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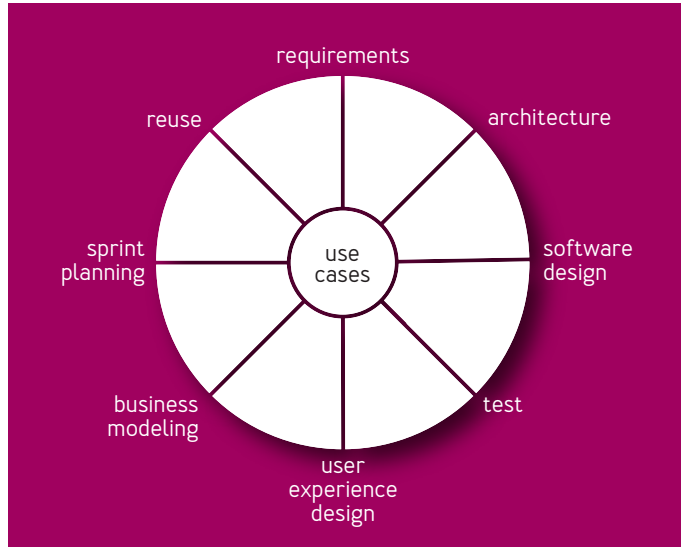
of software development—the aspects that are the spokes of the software-development wheel in which use cases are the hub (see figure 1).

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**
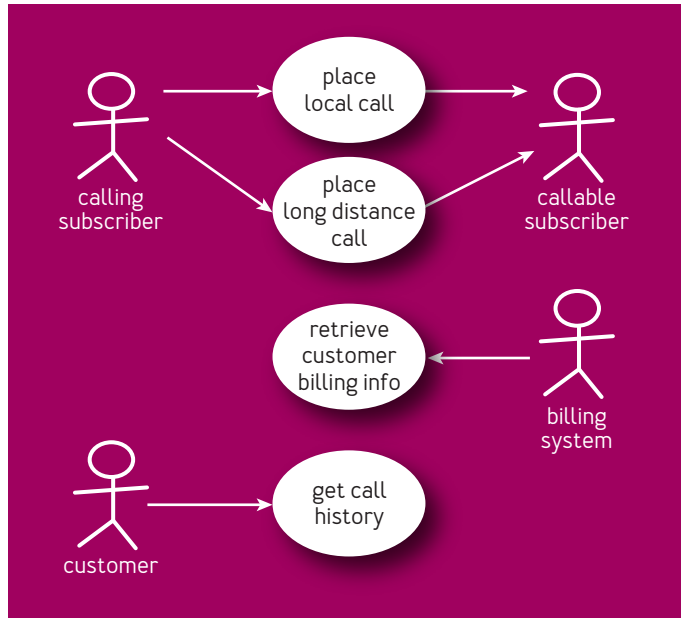
interaction, and any other parties involved. For example, a calling subscriber can place a local call or a long-distance call to any of the system's callable subscribers. You can also see that the users don't have to be people but can be other systems and, in some cases, both (for example, the role of the called subscriber might be an answering machine and not a person).
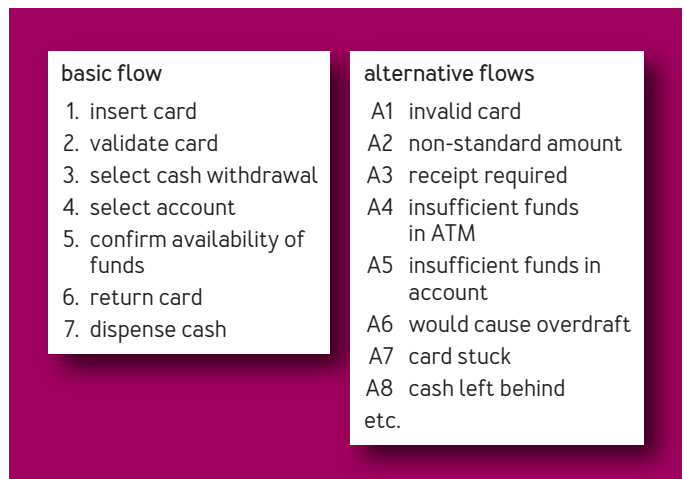
Principle 3: Focus on value

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**

basic flow
1. insert card
2. validate card
3. select cash withdrawal
4. select account
5. confirm availability of funds
6. return card
7. dispense cash

alternative flows
A1 invalid card
A2 non-standard amount
A3 receipt required
A4 insufficient funds in ATM
A5 insufficient funds in account
A6 would cause overdraft
A7 card stuck
A8 cash left behind
etc.

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**
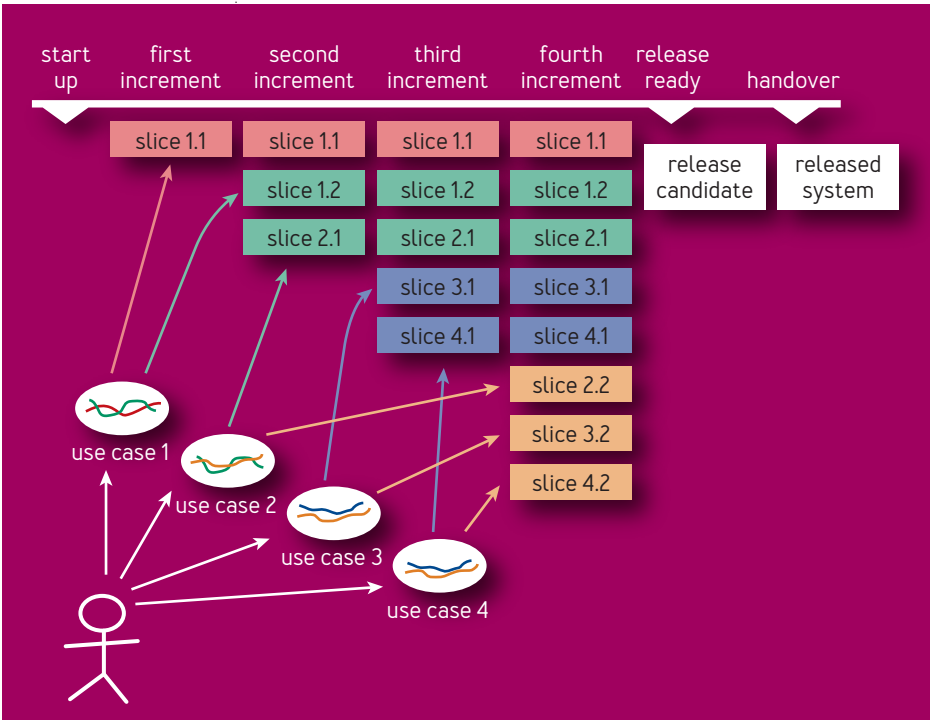
release. The first increment contains only a single slice—
the first slice from use-case 1. The second increment adds
another slice from use-case 1 and the first slice from use-
case 2. Further slices are then added to create the third
and fourth increments. The fourth increment is considered
complete 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**

shopper → 7. browse and shop

priority: MUST
release: 1
size: very large
complexity: high

a use case and its properties captured on a sticky note

7.1 select and buy 1 product

flows: BF
test: 1 product, default payment, valid details

5

7.2 select and buy 100 products

flows: BF
test: 100 products, default payment, valid details

5

7.3 support systems unavailable

flows: BF, A9, A10, A1, A12
test: select product, provide information, disconnect each system in between 13

some slices from the use case captured on their own 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**
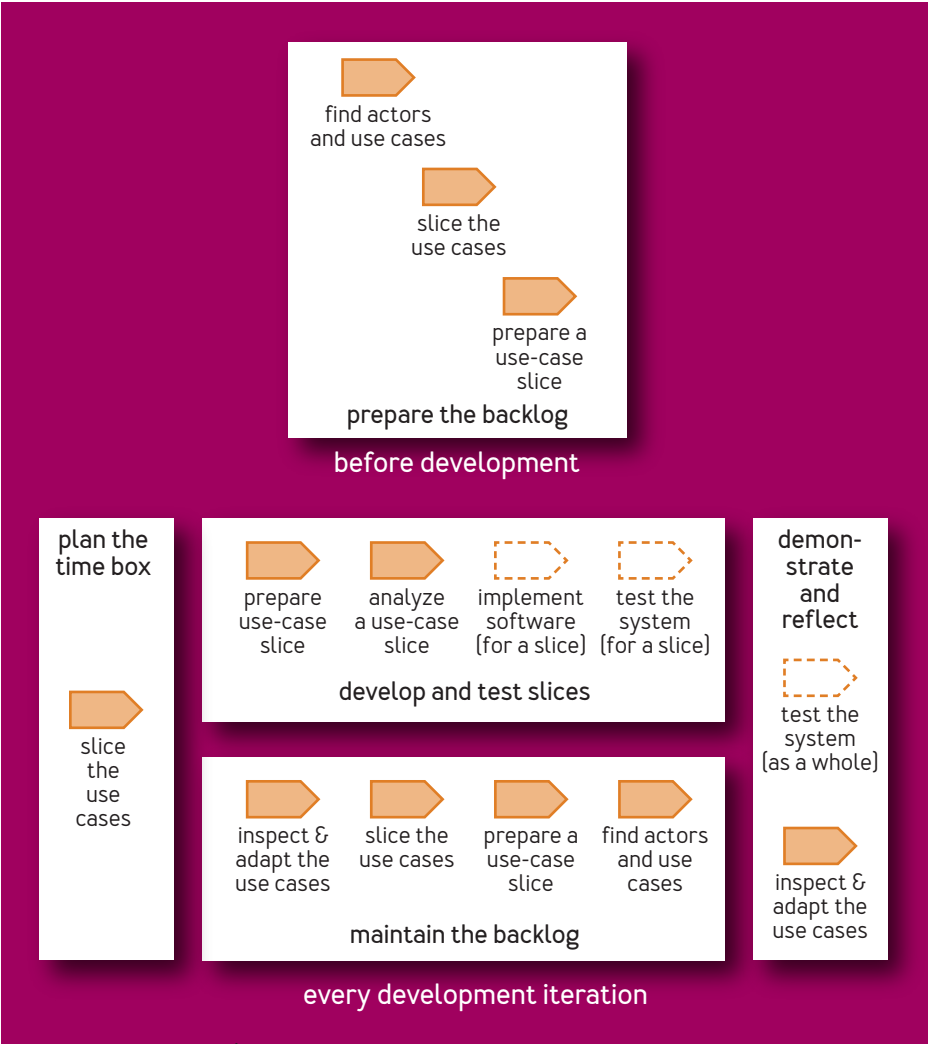
In Use-Case 2.0, the use-case slices are the primary backlog items. The use of use-case slices ensures that backlog items are well formed, as they are naturally independent, valuable, and testable. The structuring of the use-case narrative that defines them makes sure that they are estimable and negotiable, 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**
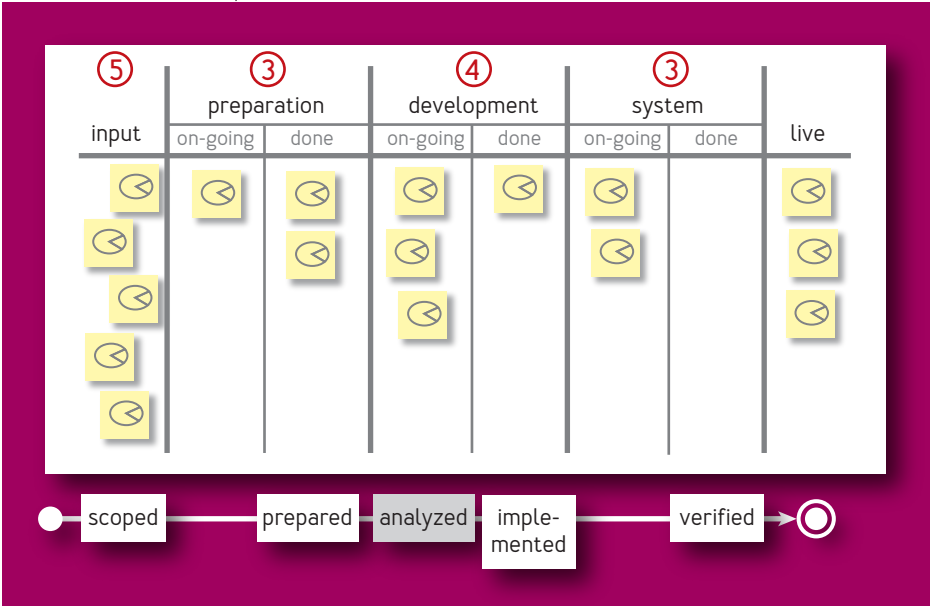
shows a simple Kanban board for visualizing the flow of use-case slices.

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.*