Why Use Use Cases?

Richard Tanner-Tremaine, Dunstan Thomas Consulting

Use Cases. I'm sure you've heard of them, or at least heard them tossed around in conversation by consultants and people “in the know”. But what's the point of drawing really simple pictures of stick figures and ellipses? Surely these "Use Cases" aren't really of much "Use"!

Well, if like many misinformed people out there you think Use Cases are just diagrams of stick figures and ellipses, then allow me to explain what Use Cases actually are and, more importantly, how they can be used effectively within your projects. In other words, before you can use Use Cases, you will need to understand what they are!

Firstly, a Use Case is not a diagram. It is perhaps a little unfortunate that whenever people talk about Use Cases they place a lot of emphasis on Use Case Diagrams instead of on the Use Cases themselves. A Use Case Diagram is actually just a UML compliant diagram to represent all of the actors (stick figures) and Use Cases (ellipses) within a project, and how they interact with each other. Use Case Diagrams are NOT Use Cases!

Use Case Diagrams are part of the UML, or Unified Modelling Language, which is a way of diagrammatically representing complex systems at different levels of abstraction to provide a simpler view of the system being developed, or parts of the system being developed. Use Case Diagrams are but one of a number of specialized diagrams in the UML, and knowing or using the rest of the UML is not a requirement for using Use Cases – just as knowing or using UML is not a requirement for using other Object-Oriented paradigms like Objects and Classes!

To better understand how Use Case Diagrams can be useful in their own right, you need to understand exactly what the elements on those diagrams are; namely Actors and Use Cases.

Some important definitions for you:

Actor

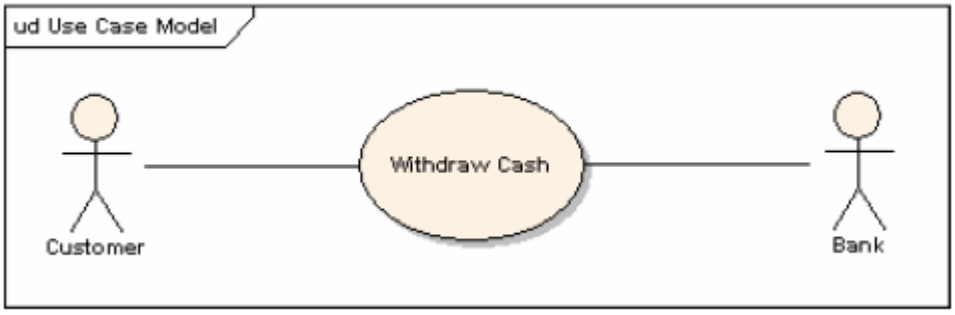
An Actor is anything external to the system that interacts with the system.

Use Case

A Use Case is a sequence of events in the system that yields an observable result of value to a particular Actor.

On a UML-compliant Use Case Diagram, Actors are represented as stick figures with the name of the Actor underneath. Somewhere, you will need to document what the Actor actually is. A little stick figure with a name doesn't really impart any information on it's own. A common practice is to either describe each Actor within the system in a Glossary document, or in a document called a Use Case Model Survey. A Use Case Model Survey is simply a document which lists all of the Actors and Use Cases in the system with a short (usually 1 paragraph) description of what each of them are, and how they interact. The idea is that having a Use Case Model Survey document in tandem with a Use Case diagram you should be able to get a reasonable high-level understanding of what a system involves. You do not *have* to have a Use Case Model Survey document, but the kind of information that document contains should be captured somewhere so that anyone looking at your Use Case diagrams will be able to decipher what each of the UML entities in your diagrams represent.

For example, a really simple Use Case Diagram might look like this:



An accompanying Use Case Model Survey would look something along the lines of this:

Actors

- Customer
  - A "Customer" is any individual with a credit or debit card who uses the ATM.
- Bank
  - A "Bank" is any banking institution at which a customer can have a bank account.

Use Cases

- Withdraw Cash
  - The system can be used by a Customer to withdraw cash as long as they insert the bank card, enter the correct pin, and have sufficient funds available in their bank account in order to withdraw the amount that they request.

Suddenly the Use Case Diagram makes perfect sense (or at least we can say that we now understand what the Use Case Diagram is trying to communicate). Without a description of the Actors and the Use Case, the diagram would be open to multiple interpretations. Having the accompanying document allows us to narrow the scope so that ambiguity is reduced and there is improved understanding for anyone reading our documentation or looking at our diagram.

So how much detail should be included in these descriptions? The idea of a Use Case Model Survey document is really just to give a high-level understanding of the system to the reader rather than a detailed technical description. For the detailed description, we will write an additional document per Use Case that specifies exactly what happens for each Use Case. These additional documents are our Use Case Specifications. We will also write a Supplementary Specification that will capture all of our non-functional requirements (more on that later).

Before I get stuck into Use Case Specifications, let's take another look at the Actors in the above example. Bear in mind the definition for an Actor: Anything external to the system that interacts with the system. The system we are developing is an Automatic Teller Machine (ATM). Anyone using the ATM (customer) is external to the system, so is an Actor. Likewise, our ATM communicates with the banking institutions' systems, which are external to our system, and so they are also represented as an Actor. The key here is that you should think of Actors as Roles rather than as individuals. In other words, we could potentially have millions of individual people wanting to use our ATMs - we are not going to represent each possible customer as an Actor in their own right! Likewise, our ATM might communicate with several different banking institutions (there are literally thousands worldwide), but we will conceptualise them as a role in the form of an Actor called "Bank".

For a particular scenario, one specific customer (Mr. Brown) will want to withdraw £10 from his bank account (held at HSBC). For that scenario, the "Customer" Actor's role will be filled by Mr. Brown, and the "Bank" Actor's role will be filled by HSBC... but our "Withdraw Cash" Use Case sequence of events should remain the same regardless of which bank and which customer we're talking about. Take another look at the definition of a Use Case: A sequence of events in the system that yields an observable result of value to a particular Actor. The sequence of events should not be affected by the fact that it's Mr. Brown and HSBC we're talking about for this scenario. Sure, there might be minor alterations when we get down to actually implementing the interface that communicates with the different banks, but at a conceptual level nothing will change.

The "observable result of value" for the "Withdraw Cash" Use Case is that Mr. Brown (the "particular Actor" who initiated the Use Case) ends up with the £10 he asked for, or is told that he can't have any money right now for whatever reason (no money in his account, the ATM is out of cash, the banking system is offline, etc).

Now that we have a better understanding of what Use Cases and Actors are, let's see how we go about defining a Use Case - this is done in a Use Case Specification document. The purpose of a Use Case Specification is to detail exactly what the sequence of events is during the lifetime of a Use Case that will lead to the "observable result of value" being delivered to the Actor who initiated the Use Case.

The real power of using Use Cases lies in these Use Case Specification documents. They are the most important aspect of capturing requirements using Use Cases, and without them, the Use Case diagram and Use Case Model Survey documents are almost pointless as they alone do not effectively capture any requirements - they help give a high-level understanding of what's involved in the system being developed, but they don't capture the actual requirements for it! That's the job of the Use Case Specification.

Once again, we'll consider the example above and refer to the "Withdraw Cash" Use Case identified for our ATM system. It's important to make sure we understand what a Use Case is before we can write out a specification for one! The definition is worth breaking down.

"... sequence of events ..."

- It's a sequence, so we expect A to happen, then B, then C, etc. In our example, we'd expect something along the lines of "insert card", "enter pin", etc.

"... yields an observable result of value ..."

- At the end of the Use Case sequence of events, we expect there to be some kind of observable result. In this example, we're expecting the Customer to have cash in hand, or an explanation for why he doesn't.

"... to a particular Actor ..."

- The Use Case is viewed from a particular Actor's perspective, so we'd expect the sequence of events to be written from that Actor's perspective too.

Typically, a Use Case Specification will contain the following:

- A brief description of what the Use Case is about.
- A flow of events written in the form of a dialogue between the Actor who initiated the Use Case and how the system reacts.

- Any alternate possible flows of events that could occur.

Additional information a Use Case Specification might contain:

- Pre and Post Conditions
- Special Requirements
- A list of extension points.

For the purposes of this discussion, I'll focus on the first 3, which will typically appear in all Use Case Specifications in some form or another. Here is a simplified example extract of what a Use Case Specification for the Withdraw Cash Use Case would look like:

Use Case Specification: Withdraw Cash

Brief Description

The system can be used by a Customer to withdraw cash as long as they insert the bank card, enter the correct pin, and have sufficient funds available in their bank account in order to withdraw the amount that they request.

Basic Flow

1. The Use Case begins when the Customer inserts their bankcard into the ATM. The system responds by prompting the Customer to enter their PIN number.
2. The Customer types in their PIN number. The system then prompts the Customer to select the action they wish to perform.
3. The Customer selects “Withdraw Cash” from the options provided. The system responds by asking the customer how much they would like to withdraw: £10, £20, £30, £40, £50, £100, £200, Other.
4. The Customer selects one of the given amounts (not Other). The system connects to the Bank to check for available funds. The system asks the Customer if they wish to have a receipt printed for this transaction.
5. The Customer selects “No Receipt”. The system informs the Bank of the transaction amount, and ejects the Customer’s bankcard.
6. The Customer takes their bankcard out of the ATM. The system dispenses the cash requested.
7. The Customer takes the cash from the ATM and the Use Case Ends.

Alternate Flows

1. In Step 2 of the Basic Flow, the Customer enters the incorrect PIN number. The system then re-prompts for the PIN number up to a maximum of 3 times. If the correct PIN number is entered, the system then prompts the Customer to select the action they wish to perform and the Use Case resumes at Step 3 of the Basic Flow. If the maximum number of tries is reached, the Customer’s bankcard is retained and they are told to speak to the bank about retrieving it and the Use Case ends.
2. In Step 3 of the Basic Flow, if the Customer selects another function such as “Balance Enquiry”, “Change Pin Number”, etc., refer to the Applicable Use Case, and this Use Case ends.
3. In Step 4 of the Basic Flow, if the Customer selects “Other” for the amount, the system prompts them to enter a value in multiples of £10 up to a maximum of £300. The Customer enters the desired amount and the system connects to the Bank to check for available funds.

The system asks the Customer if they wish to have a receipt printed for this transaction. The Use Case resumes at Step 5 of the Basic Flow.

4. In Step 4 of the Basic Flow (and in Alternate Flow 3), if there are insufficient funds available for the amount requested, the Customer is informed of the problem and the Use Case resumes at Step 4 of the Basic Flow.
5. In Step 5 of the Basic Flow, if the Customer chooses to have a receipt printed, the system will inform the bank of the transaction amount and print out the receipt before ejecting the bankcard. The Use Case resumes at Step 6 of the Basic Flow.
6. At any time, the Use Case can be ended by the Customer selecting "Cancel". The Use Case ends.
7. At any time, if communications with the bank are unavailable or interrupted, the Customer's card is returned to them and a suitable error message is displayed on screen. The Use Case ends.

As you can see, the Use Case is written from the Customer's perspective. It tells us what actions the Actors need to be able to perform, and how our system should respond to those actions. It is a dialogue between the Actor who initiates our Use Case and our system. You will also note that in this example, the Brief Description is the same as the description given in the Use Case Model Survey document we looked at earlier. This is common practice, though is not required. If you feel more detail should be added to the brief description in the Use Case Specification, feel free to expand on it – just be careful not to get stuck into the flow of events in the description.

The Basic Flow is meant to be the "most likely sequence of events" for a Use Case. Typically, the Basic Flow is when everything goes according to plan! As a result, the Basic Flow is often referred to as the "Happy Day" scenario.

Any deviations from the "Happy Day" scenario are expressed as Alternate Flows. Complex Alternate Flows can be further broken down into Sub-Flows, as can complex steps in the Basic Flow. The idea is that this document should clearly define the interaction between our system and the Actor in terms of this Use Case (this piece of required functionality). If adding in a flow-chart or sequence diagram or sub-flow will help us in communicating the concepts and sequence of events, then we should feel free to do so. There is no hard-and-fast rule saying we have to stick to any particular formula! If everyone involved in the project (from your customer to the developers, testers, and project managers) can understand what is being said, then it is a good Use Case Specification! If not, then it needs some work.

While the above example is fairly simple, I hope it gives you an idea of what to expect when reading Use Case Specifications, and how to go about writing your own. It is suggested, however, that for any given project you use some form of template in writing these documents so that moving from one Use Case Specification to another is easier for all concerned.

As with any requirements gathering exercise, it is important to make sure that we include in our discussions anyone who has a vested interest in the Use Case. It is also important to ensure that our Use Case Specification (and indeed our whole Software Requirements Specification) can be said to adhere to the following:

- It is correct
- It is complete
- It is consistent
- It is unambiguous

- It is verifiable
- It is modifiable
- It is understandable

Note that a Use Case Specification is specific to a particular Use Case - we're only talking about the Withdraw Cash functionality and haven't concerned ourselves with other possible Use Cases here (for example, Balance Enquiry or Change Pin Number – we've simply referred the reader elsewhere for that information). It is also concerned only with Functional requirements (i.e. how the system functions or is used). Non-Functional Requirements such as those relating to Usability, Reliability, Performance and Supportability are not included in a Use Case Specification. The reason for this is that those non-functional requirements typically span across multiple use cases and are seldom specific to a single piece of required functionality. So, by separating non-functional requirements and placing them in another document called the Supplementary Specification, we can reduce the complexity of our Use Case Specifications by letting them focus exclusively on the functionality required by the system.

Effectively, instead of the traditional monolithic document that contains paragraph after paragraph and page after page of descriptions (we've all had to trawl through 100 page software requirement specs before - not a pleasant task!), we now have several shorter, more focused, and essentially eminently more useful documents. Our Software Requirement Specification (SRS) is actually now made up of a number of Use Case Specifications (one per Use Case in the project) and a Supplementary Specification (one per project), with a Use Case Diagram and Use Case Model Survey being optional extras as supporting documents.

Yes, you read that last bit correctly! The Use Case Diagram is OPTIONAL. You can use Use Cases to capture your functional requirements without ever having a Use Case Diagram! After all, it's just a picture! So why have them at all? To aid in communication of course! We usually include a Use Case Diagram because it's a visual way of summarising the Actors and Use Cases in our system and the relationships between them. It serves as a very useful tool when discussing the project at a high level, and is invaluable when used as a quick overview. And since it only takes a few minutes to draw one, we'll usually have one when we're working with Use Cases. But bear in mind that the diagram is like the "Index" to the "Book" that is your software specification, rather than the specification itself.

Having our SRS split into smaller documents has other benefits too. Easier change control, easier management for distribution of sensitive material, and greater opportunities for parallel development are just a few that spring to mind. It's also much easier to meet the user's requirements if you write your functional specifications from the user's perspective (which is what Use Cases essentially are).

There is no denying that using Use Cases still means that there will be a fair amount of documentation. Unfortunately, there is no magic pill we can take to convert our old 100-page software specs into one or two 5-page documents! The idea of splitting out our SRS into several smaller documents focused on particular bits of functionality requirements does, however, make managing our documentation a lot easier, both from the perspective of keeping it up to date (change control), as well as distribution since we only need to send out the document that changed instead of the whole SRS!

There are hundreds of books, articles and training courses on effective Use Case writing, so it's worth realising that this is a big topic for discussion and that there is definitely "a knack" to writing them. While I won't go into that discussion too much in this article, just to give you an idea of what

to expect, here are a few guidelines when setting out to capture your functional requirements in the form of Use Cases:

1. **Always remember that Actors should be viewed as roles rather than individuals.** For example, a web-based content management system might have roles for Readers, Authors, Editors and Administrators. A single individual might only take on one of those roles, but they could also take on multiple roles. For example, John Smith might only read the articles on the site, but Brian Jones writes articles for the site (he's an Author) as well as reads articles on the site. Malcolm Watts will also read articles, but he's responsible for managing the content, so he's an Editor too. Jack O'Falltrades might be the Administrator responsible for things like data backup, user management, etc., but has no interest in the articles and has no responsibility for the content, so he's only an Administrator. If you identify your Actors as roles rather than as individuals, you can greatly simplify your Use Case specifications by not having to have an Actor defined for each possible combination of roles.
2. **Always write your use cases from the user's perspective.** Perhaps the greatest benefit of Use Cases is that when correctly written, everyone involved on the project, from user/customer to developer can read a Use Case and derive some benefit from it. Use Cases should be written from the user's perspective to ensure that everyone understands what it is the user needs the system to do.
3. **When naming your Use Cases, think "Verb - Noun".** For example, "Manage", "Report", and "Print" aren't particularly good Use Case names, whereas "Manage Stock", "Submit Report", and "Print Invoice" are much better.
1. **The flow of events (Basic and Alternate Flows) should be written as a dialogue between the user and the system.** This makes it easier for the user to understand how the system will eventually work, and is also useful to the software analysts / designers, as well as for the testers. A well-written Use Case will allow the testers to being outlining their test plans very early on in the project since they already know how the system is supposed to behave!
4. **Try to keep the number of Use Cases to a minimum.** Typically, aim to have a maximum of around 5 Use Cases for a conceptually simple system, from 5 to 10 for most systems, and less than 20 for ANY system. This might sound difficult at first since there is a tendency to write Use Cases for absolutely every possible action that can be performed on a system. Try and avoid that trap! Think about how you can combine candidate Use Cases you identify early on into a single functionally cohesive Use Case. For example, if the system needs a module for user management, you might identify "Add User", "Edit User", "Delete User", and "Change Password" as possible Use Cases. A better approach would be to have a single Use Case called "Manage Users" with the most commonly used of those initially identified Use Cases being the Basic Flow, and the others as being Alternate Flows of the "Manage Users" Use Case.

Now that we understand what Use Cases are, and how they are identified and documented, we can return to our original question...

Why Use Use Cases?

There are several ways to answer this question, but there are 4 main considerations.

1. Use Cases as a Better Form of Documentation

Instead of one monolithic Software Requirement Specification (SRS) in a single document, using Use Cases encourages us to split up our SRS into more focused smaller documents which together cover all of the requirements a traditional SRS would contain. Each Use Case has a Use Case Specification

document which focuses exclusively on the functional requirements for that particular Use Case. This allows us to refer to a single document that is primarily concerned with the functional requirements when considering a particular piece of functionality, instead of having to trawl through hundreds of pages in order to extract bits and pieces from the traditional SRS to identify all of the requirements for it. If we need to change a particular piece of functionality, we only need to change the Use Case that functionality is expressed in, and won't need to go through the documentation for the entire project!

Similarly, by separating out the non-functional requirements into a Supplementary Specification document, we have a single point of reference and control for all non-functional requirements in our project.

As a consequence, change control and configuration management are made easier. Instead of having to distribute one massive document to everyone on the project every time a minor change is made, we can now version control each individual Use Case Specification and the Supplementary Specification, each of which can have a smaller distribution list of only those people who need to know about changes to a particular Use Case Specification. It is also easier to manage additional concerns such as confidentiality and security since each Use Case Specification can have its own rules for distribution.

As an added advantage, if you adopt a template approach to writing Use Case Specifications, there will be greater consistency in the presentation of documents on your project, and indeed across projects in your organisation, which should help everyone involved on the projects to more easily read and understand the functional requirements across all of the Use Cases.

2. Use Cases as a Tool For Communication

Since a good Use Case Specification is one that is written from the user's perspective, it serves as a useful document for discussing the functional requirements of the system being developed with everyone involved on the project. It should be something that the end-user (customer) is able to read and understand relatively easily, while still giving the analysts, designers, developers and testers sufficient understanding for doing their jobs. A Use Case Specification is not intended as a highly technical document, but rather as a document that all parties involved on the project can use to agree the scope and functionality of the project.

Since it's written from the user's perspective, it is much easier to involve the user throughout the project, and it is easier to get agreement on what will be developed. Traditionally, projects often fail because there is a gap between the user expectations and the project team's understanding of the user's requirements. A major aim of Use Cases is to have a common document that covers all functional aspects of the software being developed and that serves as a facilitator for discussions about the system being developed and allows for common understand of and agreement on what is being developed.

Essentially, Use Case Specifications can aid communication between all interested parties on a project and helps to ensure that everyone is "on the same page". They help us manage user expectations.

3. Use Cases as a Tool For Project Management

Having your Software Requirement Specification captured in the form of Use Case Specifications and a Supplementary Specification, there are several additional benefits for Project Management. Since we are defining the functional requirements in terms of Use Cases, we are effectively breaking down the system into smaller parts in terms of functional decomposition. While this does not mean that all functionality will be exclusive to a particular Use Case (some functionality, or rather some implementations of functionality, might be shared across Use Cases), it certain lends itself to parallel development for disparate pieces of functionality. This is especially true if we refrain from thinking of development as the same thing as coding! For example, it's possible for our systems analysts and test writers to work on the same Use Case simultaneously. It is also possible that while designers are evolving the results of analysis for a particular Use Case, the analysts could be working on another one. And while the designers are coming up with the designs for one Use Case, programmers could be implementing another one that has already been through the design phase. Essentially, the very act of decomposing the project into smaller parts based on functionality lends itself to exploiting opportunities for parallel development.

Another interesting side effect of this functional decomposition aspect to Use Cases as far as project management is concerned is that we immediately have some potential for monitoring project progress. If we can identify a few key variables for each Use Case such as risk, complexity, size, and available resources, we should be in a better position to gauge not only how long a Use Case will take to develop in its entirety, but also to gauge how far through the project we are. Naturally, as with any estimation technique, learning to accurately predict how long any part of a software project will take is somewhat more of an art than a skill, but being given a good starting point by considering each individual Use Case makes the job of the Project Manager that little bit easier than it would be having to estimate off the traditional single SRS.

There is a tendency to associate Use Cases with iterative development processes. It is worth noting at this stage that using Use Cases to capture functional requirements does **not** imply that you will be using UML, or that you will be using an iterative development process (for example, the Unified Process). However, having said that, both UML and Iterative processes leverage the power of Use Cases very effectively, and so are often used in tandem. In a typical iterative approach, the first phase of any project involves identifying all of the Use Cases for the project, and doing some initial analysis on them with the aim to identify those Use Cases that pose the greatest risk to the projects overall chance of success or failure. The idea would then be to tackle the high-risk Use Cases in early iterations in the project to eliminate project risk. If the risk cannot be eliminated early on then the project will still fail, but with as little amount of wasted effort and expenditure as possible. This is different to the traditional "waterfall" approach where the tendency is to put off the tricky stuff until last, or where you cannot actually gauge the success or failure of the project until you're about to deliver the final product! When used in conjunction with an iterative process, Use Cases give us an excellent way of managing project risk.

4. Use-Case-Driven Development

Using UML is not a requirement for using Use Cases to capture functional requirements for a system. However, you will often find that project teams that employ Use Cases also use UML to model the system being developed. There are established techniques for using the flow of events in a Use Case Specification to define analysis versions of Sequence and Communication Diagrams, which in turn are used in identifying analysis classes, the results of which directly feed into the design process wherein design classes are identified and fleshed out. Those design classes are then used by the

programmers to implement the system. The whole procedure forms part of what is known as "Use-Case-Driven Development", where Use Cases are used to drive the development process. Through each of the stages in the process, the Use Case Specification is constantly referred back to (and if necessary, discussed and updated) to ensure that the decisions made in analysis, design and implementation are consistent with the functional requirements expressed in the Use Case Specification.

As mentioned before, the Use Case Specification also "feeds" the testing aspect of a project in that the test team can begin scripting their test cases even before the design or implementation are complete. In fact, even plans for eventual deployment of the system can be partially addressed directly from the Use Case. The whole process is "Use-Case-Driven" in that almost everything stems directly from the Use Cases.

Conclusion

Any one of those main reasons on their own should be enough for you to consider using Use Cases on your projects. Use Cases, when used effectively, can help alleviate some many of the common problems encountered in modern software development. Indeed, many of these problems are in fact common to projects in other disciplines, and Use Cases can be (and are) used in areas other than software development to help manage those problem areas. Use Cases aren't just the pretty diagrams of stick figures and ellipses that many of us think of them as being. They are a proven way of capturing and managing functional requirements for any system being developed, and can be an integral part of improving the way you work from requirements gathering right through the project life-cycle up to user acceptance testing and deployment.

If you aren't using Use Cases on your projects, now that you know just how useful they can be, why not give them a try? Adopting Use Cases as your method of documentation does not imply having to adopt UML, RUP, or any other process or tool which deals with Use Cases - even on their own, Use Cases are an effective method of documentation that I believe any project can benefit from.