system. Use-case slices, though, can be sized appropriately and tuned to meet the needs of the team.

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. Enough items need to be in the system to keep the team busy. Work-in-progress limits are used to level the flow and prevent any wasteful backlogs from building up. Figure 7

FIGURE 7: **USE-CASE SLICES ON A KANBAN BOARD**



shows a simple Kanban board for visualizing the flow of use-case slices.

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. In this figure the work-in-progress limit is five, and the customers, product owner, or requirements team that are the source of the requirements try to keep five use-case slices ready for implementation at all times.

Note that there is no definitive Kanban board or set of work-in-progress limits; it is dependent on team structure and working practices. The board and work-in-progress

limits should be tuned the same as practices. The states for the use-case slices are a great aid to this kind of work design as they can clearly define what state the slice should be in when it is to be handed to the next part of the chain.

Use-Case 2.0 and waterfall

For various reasons you may need to develop 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 upfront before they are handed over to a third party for development.

In a waterfall approach the use cases are not continually worked on and refined to allow the final system to emerge but are defined in one go at the start of the work.

The one-thing-at-a-time nature of the waterfall approach means that the makeup of the team is continually changing over time, 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 beyond the bare essentials.

Use-Case 2.0—It's not just for one type of team

Another important aspect of Use-Case 2.0 is its ability to adapt to existing team structures and job functions while encouraging teams to eliminate waste and increase efficiency. To this end, Use-Case 2.0 does not predefine 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 2.0 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 improve their performance continuously.

Scaling to meet your needs—in, out, and up

No one predefined approach fits everyone, so the use of Use-Case 2.0 needs to be scaled in a number of different dimensions:

- 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.
- They *scale out* to cover the entire lifecycle, covering not only analysis, design, coding, and test, but also operational usage and maintenance.
- 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 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 under

development, the development team always starts in the same way by identifying the most important use cases and creating a big picture summarizing what needs to be built. Use-Case 2.0 can then be adapted to meet the emerging needs of the team. In fact, the Use-Case 2.0 practice 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.

USER STORIES AND USE CASES—WHAT IS THE DIFFERENCE?

The best way to answer this question is to look at the common properties of user stories and use cases—the things that make both work well as backlog items and enable both to support popular agile approaches such as Scrum, Kanban, test-driven development, and specification by example.

Use-Case slices and user stories[3] share many common characteristics. For example:

- They both define slices of the functionality that teams can get done in a sprint.
- They can both be sliced up if they are too large, resulting in more, smaller items.
- They can both be written on index cards.
- They both result in test cases that represent the acceptance criteria.
- They are both placeholders for a conversation and benefit from the three Cs invented by Ron Jeffries: card, conversation, and confirmation.

- They can both be estimated with techniques such as Planning Poker.

So, given that they share so many things in common, what is it that makes them different? Use cases and use-case slices provide added value:

- A big picture to help people understand the extent of the system and its value.
- Increased understanding of what the system does and how it does it.
- Better organization, understanding, application, and maintenance of test assets.
- Easy test-case generation and analysis.
- Support for ongoing impact analysis.
- Active scope management allowing easy focus on providing the minimal viable product.
- Flexible, scalable documentation to help cope with traceability or other contractual constraints.
- Support for simple systems, complex systems, and systems of systems.
- Easier identification of missing and redundant functionality.

The question remains: which technique should you use, which, once you go beyond personal preferences, is very context dependent. Consider the following factors: how much access is there to the SMEs (subject matter experts); and how severe will requirements errors be if they escape to a live environment.

The sweet spot for user stories is achieved when there is easy access to a SME and the severity of errors is low. Use cases and use-case slices are more suitable when there is no

easy access to a SME or when error consequences are high. Since the use-case approach can scale down to the sweet spot of user stories, however, you may still want to apply them. If the subject system will always be in the sweet spot of user stories, then user stories are fine, but if you expect it to grow outside that area, you should consider use cases and use-case slices.

CONCLUSION

Use-Case 2.0 exists as a proven and well-defined practice that is 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 or lean fashion. It also has the completeness and rigor required to support teams that work in a more formal or waterfall environment.

More details about the fully documented Use-Case 2.0 practice are available at http://www.ivarjacobson.com.

References

1. Booch, G., Jacobson, I., Rumbaugh, J. 2004. *The Unified Modeling Language Reference Manual,* second edition. Addison-Wesley Professional.
2. Cockburn, A. 2001. *Writing Effective Use Cases.* Addison-Wesley Professional.
3. Cohn, M. 2004. *User Stories Applied.* Addison-Wesley Professional.

4. Constantine, L., Lockwood, L. 1999. *Software for Use*. Addison-Wesley Professional.
5. Jacobson, I. 2003. Case for aspects, part II. *Software Development Magazine* (November): 42-48.
6. Jacobson, I. 1987. Object-oriented software development in an industrial environment. In Conference Proceedings of Object-oriented Programming Systems, Languages, and Applications (OOPSLA 87).
7. Jacobson, I., Christerson, M., Johnsson, P., Overgaards, G. 1992. *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley Professional.
8. Jacobson, I., Ng, P.W. 2005. *Aspect-oriented Software Development with Use Cases*. Addison-Wesley Professional.
9. Slama, D., Puhlmann, F., Morrish, J., Bhatnagar, R. 2015. *Enterprise Internet of Things*; http://enterprise-Internet of Things.org/book/enterprise-Internet of Things/.

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Ivar Jacobson, *Ph.D, 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 method (or process) simply as a composition of practices. Jacobson is also one of the founders of the SEMAT (Software Engineering*

Method and Theory) community, the mission of which is to refound software engineering. He is the principal author of seven influential and best-selling books and a large number of papers. He was awarded the Gustaf Dalén medal ("the little Nobel Prize"), and he is an honorary doctor at San Martin de Porres University, Peru.

Ian Spence *is CTO at Ivar Jacobson International and the team leader for the development of the SEMAT (Software Engineering Method and Theory) kernel. An experienced coach, he has introduced hundreds of projects to iterative and agile practices. He has also led numerous successful large-scale transformation projects working with development organizations of up to 5,000 people. His current interests are agile for large projects, agile outsourcing, and driving sustainable change with agile measurements.*

Brian Kerr *is an experienced agile coach, consultant, and change agent, and is a principal consultant at Ivar Jacobson International. He works with teams and organizations, helping them adopt key software-development practices in a pragmatic and sustainable way. He has particular expertise in the requirements space and has used, taught, and consulted in the use-case approach for the past 20 years across many industries and domains. He has been involved in the thought work behind the SEMAT (Software Engineering Method and Theory) initiative and the latest ideas captured in the Use-Case 2.0 practice.*

Ten Cosmic Truths About Software Requirements

Karl Wiegers Follow
Aug 12, 2019 · 10 min read ★

I have been working in the field of software requirements and business analysis for about thirty years, as a practitioner, manager, consultant, trainer, and speaker. Having worked with more than 100 organizations of all sizes and types, I've observed some facts about requirements that appear to be universally applicable. This article (adapted from my book *More About Software Requirements*) presents some of these "cosmic truths" and their implications for the practicing business analyst.

Cosmic Truth #1: If you don't get the requirements right, it doesn't matter how well you execute the rest of the project.

Requirements serve as the foundation for all the project work that follows. By "requirements" I don't mean an initial specification you come up with early in the project, but rather the full set of requirements knowledge that is developed incrementally during the course of the project.

The purpose of a software development project is to build a product that provides value to a particular set of customers. Requirements development seeks to determine the mix of capabilities and characteristics in a solution that will best deliver this customer value. This understanding evolves over time as customers provide feedback on the early work and refine their expectations and needs. If a business analyst doesn't adequately explore these expectations and craft them into a set of product features and attributes, the chance of satisfying customer needs is slim.

One technique for validating requirements is to work with suitable customer representatives to develop user acceptance criteria or acceptance tests. These criteria define how customers determine whether they're willing to pay for the product or to begin using it to do their work. Acceptance tests aren't a substitute for thorough system testing, but they do provide a valuable perspective to determine whether the requirements are indeed right.

Cosmic Truth #2: Requirements development is a discovery and invention process, not just a collection process.

People often talk about "gathering requirements." This phrase suggests that the requirements are just lying around waiting to be picked like flowers or to be sucked out of the users' brains by the BA. I prefer the term *requirements elicitation* to *requirements gathering*.

Elicitation is an exploratory activity. It includes some discovery and some invention, along with recording the requirements information customer representatives present. Elicitation demands iteration. The participants in an elicitation discussion won't think of everything they'll need up front, and their thinking will change as the project continues.

A business analyst is an investigator, not simply a scribe who records what customers say. An adroit BA asks questions that stimulate the customers' thinking, uncover hidden information, and generate new ideas.

It's fine for a BA to propose requirements that might meet customer needs, provided the customers agree that those requirements add value. A BA might ask, "Would it be helpful if the system could do <whatever idea he has>?" The customer might reply, "Wow, that would be great! We didn't even think to ask for that feature, but it would save our users a lot of time." This creativity is part of the value the BA adds to the requirements conversation.

Cosmic Truth #3: Change happens.

It's inevitable that requirements will change. Business needs evolve, new users or markets are identified, business rules and government regulations are updated, and operating environments change over time. Requirements become clearer as the key stakeholders are prompted to think more carefully about what they really are trying to do with the product.

Some people fear a "change control process." The objective of such a process is not to inhibit change, but rather to ensure that the project incorporates the right changes for the right reasons. You need to anticipate and accommodate changes so as to produce the minimum disruption and cost to the project and its stakeholders. Excessive churning of the requirements after they've been agreed upon suggests that elicitation was incomplete or ineffective — or that agreement was premature.

To help manage change effectively, establish a rational and appropriate change control process. When I implemented such a process in a web development group once, the team members properly viewed it as a useful structure, not as a barrier. The group found this process invaluable for dealing with its mammoth backlog of change requests.

Nearly every software project becomes larger than originally anticipated, so expect your requirements to grow over time. A growth rate of several percent per month can significantly impact a long-term project. To accommodate some expected growth, build contingency buffers — also known as management reserve — into your project schedules. These buffers will keep your commitments from being trashed with the first change that comes along.

Instead of attempting to get all the requirements "right" up front and freeze them in a classic waterfall approach, baseline the first set of requirements based on what is

known at the time. A *baseline* is a statement about the requirements set at a specific point in time: “We believe these requirements will meet a defined set of customer needs and are a suitable foundation for proceeding with the next stage of design and construction.” Then implement the initial, top-priority set of requirements, get some customer feedback, and move on the next slice of functionality. This is the intent behind agile and other incremental approaches to software development.

Change is never free. Even the act of considering a proposed change and then rejecting it consumes effort. Software people need to educate their project stakeholders so they understand that, sure, we can make that change you just requested, and here’s what it’s going to cost. Then the stakeholders can make appropriate business decisions about which changes should be incorporated and when.

Cosmic Truth #4: The interests of all the project stakeholders intersect in the requirements process.

A *stakeholder* is an individual or group who is actively involved in the project, who is affected by the project, or who can influence its outcome. Figure 1 identifies some typical categories of software project stakeholders. Certain stakeholders are internal to the project team. others are outside the team but within the developing organization, and still other stakeholders are external to the organization.

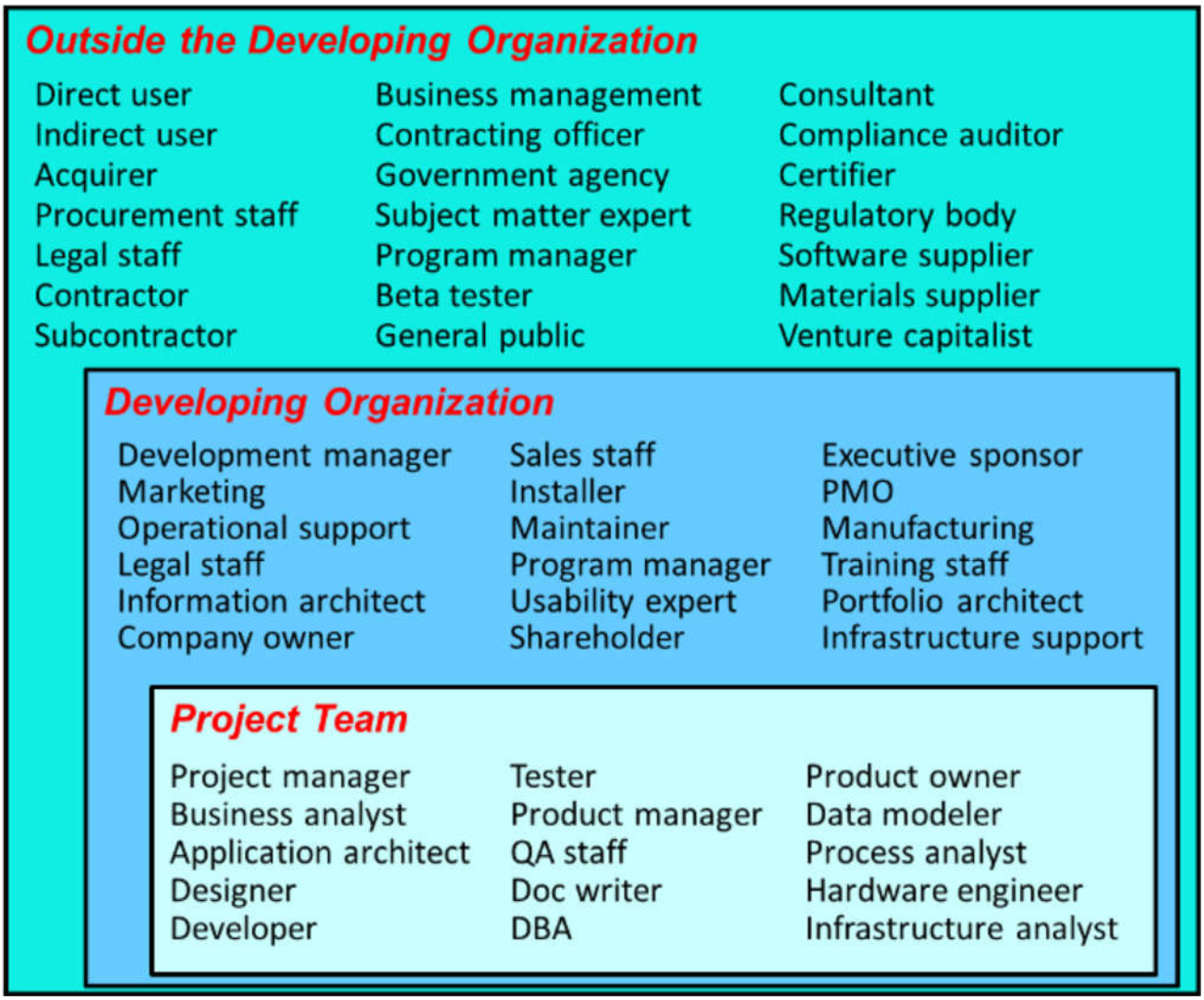


Figure 1. Some possible software project stakeholders.

The BA plays a vital communication role, interacting with all these stakeholders to specify a solution that will best satisfy all their needs, constraints, and interests. Identify your key stakeholder groups at the beginning of the project. Then determine which individuals can best represent the interests of each group. You can count on stakeholders having conflicting interests that must be reconciled. Therefore, identify the decision makers who will resolve these conflicts and have them agree on their decision-making process — before they confront their first significant decision.

Cosmic Truth #5: Customer involvement is the most critical contributor to software quality.

Inadequate customer involvement is a leading cause of software project failure. Customers often claim they can't spend time working on requirements. However, customers who aren't happy with the delivered product always find plenty of time to

point out the problems. You're going to get the customer input eventually. It's just a lot cheaper — and a lot less painful — to get that input early on, rather than after you've implemented the solution.

Customer involvement requires more than holding a workshop or two early in the project. Ongoing engagement by empowered and enthusiastic customer representatives is a critical success factor for software development. Following are some good practices for engaging customers in requirements development (see *Software Requirements, Third Edition* for more information about these practices):

Identify user classes. Customers are a subset of stakeholders, and users are a subset of customers. You can further subdivide your user community into multiple user classes that have largely distinct needs. Some might be favored user classes, whose needs are more strongly aligned with the project's business objectives. Unrepresented user classes are likely to be disappointed with the project outcome.

Select product champions. You need to determine who will serve as the literal voice of the customer for each user class. I call these people product champions. Ideally, product champions are actual members of the user class they represent. Sometimes, though, you might need surrogates to speak for certain user classes to the best of their ability. When developers are forced to define user requirements, they often don't do a great job.

Agree on customer rights and responsibilities. People who must work together rarely discuss just how they'll collaborate. The BA should negotiate with the customer representatives early in the project to agree on the responsibilities each party has to the requirements process.

Build prototypes. Prototypes let user representatives interact with a simulation or a portion of the ultimate system. Prototypes are far more tangible than written requirements specifications and easier for users to relate to. However, prototypes aren't a substitute for documenting the requirements details so everyone is working toward the same objective.

Cosmic Truth #6: The customer is not always right, but the customer always has a point.

We all know the customer is not always right. Sometimes customers are in a bad mood, uninformed, or unreasonable. If you receive conflicting input from multiple

customers, which one of those customers is “always right”? Following are some examples of situations in which a customer might not be right:

- Presenting solutions in the guise of requirements.
- Failing to prioritize requirements or expecting the loudest voice to get top priority.
- Not communicating business rules and other constraints, or trying to get around them.
- Expecting a new software system to drive business process changes.
- Not supplying appropriate representative users to participate in requirements elicitation.
- Failing to make timely decisions when a BA or developer needs an issue resolved.
- Not accepting the need for trade-offs in both functional and nonfunctional requirements.
- Demanding impossible commitments.
- Not accepting the cost of change.

The customer might not always be right, but the BA needs to understand and respect whatever point each customer is trying to make through his request for certain product features or attributes. Rather than simply promising anything a customer requests, strive to understand the rationale behind the customer’s thinking and conceive an acceptable solution.

Cosmic Truth #7: The first question to ask about a proposed new requirement is “Is this in scope?”

Anyone who’s been in IT for long has worked on a project that has suffered from scope creep. It is normal — and often beneficial — for requirements to grow over the course of a project. Scope creep, though, refers to an uncontrolled and continuous increase in requirements that makes it impossible to deliver a product on schedule.

To minimize scope creep, the project stakeholders must first agree on a scope

definition, a boundary between the desired capabilities that lie within the scope for a given product release and those that do not. Then, whenever some stakeholder proposes some new functionality the BA can ask, "Is this requirement in scope?" If the answer is "no," then either defer (or reject) the requirement or expand the project scope, with the associated implications of cost and schedule increase. A poorly defined scope boundary is an open invitation to scope creep.

Cosmic Truth #8: Even the best requirements document cannot replace human dialog.

Even an excellent written requirements specification won't contain every bit of information developers and testers need to do their jobs. BAs and developers will always need to talk with knowledgeable users and subject matter experts to refine details, clarify ambiguities, and fill in the blanks. This is the rationale behind having some key customers, such as product champions, work closely with the BA and developers throughout the project.

A set of written requirements is still valuable and necessary, though, whether stored in a document, a spreadsheet, a requirements management tool, or some other form. A documented record of what stakeholders agreed to at a point in time — a *group memory* — is more reliable than human memories, and it can be shared with people who weren't privy to the original discussions.

The requirements specifications need more detail if you won't have opportunities for frequent conversations with users and other decision makers. This happens when you're outsourcing the implementation of a requirements set your team created. Expect to spend considerable time on review cycles to clarify and agree on what the requirements mean in those situations.

Cosmic Truth #9: The requirements might be vague, but the product will be specific.

Specifying requirements precisely is hard! You're inventing something new, and no one's exactly sure what the product should be and do.

People sometimes are comfortable with vague requirements. Customers might like them because it means they can redefine those requirements later on to mean whatever they want them to mean. Developers sometimes favor vague requirements because they allow them to build whatever they want to build. This is all great fun,

but it doesn't help you create high-quality solutions.

Ultimately, you are building only one product, and someone needs to decide just what that product will be. Customers and BAs who don't make the decisions force developers to do so, though they likely know far less about the problem or the business. Precise requirements, however represented and communicated, lead to a better shared expectation of what you'll have at the end of the project.

Cosmic Truth #10: You're never going to have perfect requirements.

There's no way to know for certain that you haven't missed some requirement, and there will always be some requirements that aren't documented. Rather than declaring the requirements "done" at some point, define a baseline, a snapshot in

at all on their next project. This is an even more certain path to failure.

You don't succeed in business by striving for perfection. Practically speaking, strive to develop requirements that are *good enough* to allow the team to proceed with design, construction, and testing at an acceptable level of risk. The risk is the threat of having to do expensive and unnecessary rework. Keep this practical goal of "good enough" in mind as you pursue your quest for quality requirements.

If you're interested in requirements and business analysis, Process Impact provides numerous useful publications and other resources.

IVAR JACOBSON
INTERNATIONAL

USE-CASE 2.0

The Guide to Succeeding with Use Cases

Ivar Jacobson
Ian Spence
Kurt Bittner

December 2011

About this Guide

This guide describes how to apply use cases in an agile and scalable fashion. It builds on the current state of the art to present an evolution of the use-case technique that we call Use-Case 2.0. The goal is to provide you with a foundation to help you get the most out of your use cases; one that is not only applicable for small co-located agile teams but also large distributed teams, outsourcing, and complex multi-system developments.

It presents the essentials of use-case driven development as an accessible and re-usable practice. It also provides an introduction to the idea of use cases and their application. It is deliberately kept lightweight. It 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 2.0? – A one page introduction to the practice.
- First Principles – An introduction to use cases based around the 6 principles that act as the foundation for the practice.
- Use-Case 2.0 Content – The practice itself presented as a set of key concepts, activities, work products, and the rules that bind them together.
- Using Use-Case 2.0 – A summary of when and how to apply the practice.

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 2.0?", the "First Principles", and the "Using Use-Case 2.0" chapters to understand the basic concepts. You can then dip into the "Use-Case 2.0 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 2.0 Content" and "Using Use-Case 2.0" chapters once you've read the "What is Use-Case 2.0?" chapter. This will help you compare Use-Case 2.0 with your own experiences and understand what has changed.

Alternatively you could just read all the chapters in the order presented.

What is Use-Case 2.0?

Use Case: A use case is all the ways of using a system to achieve a particular goal for a particular user. Taken together the set of all the use cases gives you all of the useful ways to use the system, and illustrates the value that it will provide.

Use-Case 2.0: A scalable, agile practice that uses use cases to capture a set of requirements and drive the incremental development of a system to fulfill them.

Use-Case 2.0 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 2.0 is:

- Lightweight
- Scalable
- Versatile
- Easy to use

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. As mentioned before they can and should be used to drive the development, which means that Use-Case 2.0 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 2.0 exists as a proven and well-defined practice. Although the term Use-Case 2.0 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 20 years of experience from all over the world and all areas of software development.

First Principles

There are six basic principles at the heart of any successful application of use cases:

1. Keep it simple by telling stories
2. Understand the big picture
3. Focus on value
4. Build the system in slices
5. Deliver the system in increments
6. Adapt to meet the team's needs

In this chapter we look at these principles in more detail and use them to introduce the concepts of use-case modeling and use-case driven development.

Principle 1: Keep it simple by telling stories

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 2: Understand the big picture

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. 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).

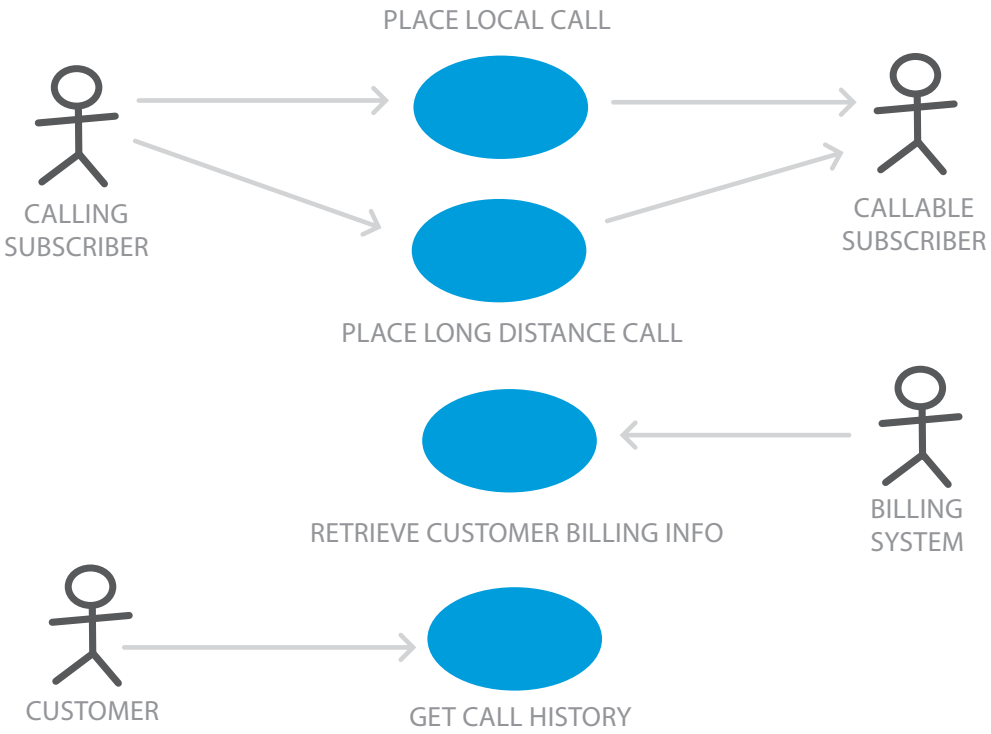


FIGURE 1: THE USE-CASE DIAGRAM FOR A SIMPLE TELEPHONE SYSTEM

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 1**, the actors are shown as stick-men 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

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; so it is much better to focus on how the system will be used than on 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 2 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 stories, and help us to find the test cases that complete their definition.

BASIC FLOW	ALTERNATIVE FLOWS
1. Insert Card	A1 Invalid Card
2. Validate Card	A2 Non-Standard Amount
3. Select Cash Withdrawal	A3 Receipt Required
4. Select Account	A4 Insufficient Funds in ATM
5. Confirm Availability of Funds	A5 Insufficient Funds in Acct
6. Return Card	A6 Would Cause Overdraft
7. Dispense Cash	A7 Card Stuck
	A8 Cash Left Behind
	etc..

FIGURE 2: THE STRUCTURE OF A USE-CASE NARRATIVE

Figure 2 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 of 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 Alternative Flows, but concentrate on the Basic Flow. You can then return to complete the alternative flows later as and when they are needed.

This kind of bulleted outline may be enough to capture the stories 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 stories 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: Build the system in slices

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 has to 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 2.0, where the use cases are sliced up to provide suitably sized work items, and where the system itself is evolved slice by slice.

Slicing up the use cases

The best way to find the right slices is to think about the stories and how we capture them. Each story is a good candidate slice. Each story 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 the stories real.

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 story described by 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.

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 your development.

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 3, 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.

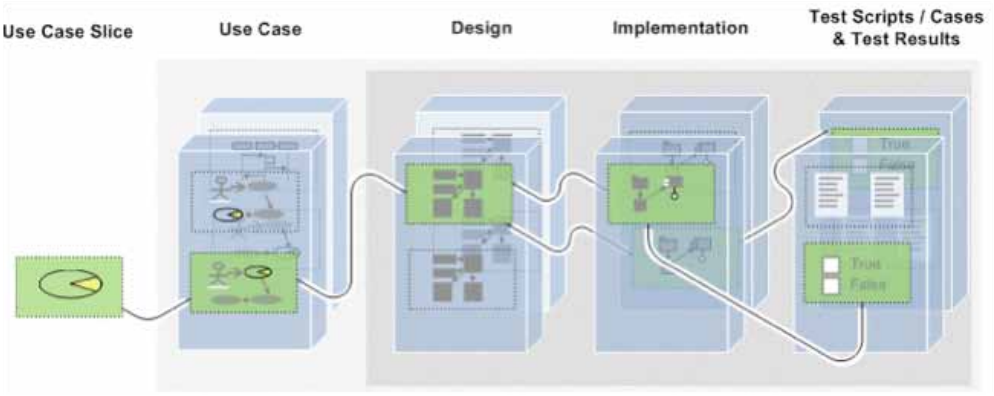


FIGURE 3: A USE-CASE SLICE IS MORE THAN JUST A SLICE OF THE REQUIREMENTS

On the left of Figure 3 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 actually 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 2.0 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 Slices: The most important part of Use-Case 2.0

The concept of a use-case slice is as integral to Use-Case 2.0 as the use case itself. It is the slices that enable the use cases to be broken down into appropriately sized pieces of work for the development team to tackle. 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.

Principle 5: Deliver the system in increments

Most software 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 4 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.

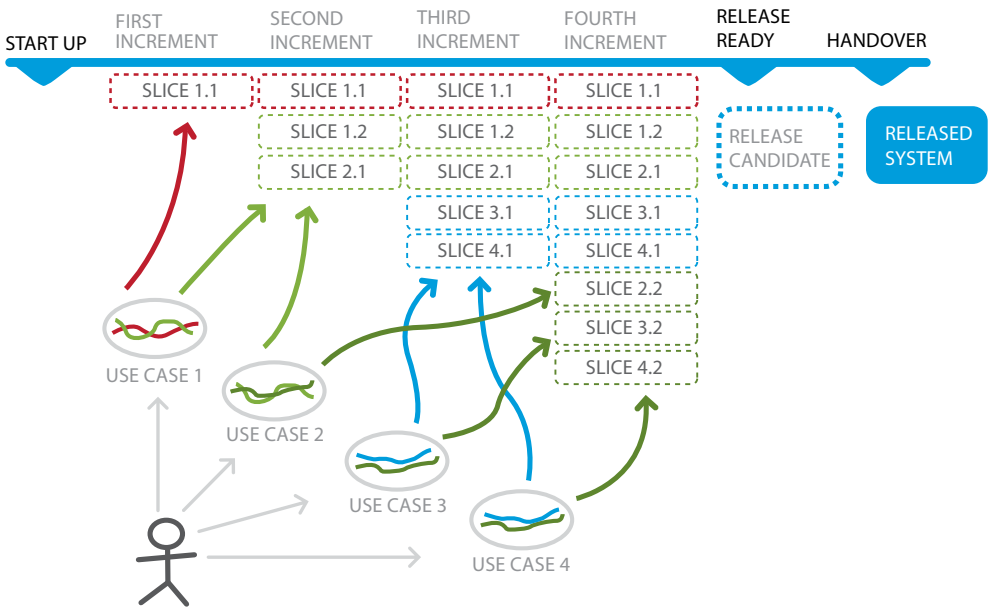


FIGURE 4: USE CASES, USE-CASE SLICES, INCREMENTS, AND RELEASES

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.

Principle 6: Adapt to meet the team's needs

Unfortunately there is no 'one size fits all' solution to the challenges of software 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 the requirements and drive the software 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 2.0 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 stories. These can be hand written 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 or not they need to go beyond the essentials, adding detail in a natural fashion as they encounter problems that the bare essentials cannot cope with.

Use-Case 2.0 Content

Use-Case 2.0 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 2.0 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 2.0 are the use case, the story 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 2.0.

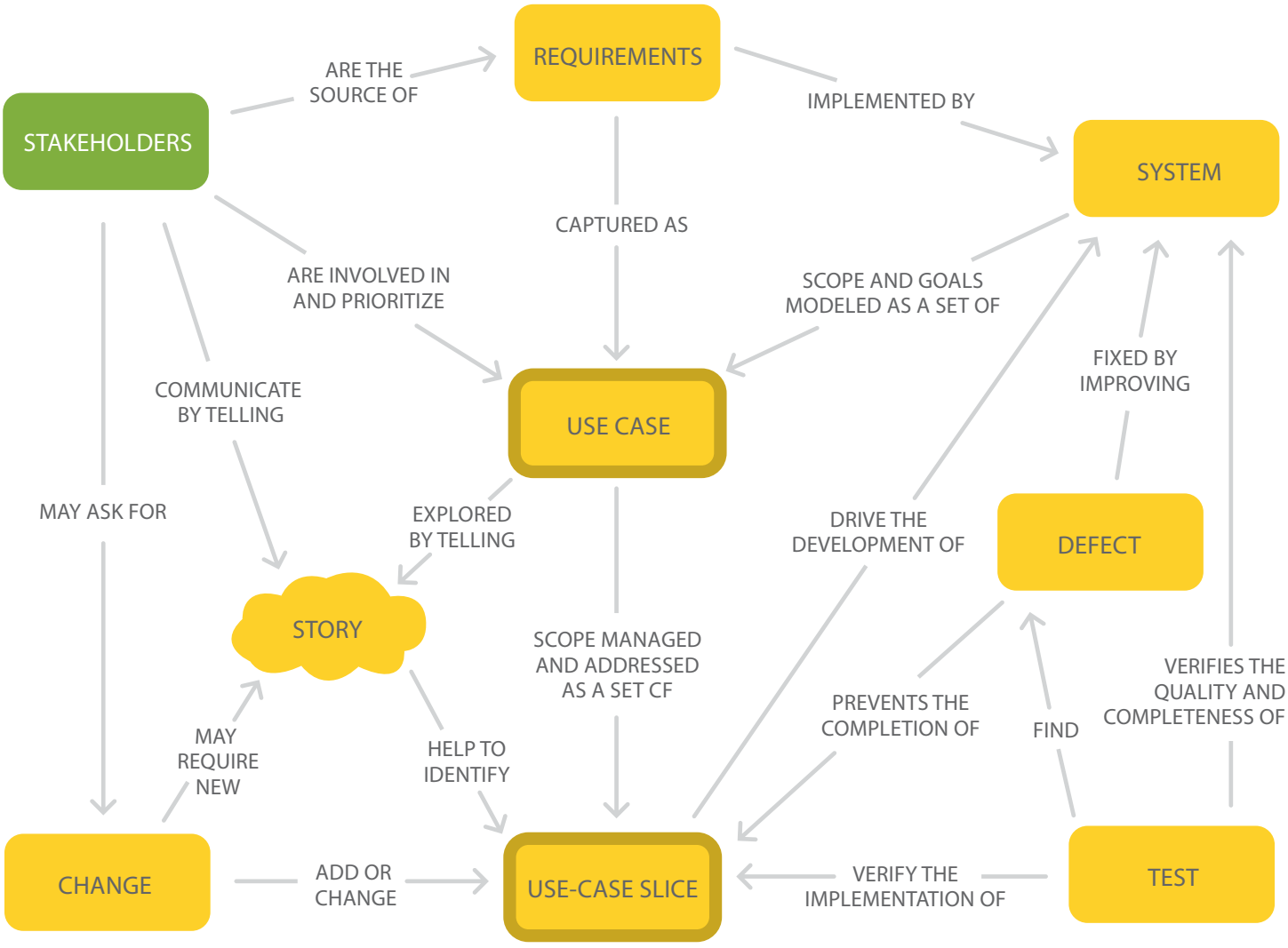


FIGURE 5: USE-CASE 2.0 CONCEPT MAP.

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 2.0 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 2.0 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 or not the implementation of the use-case slices has been a success. If defects are found during the testing then their presence will prevent the completion of the use-case slice until they have been fixed and the system improved.

Telling stories bridges 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. Understanding the stories is also the mechanism for finding the right use-case slices to drive the implementation of the system.

Use Cases

A use case is all the ways of using a system to achieve a particular goal for 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.
- That specific behavior of a system, which participates in a collaboration with a user to deliver something of value for that user.
- The smallest unit of activity that provides a meaningful result to the user.
- The context for a set of related requirements.

To understand a use case we tell stories. The stories 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 6** 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: when the goal of the use case has been established.

2) Story Structure Understood: when the structure of the use-case narrative has been understood enough for the team to start work identifying and implementing the first use-case slices.

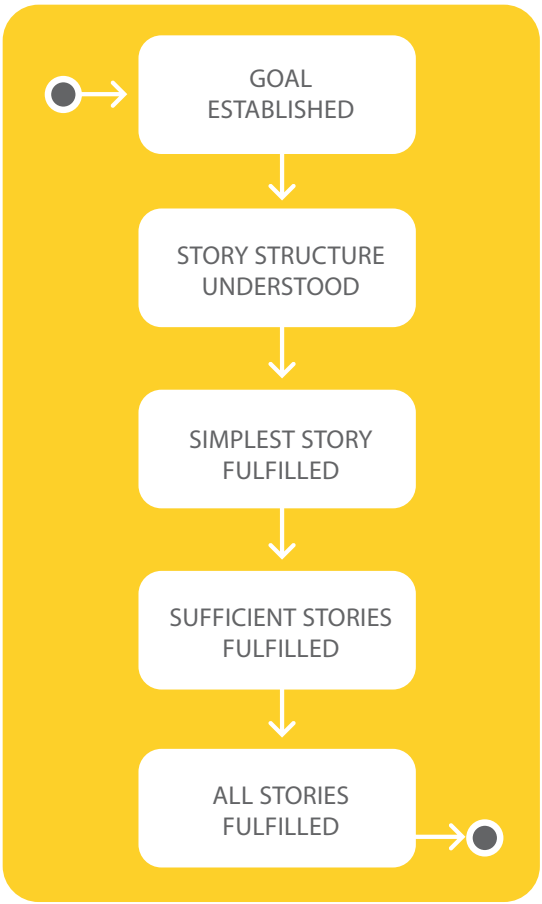


FIGURE 6: THE LIFECYCLE OF A USE CASE

3) Simplest Story Fulfilled: when the system fulfills the simplest story that allows a user to achieve the goal.

4) Sufficient Stories Fulfilled: when the system fulfills enough of the stories to provide a usable solution.

5) All Stories Fulfilled: when the system fulfills all the stories told by the use case.

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.

Use-Case Slices

Use cases cover many related stories of varying importance and priority. There are often too many stories 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 cases into smaller pieces that 1) allow us to select which pieces of the use case to deliver when, 2) provide a suitable unit for development and testing by the development team, and 3) allow us to have small and similarly sized pieces of work that flow quickly through development.

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. It acts as a placeholder for all the work required to complete the implementation of the selected stories. 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 2.0, as it is not only used to help with the requirements but to drive the development of a system to fulfill them.

Use-case slices:

- Enable use cases to be broken up into smaller, independently deliverable units of work.
- Enable the requirements contained in a set of use cases to be ordered, prioritized and addressed in parallel.
- Link the different system models (requirements, analysis, design, implementation and test) used in use-case driven development.

As **Figure 7** 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) **Scoped:** when it has been scoped and the extent of the stories covered has been clarified.

2) **Prepared:** when the slice has been prepared by enhancing the narrative and test cases to clearly define what it means to successfully implement the slice.

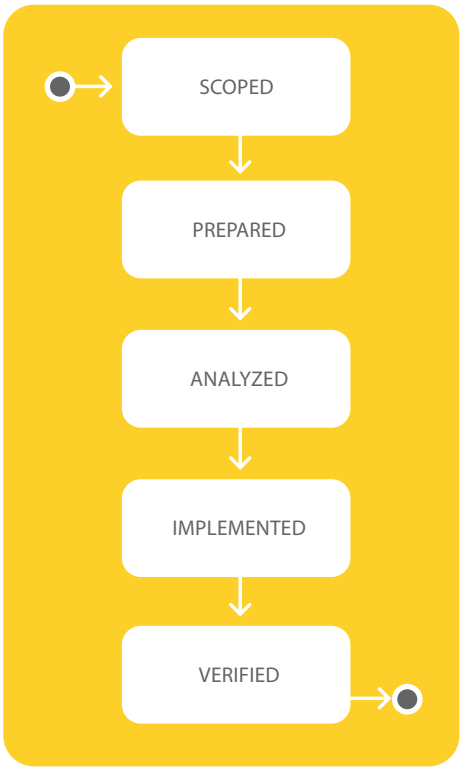


FIGURE 7:
THE LIFECYCLE OF A USE-CASE SLICE

3) **Analyzed:** when the slice has been analyzed so its impact on the components of the system is understood and the pieces affected are ready for coding and developer testing.

4) **Implemented:** when the software system has been enhanced to implement the slice and the slice is ready for testing.

5) **Verified:** and finally when the slice has been verified as done and is ready for inclusion in a release.

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: scoping>preparation>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 implemented, and a fourth being analyzed. In the next chapter we will explore this more when we look at using Use-Case 2.0 with different development approaches.

Stories

Telling stories is how we explore the use cases with our stakeholders. Each story of value to the users and other stakeholders is a thread through one of the use cases. The stories can be functional or non-functional in nature.

A story is described by part of the use-case narrative, one or more flows and special requirements, and one or more test cases. The key to finding effective stories is to understand the structure of the use-case narrative. The network of flows can be thought of as a map that summarizes all the stories needed to describe the use case. **Figure 8** illustrates the relationship between the flows of a use-case narrative and the stories it describes.

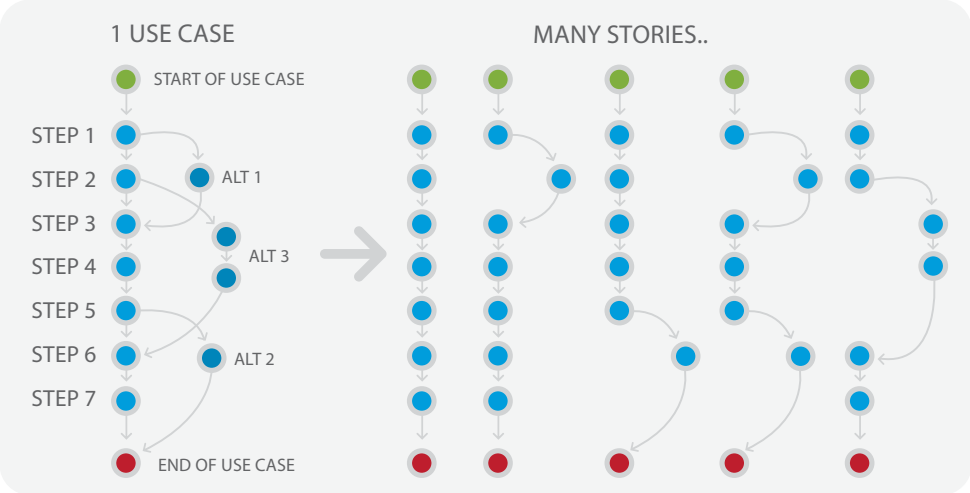


FIGURE 8:
THE RELATIONSHIP BETWEEN THE FLOWS AND THE STORIES

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. On the right of the diagram some of the stories covered by the flows are shown. Each story traverses one or more flows starting with the use case at the start of the basic flow and terminating with the use case at the end of the basic flow. This ensures that all the stories 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 story there will be one or more test cases.

There are two common approaches to identifying the stories and creating the use-case narrative:

- **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 narrative and allows them to identify their stories.

- **Bottom Up:** Using the bottom up approach we start by brainstorming some stories and then grouping these by theme to identify our use cases. The set of stories 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 missing stories and make sure that all the stories 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 stories.

The stories are a useful thinking tool to help us find the right use-case slices and, most importantly, the right test cases. We don't need to track the state of the stories themselves as it is the execution of the test cases that shows us how complete the system is, and the progress of the use cases and use-case slices that drive the development.

Defects and Changes

Although not a direct part of Use-Case 2.0, 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 story that has not been allocated to a slice and prepared, analyzed, 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, document them. Figure 9 shows the five Use-Case 2.0 work products (in blue) and their relationships to the requirements, use cases, use-case slices, stories, tests, and the system (in yellow).

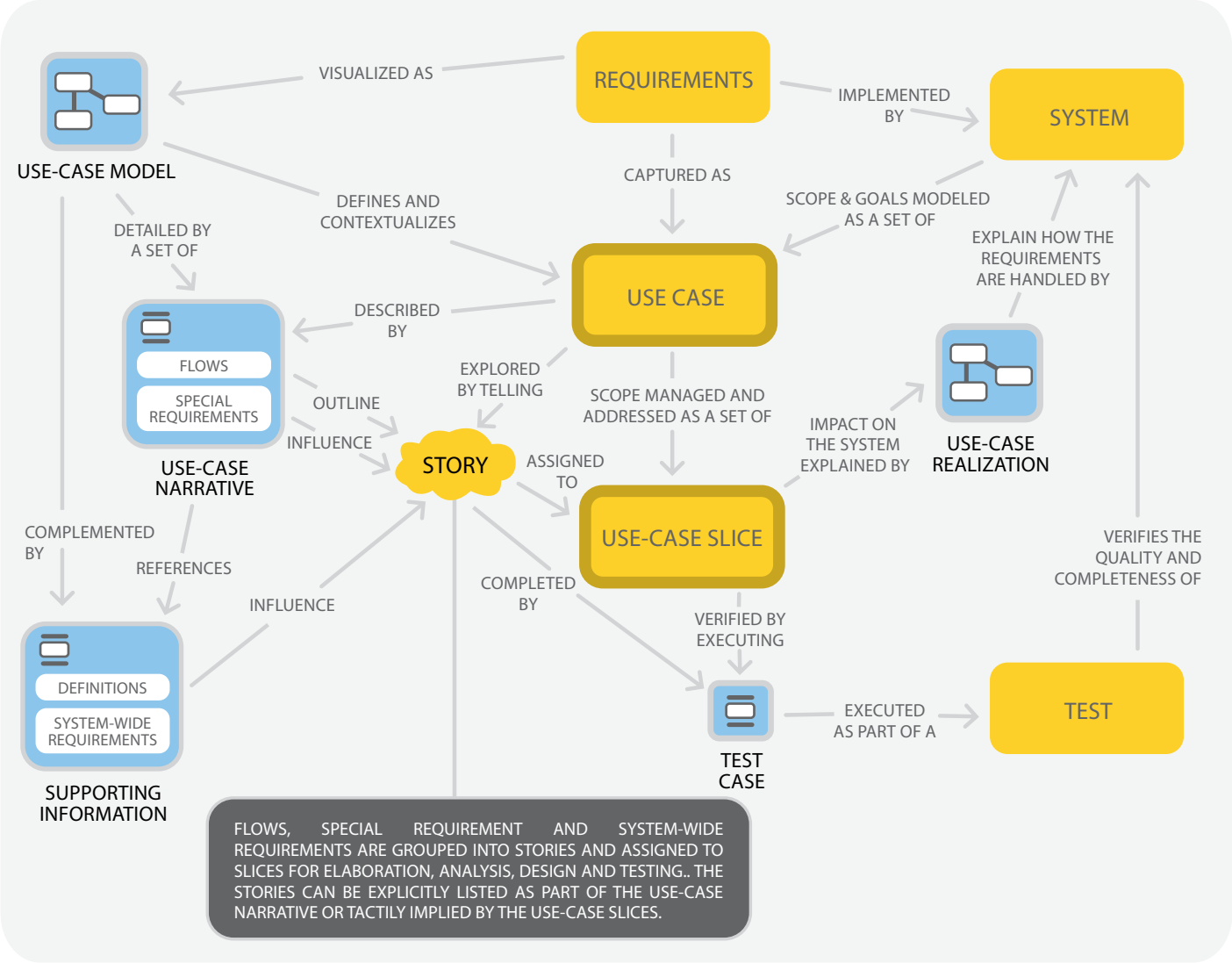


FIGURE 9: THE USE-CASE 2.0 WORK PRODUCTS

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 telling stories. Each use case is described by 1) a use-case narrative that outlines its stories and 2) a set of test cases that complete the stories. The stories are described as a set of flows. These can be complemented with a set of special requirements that will influence the stories, help you assign the right stories to the use-case slices for implementation, and most importantly define the right test cases.

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 stories in the use-case narratives. It also captures any system-wide requirements that apply to all of the use cases. Again these will influence the stories selected from the use cases and 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 stories and the use-case slices. These are not needed as they are fully documented by the other work products. If required you can list the stories 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 requirements management, change management or defect tracking tools could be used.

Figure 10 shows a use case and some of its slices captured on a set of post-it notes.

FIGURE 10: CAPTURING THE PROPERTIES OF A USE CASE AND ITS SLICES USING POST-IT NOTES

The use case shown is use-case '7 Browse and Shop' from an on-line shopping application. Slices 1 and 2 of the use case are based on individual stories derived from the basic flow: 'Select and Buy 1 Product' and 'Select and Buy 100 Products'. Slice 3 is based on multiple stories covering the availability of the various support systems involved in the use case. These stories cover a number of the alternative flows.

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 list of its stories, 2) references to the use case and the flows that define the stories, 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 stories are used to name the slice and the references to the use case are 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 post-it 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 11** shows how these post-its 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 11: USING USE CASES AND USE-CASE SLICES TO BUILD A PRODUCT BACKLOG

Figure 11 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 post-it notes falling off the whiteboard. These teams often track the state of their use cases and use-case slices using a simple spreadsheet including work-sheets such as those shown in Figure 12 and Figure 13.

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 12: 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 13: 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 of the work products are defined with a number of levels of detail. The first level of detail defines the bare 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 or not they need to go beyond the essentials, adding detail in a natural fashion as they encounter problems that the bare essentials cannot cope with.

Figure 14 shows the levels of detail defined for the set of Use-Case 2.0 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.

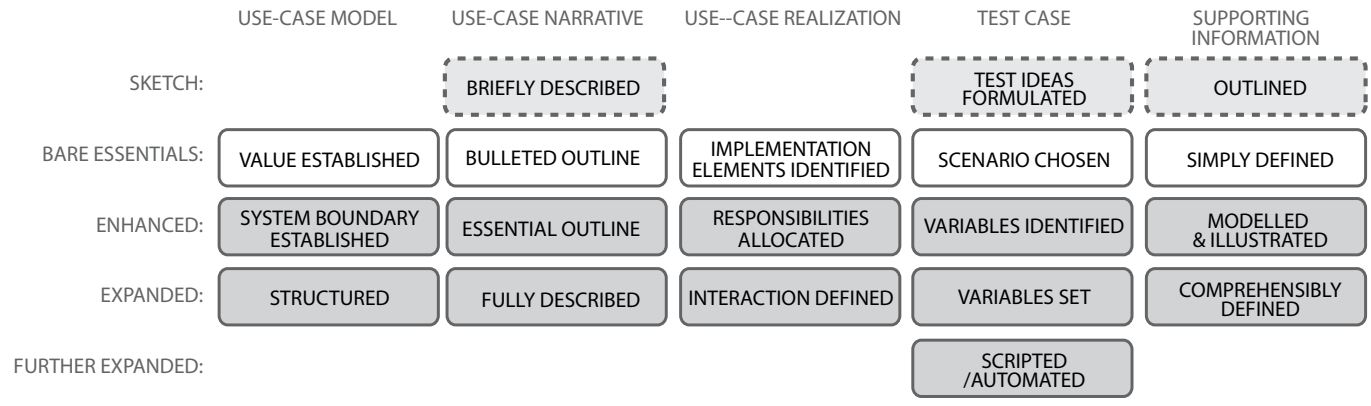


FIGURE 14: WORK PRODUCT LEVELS OF DETAIL

The good news is that you always start in the same way, with the bare 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 2.0 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 13** 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 the practice. The dashed chevrons are placeholders for other activities that the practice depends on to be successful. Use-Case 2.0 does not care how these activities are performed, it just needs them to be done.

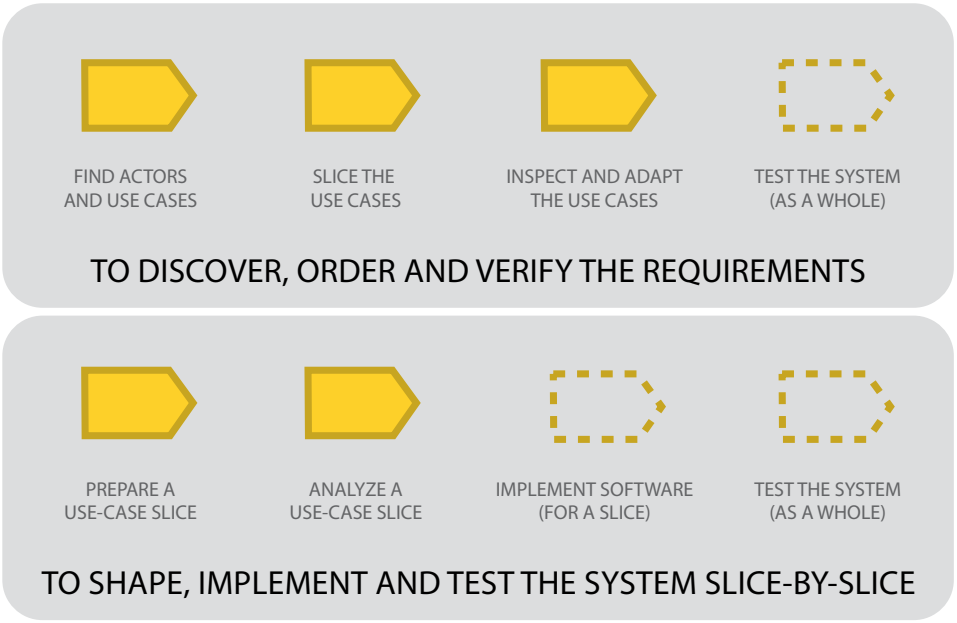


FIGURE 15: THE ACTIVITIES IN USE-CASE 2.0

Read **Figure 15** 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 provides an introduction to 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 stories. If a use case isn't needed then there is no need to discuss or document its stories. 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 post-it-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 it very easy to quantify, outline, and order.

Slice the Use Cases

Next you need to create your first use-case slices. You do this to:

- 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 multiple stories. You need to slice the use cases to select the stories 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 identify enough slices to meet the immediate needs of the team.

You don't need to completely slice up the use cases, just pull out the slices that are needed to progress the work and leave the rest of the stories in the use cases for slicing when and if they are 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 ordered 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 should always be based on the basic flow. For some complex systems this slice may not even cover the whole of the flow. You may just take a sub-set of the basic flow, skipping the detail of some steps snd stubbing up the solution for others, to attack the biggest challenges in implementing the use case and learn whether or not it can be implemented.

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.

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 stories, 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. There may still be work to do if the test cases are to be automated but there will be no doubt about what needs to be achieved by a successful implementation.

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.

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

Analyze a Use-Case Slice

Before you 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.

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.

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

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 slices. 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 2.0 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 up-front 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 of 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 2.0 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 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 project progresses it is essential that you continually work with 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 2.0 and adjust the size of the slices or the level of detail in your use-case narratives, supporting definitions 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 (scoped, prepared, analyzed, 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 project progresses you should monitor and adjust the slice size to eliminate waste and improve the team's effectiveness.

Using Use-Case 2.0

Use-Case 2.0 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 approaches.

Use-Case 2.0: 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 2.0: It's not just for user-intensive applications
In fact 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 other systems. Their applicability is far broader than most people think.

Use-Case 2.0: 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 2.0: Handling all types of requirement

Although they are one of the most popular techniques for describing 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 of 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 2.0: Applicable for all development approaches

Use-Case 2.0 works with all popular software development approaches including:

- Backlog-driven iterative approaches such as Scrum, EssUP and OpenUP
- One-piece flow based approaches such as Kanban
- All-in-one go approaches such as the traditional Waterfall

In the following 3 short sections we will illustrate how Use-Case 2.0 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 2.0 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, and project backlogs. Regardless of the terminology used they all follow the same principles. The backlog itself is an ordered list 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 16.**

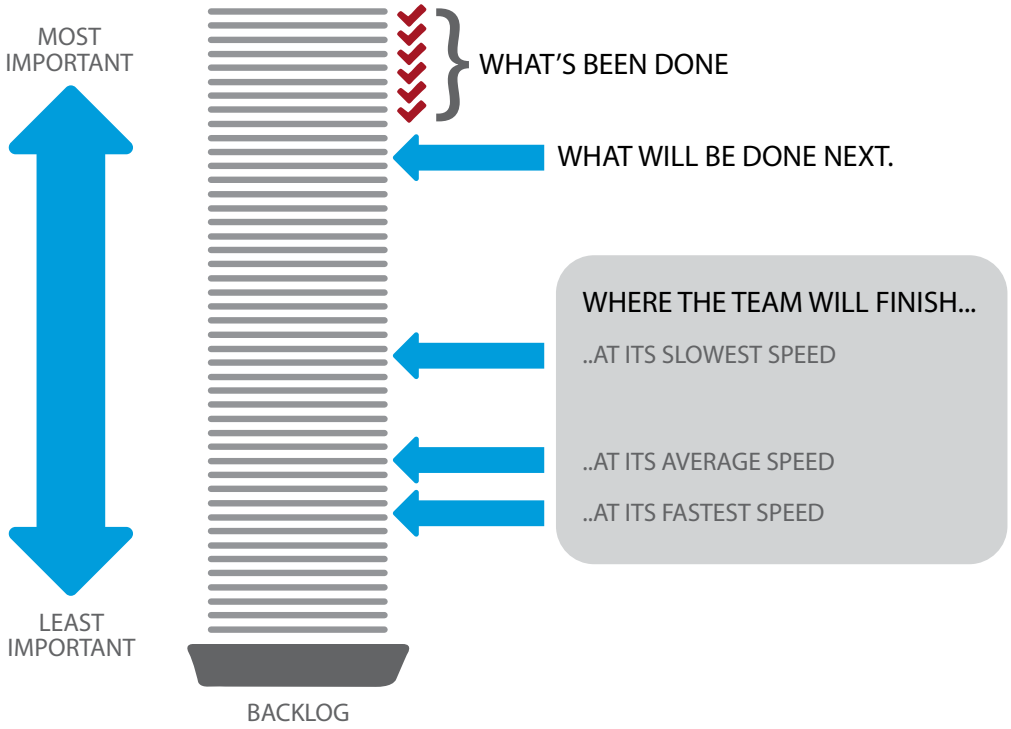


FIGURE 16: A BASIC BACKLOG

When you use Use-Case 2.0 the use-case slices are the primary backlog items. 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 grooming or maintaining the backlog. The typical sequence of activities for a backlog-driven, iterative approach is shown in **Figure 17.**

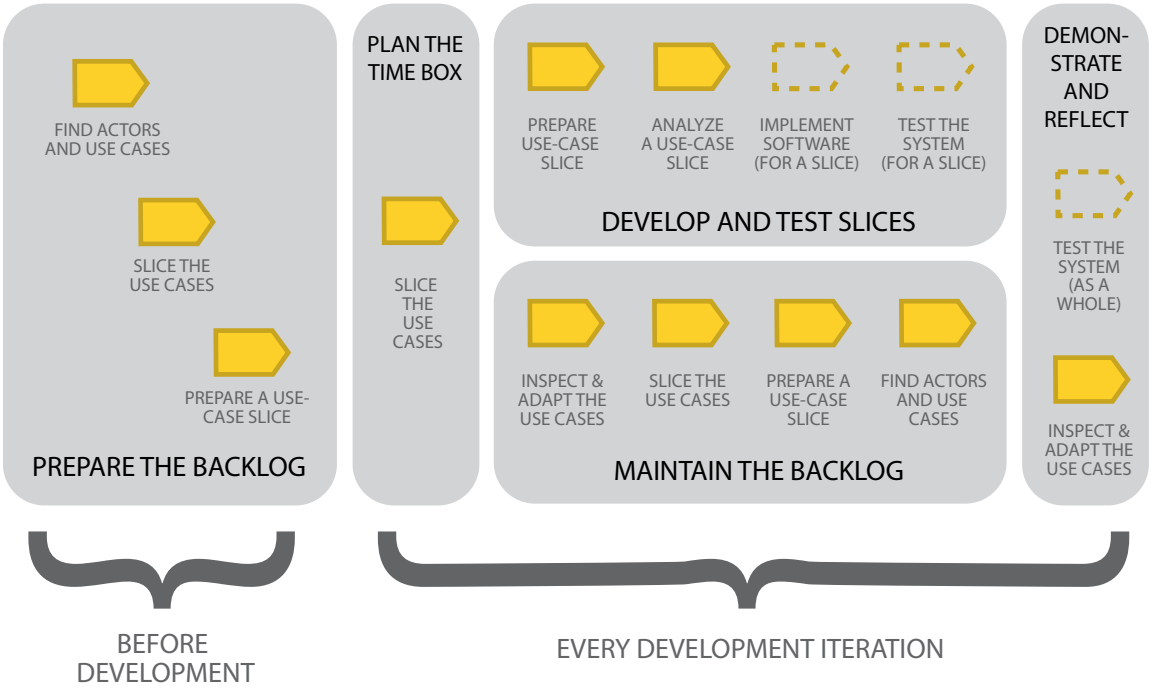


FIGURE 17: USE-CASE 2.0 ACTIVITIES FOR ITERATIVE DEVELOPMENT APPROACHES

Before the development starts the backlog is prepared; 'Find Actors and Use Cases' is used to build the initial use-case model and scope the system, 'Slice the Use Cases' is used to create the initial set of most important use-case slices to seed the backlog, and 'Prepare Use-Case Slice' is used to get one or more of these ready for development in the first iteration.

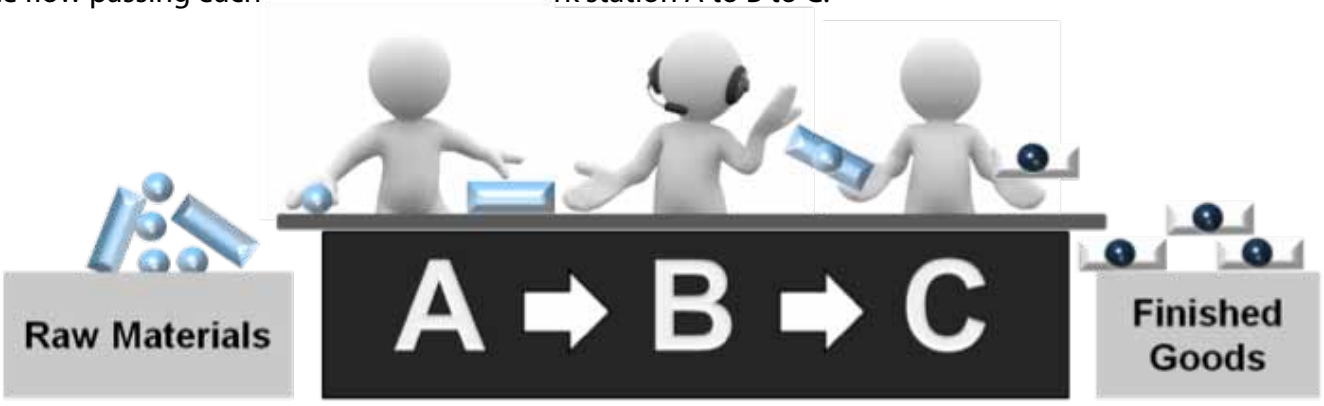
Once the backlog is up and running you can start the first development iteration. Every iteration starts with some planning. During this planning you may need to use 'Slice the Use Cases' to further slice the selected use-case slices to make sure they are small enough to complete in the iteration. The development team then uses 'Prepare a Use-Case Slice', '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', 'Slice the Use Cases' 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 2.0 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.

Use-Case 2.0 and one-piece flow

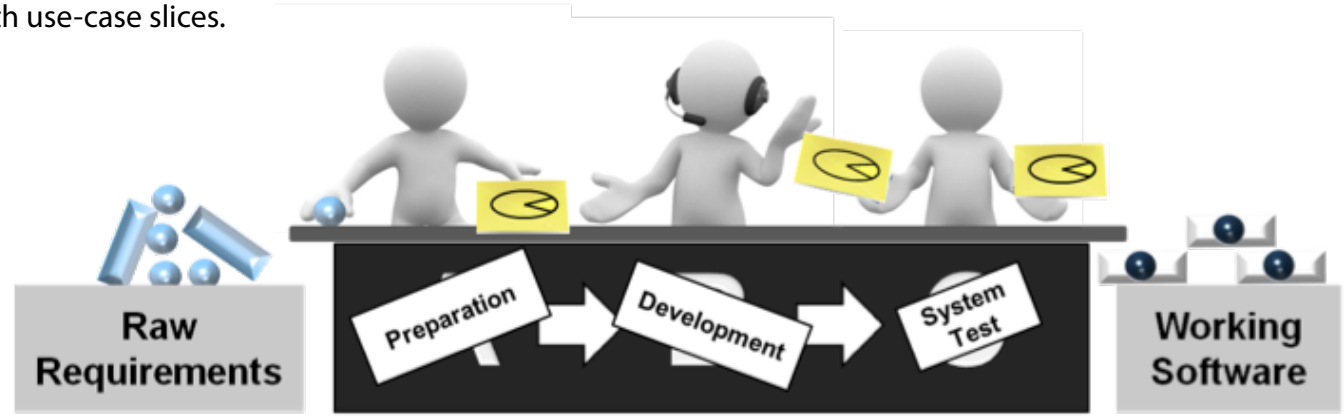
One-piece flow is an approach that avoids the batching of the requirements seen in the iterative and waterfall approaches. In a one-piece flow approach each requirements item flows through the development process. One-piece flow is a technique taken from lean manufacturing. **Figure 18** shows a small team engaging in one-piece flow passing each item directly from work station A to B to C.



People icons © ioannis kounadeas / Fotolia

FIGURE 18: 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 19** illustrates one-piece flow for software development with use-case slices.



People icons © ioannis kounadeas / Fotolia

FIGURE 19: 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 work station 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 work station. 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 analysis 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 work station, or even only one instance of each work station.

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 20** 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.

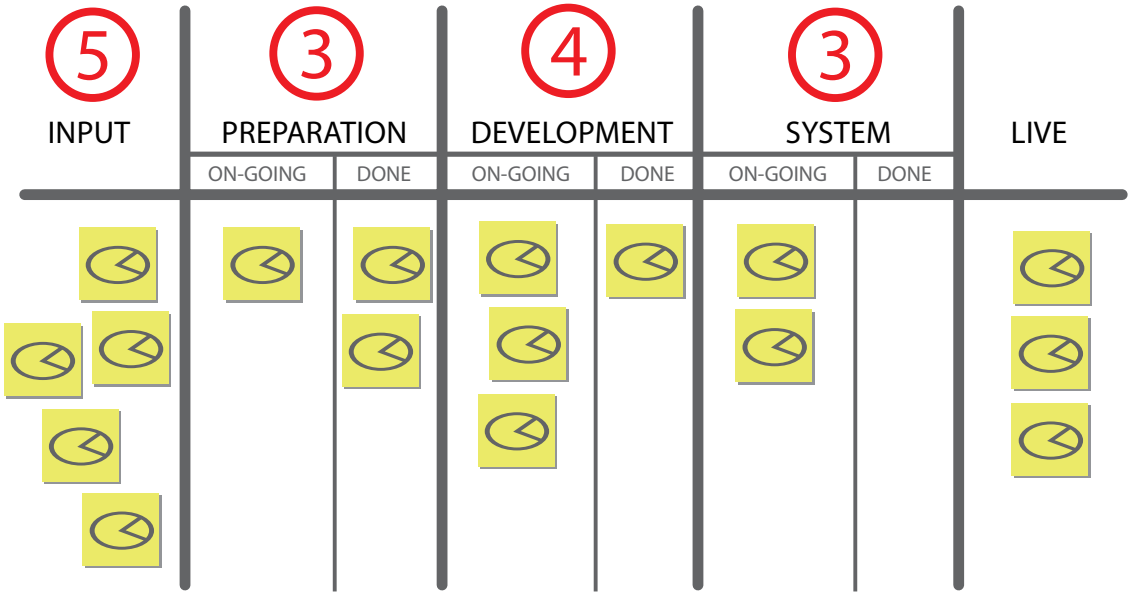


FIGURE 20: USE-CASE SLICES ON A KANBAN BOARD

The work in progress limits are shown in red. Reading from left to right you can see that slices have to be indentified 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, stories 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 21** shows the alignment between the states and the Kanban board shown in Figure 20. 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.

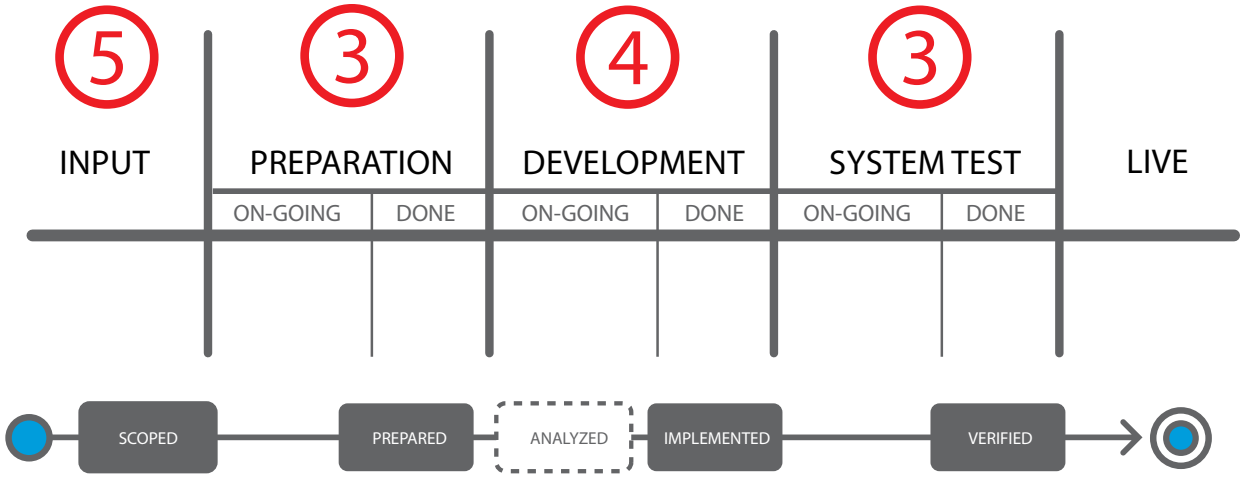


FIGURE 21: ALIGNING THE STATES OF THE USE-CASE SLICE TO THE KANBAN BOARD

Figure 22 shows where the different Use-Case 2.0 activities are applied. The interesting thing here is that "Inspect and Adapt Use Cases" is not the responsibility of any particular work station 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 work stations they have as well as their work in progress limits.

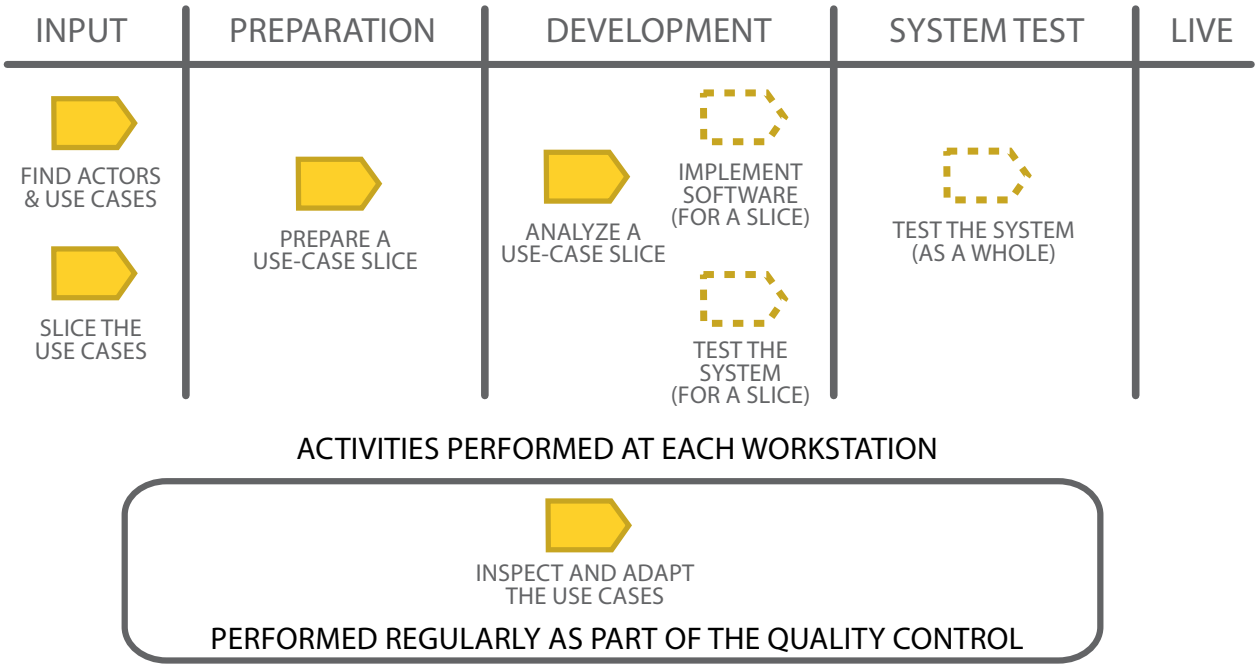


FIGURE 22: USE-CASE 2.0 ACTIVITIES FOR WATERFALL APPROACHES

For example as a result of reviewing the team's effectiveness you might decide to eliminate the preparation work station 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 work station to have finished their work resulting in the Kanban board shown in **Figure 23.**

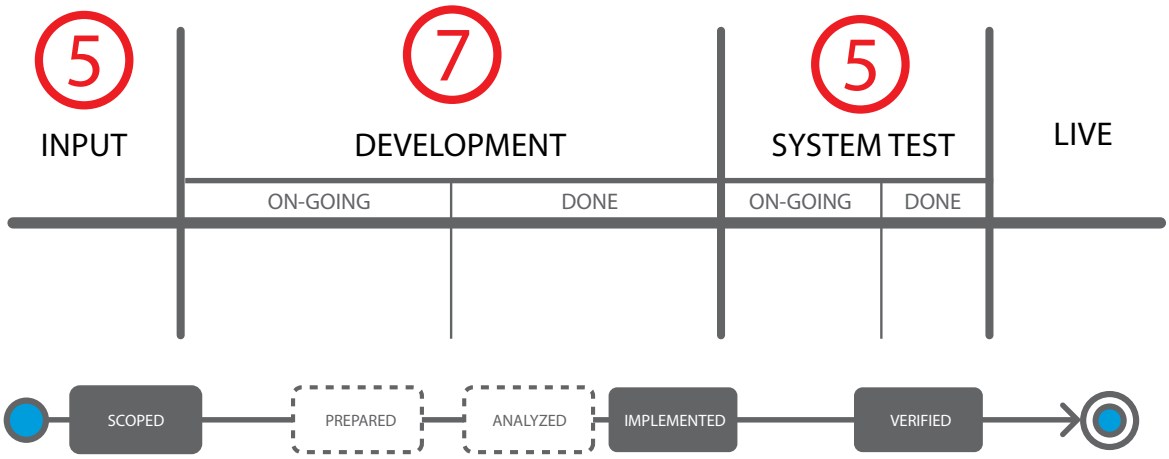


FIGURE 23: THE TEAM'S REVISED KANBAN BOARD SHOWING COMPLETION STATES

Use-Case 2.0 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 24.**

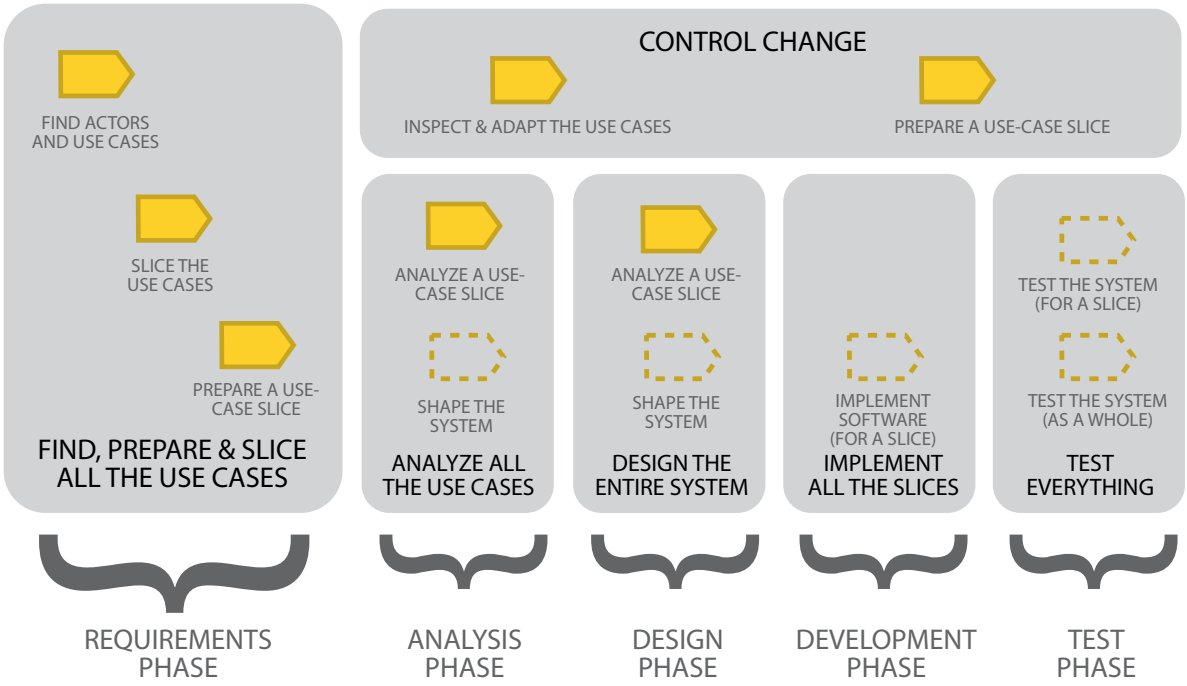


FIGURE 24: USE-CASE 2.0 ACTIVITIES FOR WATERFALL APPROACHES

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 thing 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 to 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 25 shows the level of detail typically used on waterfall projects.

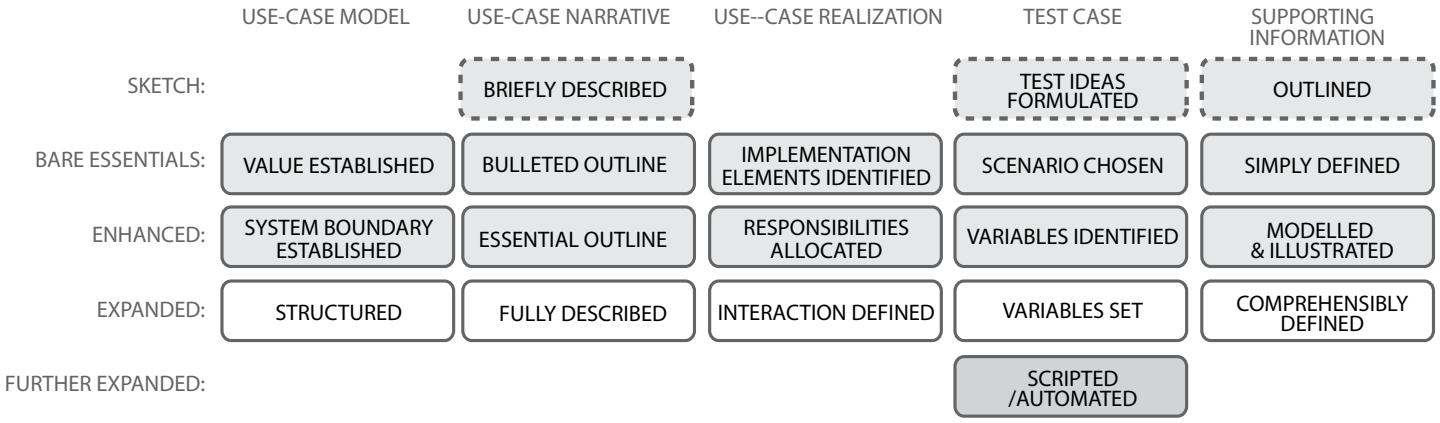


FIGURE 25: LEVELS OF DETAIL FOR THE WORK PRODUCTS USING A WATERFALL APPROACH

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 of 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 of the stories 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 2.0 – It's not just for one type of team

Another important aspect of Use-Case 2.0 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 2.0 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 2.0 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 2.0: 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 2.0 in a number of 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 2.0 to meet the emerging needs of the team. In fact Use-Case 2.0 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 2.0 exists as a proven and well-defined practice, one that is 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 2.0 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 2.0 is free and offered to the public in this guide. Use-Case 2.0's 'things to work with' and 'things to do' are non-negotiable, and although it is possible to only implement parts of Use-Case 2.0 the results are indeterminate and the practice used will not be Use-Case 2.0.

This guide is deliberately kept lightweight and may not be sufficient for you to adopt the practice. Additional information is available in the form of a fully documented, web-published practice from www.ivarjacobson.com. This is offered as a stand-alone web-site or a plug in for EssWork.

This is the first of many publications on Use-Case 2.0, 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 2.0.

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 stories 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:

Initiated: An intermediate 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 relation ships, 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 an essential complement to the use-case model itself. Without the supporting information it will be impossible 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 which slice of the requirements it applies to.

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 ex ecute. 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 sub-set 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 of 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 through-out the life-time 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 stories that the system must support, and for each story 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 of 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 flip-chart, 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 secondary actors and 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 of 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:

- Outline the stories 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 different 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 simple 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 outsourcing 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 data-base tables, collaborate together 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, story-boards, 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 par allel, or in support of multiple slices, the developers need to understand the responsibilities of the in dividual 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 inter faces 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 behaviour 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 which 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.
Story: A way of using a system that is of value to a user or other stakeholder. In Use-Case 2.0 a story is described by part of the use-case narrative, one or more flows and special requirements, and one or more test cases.
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 2.0: A scalable, agile practice 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.

Acknowledgements

General

Use-Case 2.0 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 everyday on their projects and in particular those who share their experiences in-side and out-side 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 short 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 e-book including, in no particular order, Paul MacMahon, Richard Schaff , Eric Lopes Cardozo, Svante Lidman, Craig Lucia, Tony Ludwig, Ron Garton, Burkhard Perkens-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

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

Ian is Chief Technology Officer at Ivar Jacobson International where he specializes in the agile application of the Unified Process. He is a certified RUP practitioner, ScrumMaster and an experience coach having worked with 100s of projects to introduce iterative and agile techniques. He has over 20 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 contributes to the technical direction of Ivar Jacobson International and works with the company's Technology Office to define the next generation of smart, active, software development practices. He is 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".

Kurt Bittner

Kurt is Chief Technology Officer of Ivar Jacobson International, Americas. He has worked in the software industry for more than 25 years in a variety of roles including developer, team leader, architect, project manager and business leader. He has led agile projects, run a large division of a software development company, survived and thrived in several start-ups, run an acquisition, and worked with clients in a variety of industries including aerospace, finance, energy and electronics. He was a key contributor to the early development of the Rational Unified Process as well as, more recently, IBM's Jazz project. His experience includes significant work in banking and finance, relational database system design and architecting, and consulting and mentoring a wide variety of clients on software development improvement strategies and approaches. He is the co-author of "Use Case Modeling", "Managing Iterative Software Development Projects" and "The Economics of Iterative Software Development".

About Ivar Jacobson International

IIJI 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 calibre 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

Sweden
+46 8 515 10 174
info-se@ivarjacobson.com

The Netherlands
+31(0) 20 654 1878
info-nl@ivarjacobson.com

UK
+44 (0)20 7025 8070
info-uk@ivarjacobson.com

Asia
+8610 82486030
info-asia@ivarjacobson.com

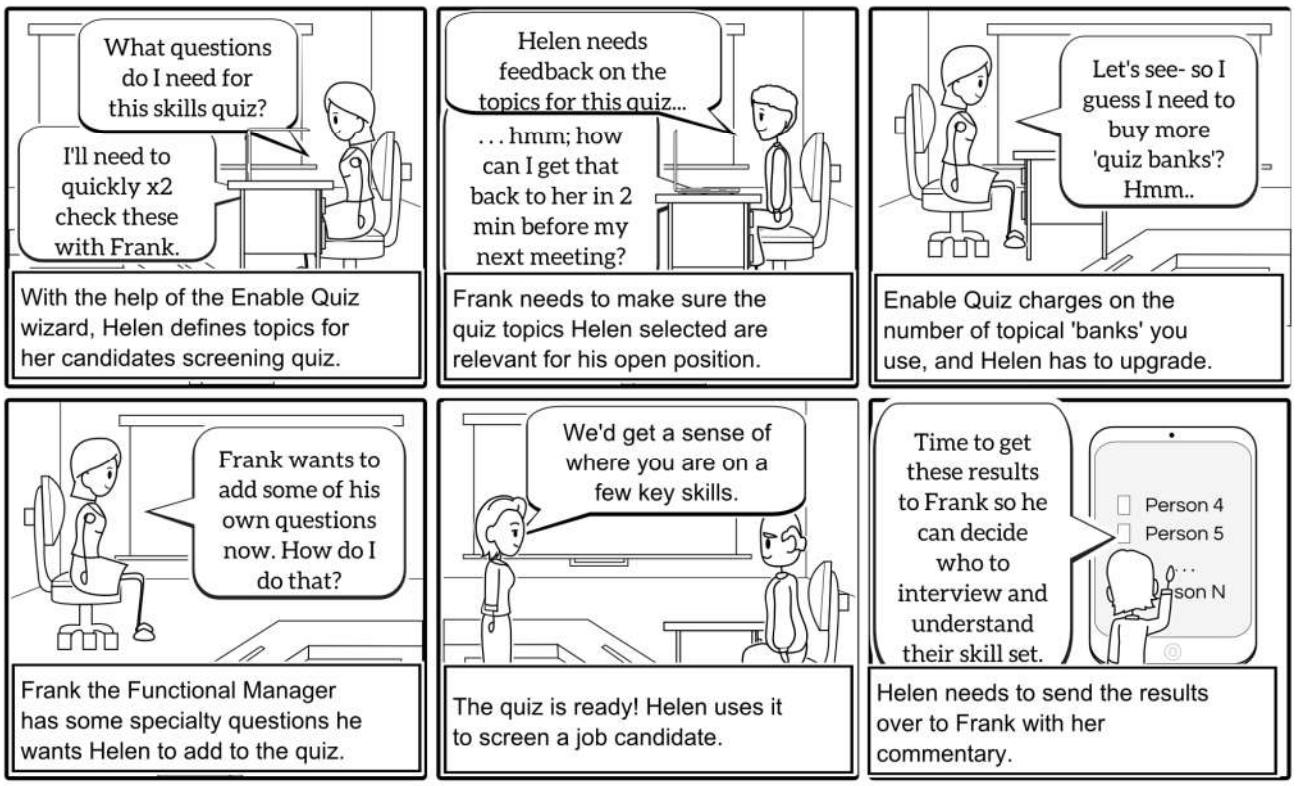
Americas
1 703 338 5421
info-usa@ivarjacobson.com

Example A: Example User Stories from Enable Quiz (Startup)

Example Epic I

This epic story deals with the example company Enable Quiz and the HR manager wanting to create a quiz to screen engineering candidates. is organized in a more conventional fashion (vs. the epic above that's storyboarded).

Epic Story: "As the HR manager, I want to create a screening quiz so that I make sure I'm prepared to use it when I interview job candidates."



USER STORY	TEST CASES
As a manager, I want to browse my existing quizzes so I can recall what I have in place and figure out if I can just	Make sure it's possible to search by quiz name Make sure it's possible to search by

reuse or update an existing quiz for the position I need now.	quiz topics included. Make sure it's possible to search by creation and last used date.
As an HR manager, I want to match an open position's required skills with quiz topics so I can create a quiz relevant for candidate screening.	Make sure the quiz topics are searchable by name. Make sure the quiz topics have alternate names, terms for searching
As an HR manager, I want to send a draft quiz to the the functional manager so I make sure I've covered the right topics on the screening quiz.	Make sure it's possible to add another user by email in this flow Make sure it's possible to include notes and customize the email Make sure it's possibly to just copy a link (for case where HR manager wants to send their own email)
As a functional manager, I want to send feedback on the screening quiz to the HR manager so I make sure I'm getting the best possible screening on candidates.	Make sure it's possibly to supply comments in-app. Make sure the above are quiz-specific by default but can also be general. Make sure it's also easy to copy the name or URL of the quiz for their own correspondence.
As an HR manager, I need to purchase an upgraded service tier so I can add additional topics to my quiz.	Make sure that users with billing privileges can upgrade the service. Make sure that If the users don't have

	billing privileges, they see a list of those that do and can contact them. Make sure the charges are correctly prorated against the billing anniversary of the account.
As an HR manager, I want to add custom questions to the quiz so we cover additional topics that are important to the functional manager.	Make sure the customer is not charged for this bank. Make sure the custom bank is owned by the client organization and not accessibly by any other accounts on the system.
'As the HR manager, I want to try out the screening quiz so that I can make sure it works as I expected and that I'm ready to both give it to candidates and share the results with the functional manager.'	Make sure there is a clear indication that the user can (and should) test the quiz Make sure there's a 'view as test taker' mode available to the admin.

Here are a few other epics that might follow this one.

Example Epic II

'As the HR manager, I want to give the screening quiz to a job candidate so I can assess their skill sets against the needs of the position.'

Example Epic III

'As the HR manager, I want to share and explain the results of our screening with the functional manager so they can decide who they want to interview.'