Use Cases -
An Introduction

Jason Gorman

www.parlezuml.com

Table of Contents

Introduction

Search on Amazon for books on use cases, and you'll find hundreds upon hundreds of them. Search on Google for web sites that mention use cases, and you'll find millions. I don't think there's any aspect of software development that's received wider coverage in the last ten years.

But for all that's been written about them, most people still don't really understand them. This is partly because a lot that's been written about use cases overcomplicates them and confuses the reader.

In theory and in practice, use cases are much simpler than many authors make out. This short tutorial is designed to introduce use cases in their simplest form, and to show you how they can be effectively applied in the software development process.

It's not intended to be a detailed or comprehensive guide to use cases and use case-driven software development. Rather, this tutorial will show you the 20% of use case theory that you will probably need to use 99% of the time.

What Is A Use Case?

In the mid-1980's, Ivar Jacobson put forward the idea of *usage cases* and *usage scenarios*. More recently these have become popularly known as **use cases** and **use case scenarios**. There's no rocket science to it at all: a usage case is simply **a reason to use a system**. For example, a bank cardholder might need to use an ATM to get cash out of their account. It's as simple as that.

Fig 1.0 – A use case describes how a type of user (called an "actor") uses a system to achieve a goal

There are three key things we need to know to describe a use case:

1. The **actor** or actors involved. An actor is a type of user (for example, cardholder) that interacts with the system.
2. The **system** being used.
3. The functional **goal** that the actor achieves using the system – the *reason* for using the system.

Believe it or not, if you can remember this, then you already know a third of what there is to know about use cases (that's worth knowing, of course.)

There's a little more to it than that, and with practice you'll soon get the hang of it. Some things to bear in mind are:

- The actor describes a role that users play in relation to the system. Maybe the cardholder is an advertising executive, but that doesn't interest us. We only care what his relationship to the system is.
- The actor is external to the system itself.
- Actors don't have to be people. They can be other systems. For example, the ATM may need to connect to the cardholder's bank. External systems that interact in a use case are also actors.
- The goal must be of value to the actor. We wouldn't have a use case called "Cardholder enters PIN" because that, by itself, has no value to the cardholder. We don't build ATM's just so people can enter their PINs!

When we are analysing functional requirements for a system, the key questions we need to ask are; **who will be using the system, and what will they be using it to do?**

What Is A Use Case Scenario?

When a cardholder tries to withdraw cash from an ATM, it doesn't always necessarily turn out the same way. Sometimes he gets his money. Other times he might have insufficient funds. Or the ATM may be out of cash. These are all examples of **use case scenarios**. The outcome is different, depending on circumstances, but they all relate to the same functional goal – that is, they're all triggered by the same need - and all have the same starting point.

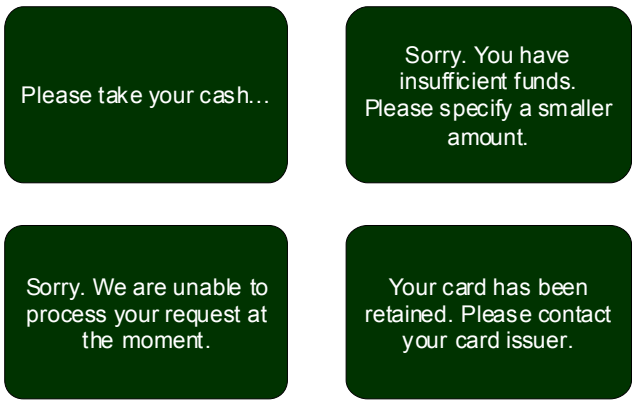

Fig 1.1. Use case scenarios for withdrawing cash from an ATM

In practice, we describe use cases by describing the key scenarios. Use case scenarios form the basis of **interaction design**, but also map directly onto other useful development artefacts like system test scripts and user documentation.

Capturing Use Case Scenarios with Essential Use Case Descriptions

Once we understand the actor and the goal for a use case, and have identified key use case scenarios, we can begin some high-level *interaction design*. Actors interact with the system – by pressing buttons, typing into text boxes, clicking on icons and so forth – to achieve the goal of the use case.

A classic mistake made at this early stage of design is to go into technical detail and commit to a specific user interface design or implementation technology. This is almost always the wrong time to be making these kinds of low-level design decisions. We first need to understand what the business logic of the interactions are, so we can focus on satisfying the business goal of the use case.

Essential use cases are a great technique for describing interactions in a way that is independent of the technical implementation of the system. Instead of saying "the user presses the enter button", we say "the user confirms their choice", for example.

A good way to write essential use cases is to split the actions into columns, one for each actor and one for the system. Then we can see at a glance not only the order of events in a use case scenario, but also exactly who is doing what.

These essential use case descriptions, one for each key use case scenario, will form the basis of our high-level object oriented design, the UI design, and are also the foundation for system test design, user documentation and other useful things we might need later in the development process.

Cardholder	ATM
•select "withdrawal" option	
	•display withdrawal options
•specify amount	
	•check cardholder has sufficient funds
	•eject card
	•prompt cardholder to take card
•take card	
	•dispense amount
	•prompt cardholder to take cash
•take cash	
	•debit cardholder's account
	•thank cardholder
	•display welcome and await next cardholder

Fig 1.2. An essential use case clearly shows the order of events and the responsibilities of the actor(s) and system in a single use case scenario, without committing to technical design decisions

WARNING!

99% of teams are unaware that use case descriptions like the one above are *not* system requirements documents. These are high-level interaction designs. The danger is that if we mix them up with real requirements – stuff the system really *has* to do – then we can get bogged down in the design decisions we make early on. Changing the design of use cases must always been seen as an option if it turns out there's a better – or cheaper – way of achieving the same functional goal. Be one of the smart 1% and always remember that use case designs aren't the same thing as requirements.

Modeling Essential Use Case scenarios using Sequence Diagrams

If you're familiar with UML sequence diagrams, then you can just as easily model a use case scenario like the one above using a simple sequence diagram. This also shows the order of events, the interactions and clearly shows who's doing what.

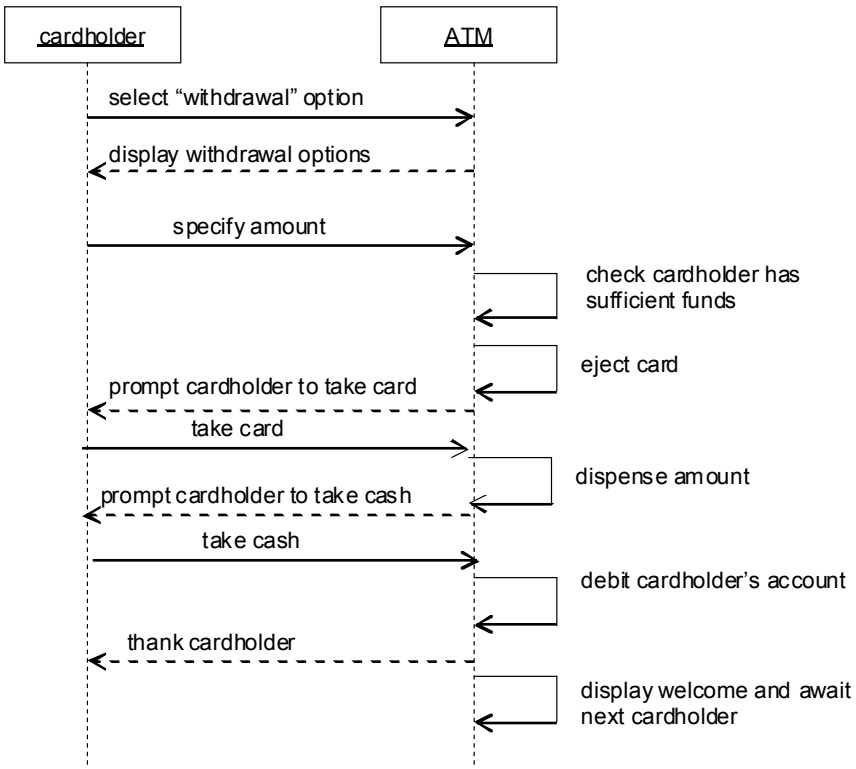

Fig 1.3. A use case scenario modelled using a sequence diagram

It has the added advantage – or disadvantage, depending on how you look at it - that you can capture it in a modelling tool that supports UML, and you can continue to flesh out the implementation design behind the user-system interactions using the same model.

Visualising Scenarios using UI Storyboards

If we're happy with the logical flow of a use case scenario, we can use a simple technique called **storyboarding** to show how it might look with a real user interface implementation. At each step in the scenario, we can draw a picture of what the user interface would look like at that point in time, and a sequence of such pictures, one for each interaction, neatly illustrates the flow of the scenario in a way even non-technical folk can readily understand.

Again, it's vitally important to remember that what we're doing here is *design*. And that means we have *choices*. Beware of committing to the first UI design you do without exploring alternative options.

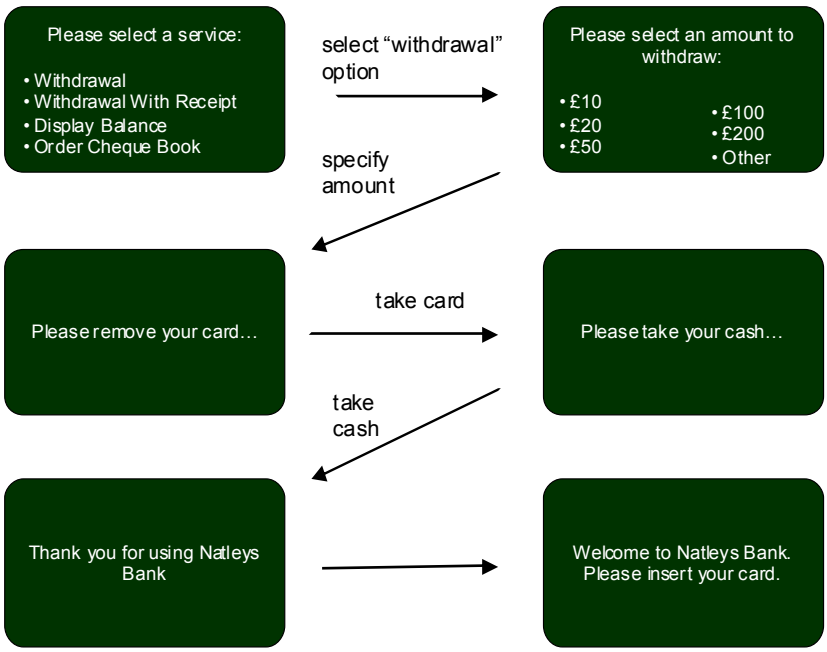

Fig 1.4. Storyboards are a powerful tool for visualising and communicating use case scenarios and UI designs

If you have identified your key use cases, and have essential use case descriptions for key use case scenarios and storyboards too, then by this stage your users probably have a pretty good idea of what it is they're going to be getting. Importantly, you will have a pretty good idea of what it is you're going to build, too. This is a critical breakthrough for development teams. You have passed *the point of no surprise*.

Use Cases & Rework

There are all sorts of reasons why we might need to change software, and some of them are inescapable – hence the need for an iterative and evolutionary approach to development. But there are two reasons that are largely avoidable and should be seen as undesirable.

1. Building something the user was expecting to get, but building it wrong.
2. Building something the user wasn't expecting to get.

If the user asks for feature X, and you deliver feature X working, and a month later the user says "actually, what I really needed was X+1" then there's not a lot we can do about that. That sort of change is unavoidable, and we must embrace the need to accommodate it.

But if the user asks for feature X, and we deliver X+1 – either because we did a bad job of delivering X, or because we misunderstood and thought the user actually wanted X+1 in the first place - then we have failed.

Use cases don't help us to avoid null pointer exceptions or un-initialised database connections, so we still have to do "coding stuff" to make sure we avoid those kinds of errors. But use cases can definitely help us to avoid misunderstandings between us and the users. In fact, I see no other good reason for them. Do you?

This gives us a useful guideline as to how effectively we're applying use cases. In theory, rework due to requirements misunderstandings should drop to almost zero. That's not to say that we won't need to do any rework: we just won't need to do any *unnecessary* rework.

Use Cases & UML

Given how closely associated they are, you probably won't be surprised to learn that the *Unified Modeling Language* has a specific notation for modelling use cases.

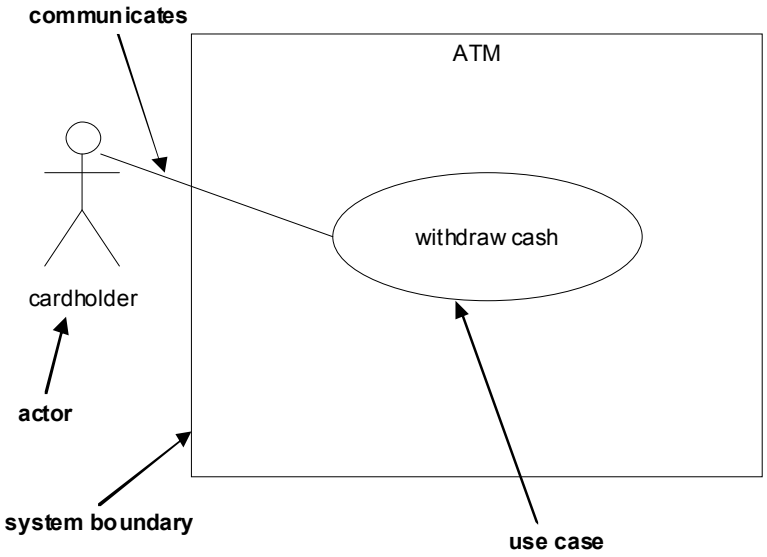

Fig 1.5. UML Use Case diagram

In UML use case diagrams, actors are drawn as stick men, with the name of the actor written underneath. A use case is drawn as an oval containing the name of the use case. To show that an actor participates in a use case, we draw a line between the actor and the use case that shows that the actor *communicates* with the use case. Optionally we can draw a box around the use cases to show where the system boundary is. The actor is always outside the system boundary (by definition).

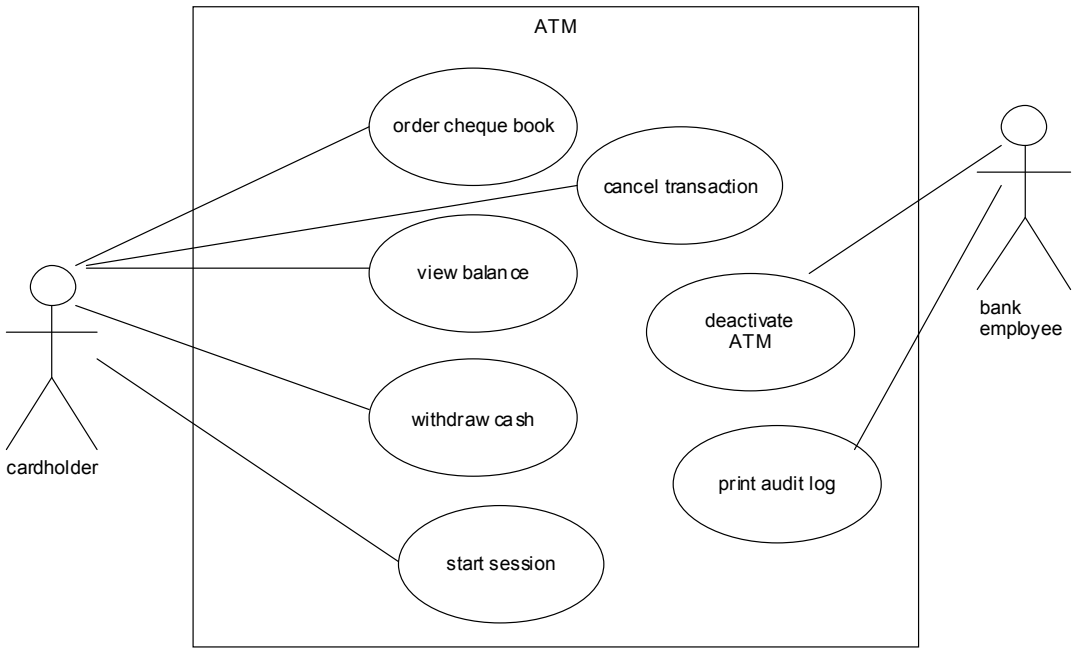

Fig 1.6. A use case diagram for the ATM

Relationships Between Use Cases

Including Use Cases

When I make a cup of tea, I boil the kettle. I also boil the kettle when I make a cup of cocoa. When two or more use cases include the flow of another use case, they are said to *include* that use case's flow.

We can illustrate this relationship in a use case diagram by simply drawing a dotted line with an arrow pointing towards the included use case from all the use cases that include it. (With me so far?) The arrow should have the UML stereotype **<<include>>** to clearly show what kind of relationship it is.

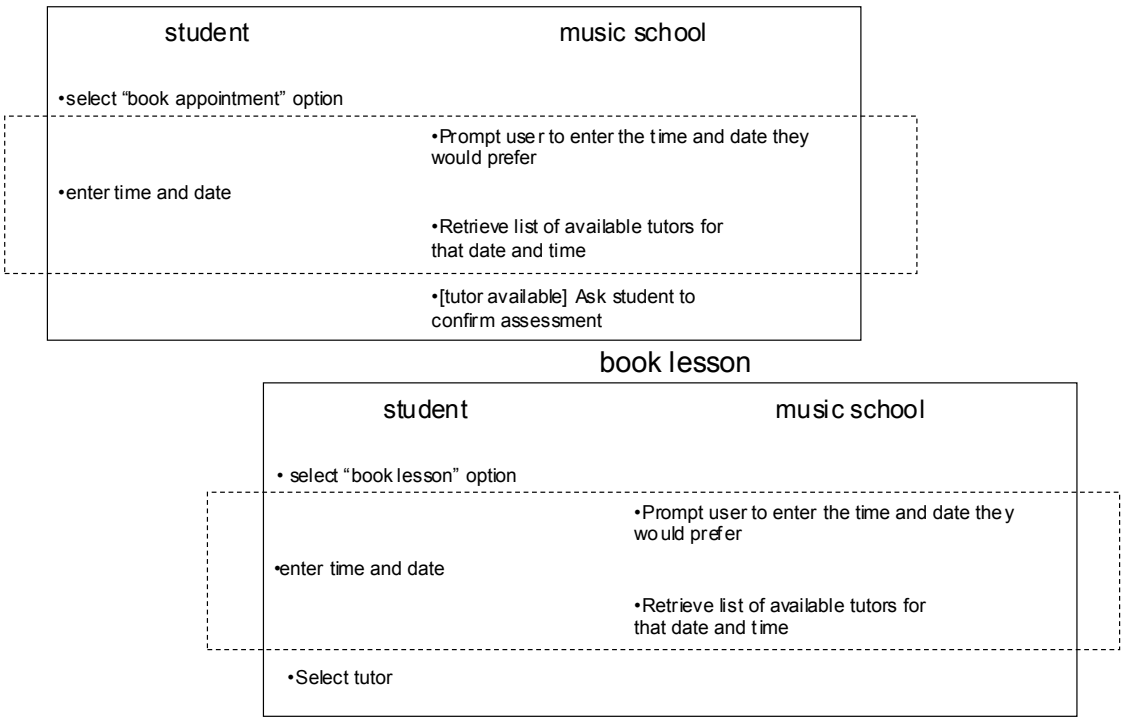

Fig 1.7. Two essential use cases share common steps

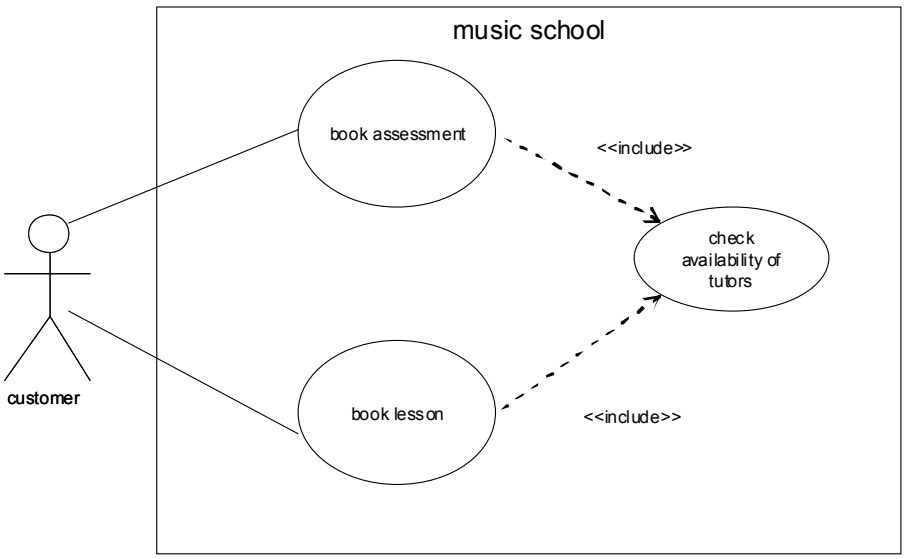

Fig 1.8. UML Use Case diagram showing two use cases including another

Extending Use Cases

Sometimes, two or more use cases will include the flow of another use case, but only under certain conditions. For example, when I make a cup of tea or make a cup of cocoa, I might boil the kettle *only if* it hasn't recently been boiled.

The <<include>> relationship means that the flow of that use case is *always* included. But a **<<extend>>** relationship means that the flow of the extending use case is only included *under specific conditions*, which must be specified as the **extension point** of the use case being extended.

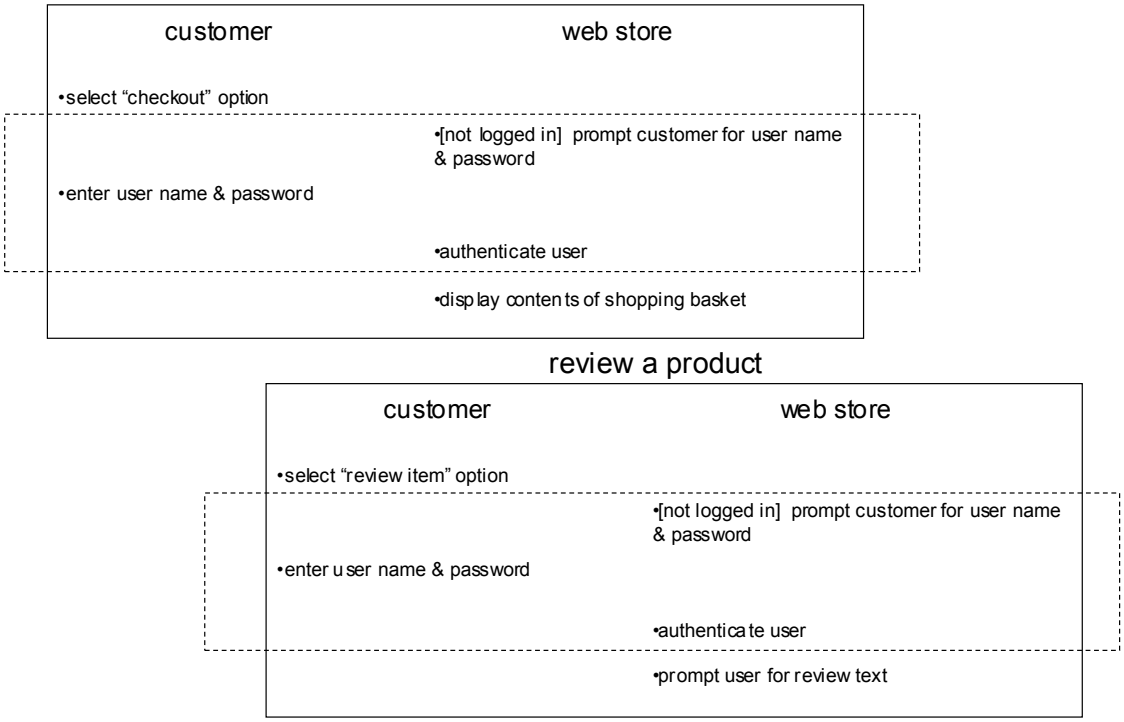

Fig 1.9. The two use case share common steps that are only executed when the actor is not logged in.

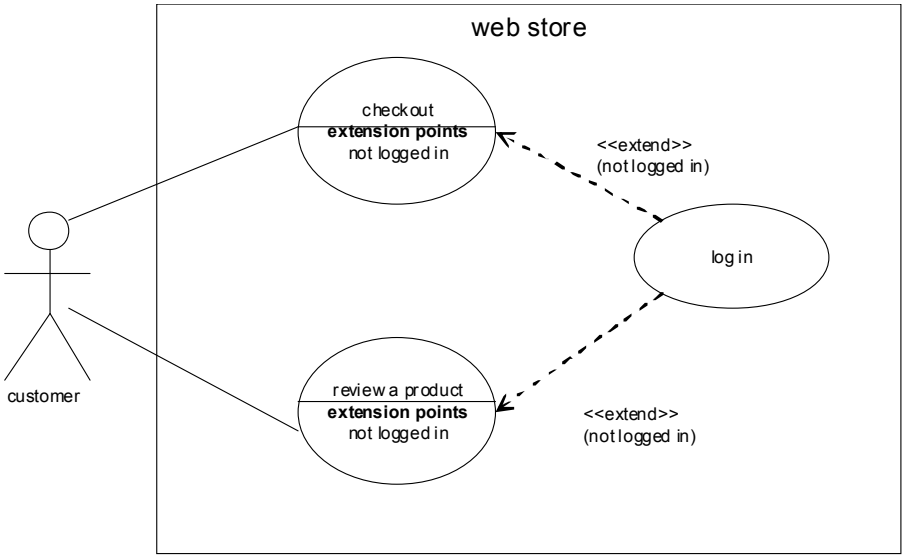

Fig 1.10. A UML Use Case diagram showing two use cases being extended by another

Many people get tripped up by <<include>> and <<extend>> relationships between use cases. Here are some tips for getting it right:

1. Make sure you've got the right kind of relationship: <<include>> means "always included", but <<extend>> means "conditionally included".
2. Make sure you've got the arrows going the right way. <<include>> should point towards the use case being included. <<extend>> should point towards the use case(s) being extended (and not the extending use case).

Applying Use Cases

As I mentioned at the beginning, a heck of a lot has been written about use cases, and not all of it is perhaps as straightforward and commonsense as it could be. Applying use cases in the software development process is not the black art some authors make it out to be.

There are a few simple steps to follow, and you can use your noodle to figure out the rest of the details:

- Identify your actors: who will be using the system?
- Identify their goals: what will they be using the system to do?
- Identify key scenarios: in trying to achieve a specific goal, what distinct outcomes or workflows might we need to consider?
- Describe in business terms the interactions between the actor(s) and the system for a specific scenario

- Create a UI prototype that clearly communicates the scenario to technical and non-technical stakeholders
- Do a high-level OO design for the scenario
- Implement the design in code
- Get feedback from your users – ideally through structured acceptance testing
- Move on to the next scenario or use case ("rinse and repeat")

The most important thing to remember is to do analysis, design and implementation one use case scenario at a time, and to get feedback from the users as soon as a new scenario is ready for them to test. You can incorporate valuable lessons from this feedback and evolve your design into something more useful.

WARNING! Do not, under any circumstances, attempt to design the entire system before writing any code. Break the design down into use cases and scenarios, and work one scenario at a time.

Reusing Use Cases

As I've alluded to already, use cases aren't just useful for capturing functional requirements or doing interaction design. In this final section, we'll take a look at 5 other ways we can exploit use cases in the development process.

Planning & Estimating

Use cases can be used as the basis for planning development projects, and for driving the development process. Use cases can be sized according to their complexity and, working from historical data, estimates can be calculated for how long each use case might take to implement. A practical discussion about the ins and outs of project planning is well beyond the scope of this guide, but if I can only get one point across, it will be this: **use iterative and adaptive planning techniques**

System Testing

Just as a storyboard adds implementation details to an essential use case scenario, so too can system test scripts add test data to the same scenarios. That's all a system test script is, in essence; a use case scenario with specific test data.

User Documentation

Put real screen grabs in your storyboards and add a bit more descriptive text, then add the words "How to…" to the title of each use case, and you have some pretty usable user documentation.

User Stories

If you're into Agile Software Development, and especially eXtreme Programming, you can easily break down your essential use cases into smaller chunks and write them on story cards. A word of advice, though, keeping user stories and use cases in synch is as problematic as keeping models and code in step. My advice is to maintain high-level traceability only: if you can map user stories on to use case names, then that's usually good enough for tracking purposes.

Business Simulations

Use case descriptions can be incorporated into higher-level business process descriptions so that we can simulate the execution of those business processes and sanity check our use cases in the context in which they will be used. This can be a very effective way of avoiding the kind of requirements errors that are very difficult to spot "from the ground".

Conclusion

Despite what you may have heard from some Agile developers, reports of the demise of use cases have been greatly exaggerated. They're by no means a silver bullet for requirements and UI design, and they certainly have their pitfalls, but overall they can be a powerful tool for most projects.

The secret is to *keep it simple*, and to involve the users right the way along in the identification and design of use cases. Remember that our aim is to eliminate rework due to requirements misunderstandings, and so we should be aiming to reach a point where there are no surprises for the users. Use cases, in conjunction with techniques like storyboarding, help to build an explicit shared understanding that everyone can take away with them – users, developers, testers, technical authors, and others.

There's absolutely no reason why use cases can't be applied in Agile projects, provided they're applied in the Agile spirit. Tackle them in small chunks, involve the users closely and seek feedback throughout the project.

Further Reading

Writing Effective Use Cases – Alistair Coburn
http://www.amazon.com/Writing-Effective-Cases-Alistair-Cockburn/dp/0201702258

Applying Use Case-driven Object Modeling with UML – Doug Rosenberg & Kendall Scott
http://www.amazon.com/Applying-Case-Driven-Object-Modeling/dp/0201730391

Use Case Zone – Andy Pols
http://www.pols.co.uk/use-case-zone/index.html

Driving Development with Use Cases – Jason Gorman
http://parlezuml.com/tutorials/usecases/usecases.pdf

Training

Requirements Analysis using UML – 2 days
http://parlezuml.com/training/umlrequirements.htm