Chapter 2. Modeling Requirements: Use Cases

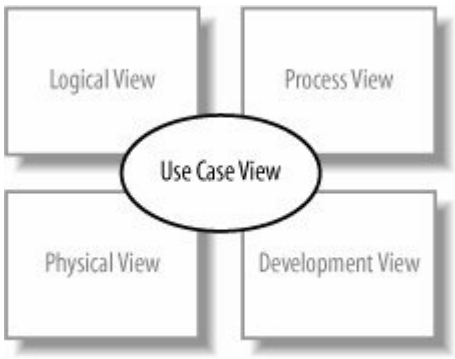
Imagine that it's Monday morning and your first day on a new project. The requirements folks have just popped in for a coffeeand to leave you the 200-page requirements document they've been working on for the past six months. Your boss's instructions are simple: "Get your team up to speed on these requirements so that you can all start designing the system." Happy Monday, huh?

To make things just a bit more difficult, the requirements are still a little fuzzy, and they are all written in the language of the userconfusing and ambiguous natural language rather than in a language that your system stakeholders can easily understand. See the "Verbosity, Ambiguity, Confusion: Modeling with Informal Languages" section in Chapter 1 for more on the problems of modeling with natural and informal languages.

What is the next step, apart from perhaps a moment or two of sheer panic? How do you take this huge set of loosely defined requirements and distill it into a format for your designers without losing important detail? UML, as you know from Chapter 1, is the answer to both of these questions. Specifically, you need to work with your system's stakeholders to generate a full set of requirements and something newuse cases .

A use case is a case (or situation) where your system is used to fulfill one or more of your user's requirements; a use case captures a piece of functionality that the system provides. Use cases are at the heart of your model, shown in Figure 2-1, since they affect and guide all of the other elements within your system's design.

Figure 2-1. Use cases affect every other facet of your system's design; they capture what is required and the other views on your model, then show how those requirements are met



Use cases are an excellent starting point for just about every facet of object-oriented system development, design, testing, and documentation. They describe a system's requirements strictly from the outside looking in; they specify the value that the system delivers to users. Because use cases are your system's functional requirements, they should be the first serious output from your model after a project is started. After all, how can you begin to design a system if you don't know what it will be required to do?

Use cases specify only what your system is supposed to do, i.e., the system's functional requirements. They do not specify what the system shall not do, i.e., the system's nonfunctional requirements. Nonfunctional requirements often include performance targets and programming languages, etc.

When you are working on a system's requirements, questions often arise as to whether the system has a particular requirement. Use cases are a means to bring those gaps in the user's requirements to the forefront at the beginning of a project.

This is a real bonus for the system designer since a gap or lack of understanding identified early on in a project's development will cost far less in both time and money than a problem that is not found until the end of a project. Once a gap has been identified, go back to the system's stakeholdersthe customers and usersso they can provide the missing information.

It's even better when a requirement is presented as a use case and the stakeholder sees that the requirement has little or no value to the system. If a stakeholder can discard unnecessary requirements, both money and time are saved.

Once priority and risk are assigned to a use case, it can help manage a project's workload. Your use cases can be assigned to teams or individuals to be implemented and, since a use case represents tangible user value, you can track the progress of the project by use cases delivered. If and when a project gets into schedule trouble, use cases can be jettisoned or delayed to deliver the highest value soonest.

Last but not least, use cases also help construct tests for your system. Use cases provide an excellent starting point for building your test cases and procedures because they precisely capture a user's requirements and success criteria. What better way to test your system than by using the use cases that originally captured what the user wanted in the first place?

2.1. Capturing a System Requirement

Enough theory for now; let's take a look at a simple example. Suppose we're defining requirements for a weblog content management system (CMS).

Requirement A.1

The content management system shall allow an administrator to create a new blog account, provided the personal details of the new blogger are verified using the author credentials database.

There's actually no specific "best way" to start analyzing Requirement A.1, but one useful first step is to look at the things that interact with your system. In use cases, these external things are called actors .

The terms shall and should have a special and exact meaning when it comes to requirements. A shall requirement must be fulfilled; if the feature that implements a shall requirement is not in the final system, then the system does not meet this requirement. A should requirement implies that the

requirement is not critical to the system working but is still desirable. If a system's development is running into problems that will cause delivery delays, then it's often the should requirements that are sacrificed first.

Blog Features

Weblogs, commonly referred to as blogs, originally started out as privately maintained web pages for authors to write about anything. These days, blogs are usually packaged into an overall CMS. Bloggers submit new entries to the system, administrators allocate blogging accounts, and the systems typically incorporate advanced features, such as RSS feeds. A well-publicized blog can attract thousands of readers (see O'Reilly's blogging site at http://weblogs.oreillynet.com).

2.1.1. Outside Your System: Actors

An actor is drawn in UML notation using either a "stick man" or a stereotyped box (see "Stereotypes" in Chapter 1) and is labeled with an appropriate name, as shown in Figure 2-2.

Figure 2-2 captures the `Administrator` role as it is described in Requirement A.1. The system that is being modeled is the CMS; the requirement's description indicates

Figure 2-2. Requirement A.1 contains an Administrator actor that interacts with the system to create a blog account

that the `Administrator` interacts with the system to create a new blogger's account. The `Administrator` interacts with the system and is not part of the system; therefore, the `Administrator` is defined as an actor.

What's in a Name?

It's actually worth being very careful when naming your actors. The best approach is to use a name that can be understood by both your customer and your system designers. Wherever possible, use the original term for the actor as identified within your customer's requirements; that way, at least your use cases will be familiar to your customers. This approach also lets system designers get comfortable with the system's unique context.

Deciding what is and what is not an actor is tricky and is something best learned by experience. Until you've gained some of that experience, Figure 2-3 shows a simple technique for analyzing a "thing" that you've found in your requirements and how to decide whether it is an actor or not.

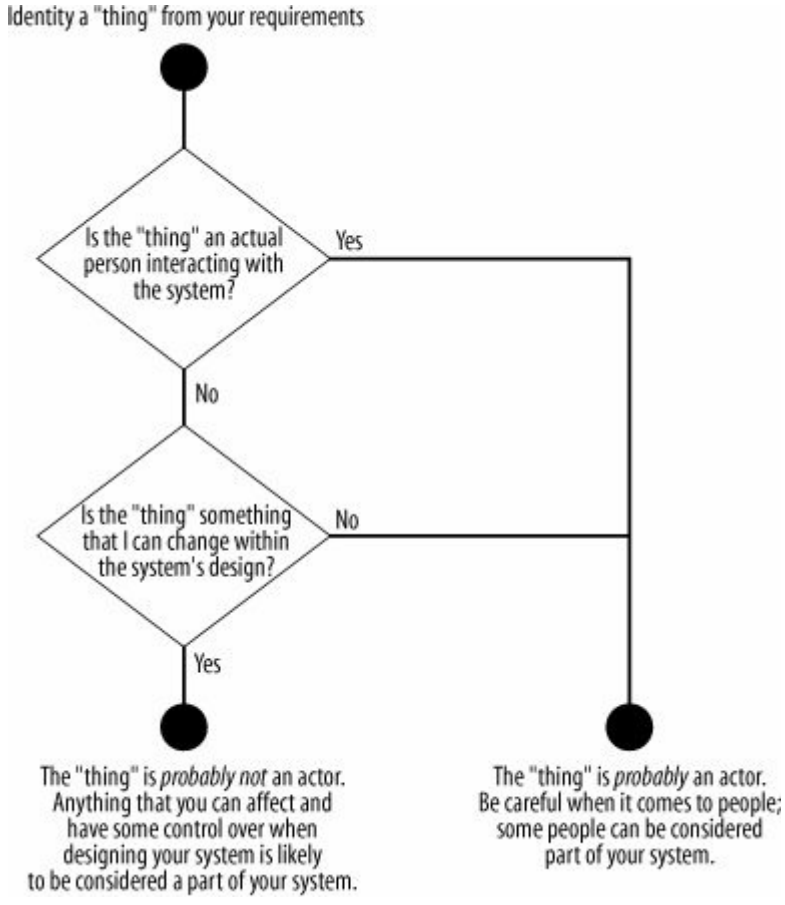
Actors don't have to be actual people. While an actor might be a person, it could also be a third party's system, such as in a business-to-business (B2B) application. Think of an actor as a black box: you cannot change an actor and you are not interested in how it works, but it must interact with your system.

2.1.1.1. Tricky actors

Not all actors are obvious external systems or people that interact with your system. An example of a common tricky actor is the system clock. The name alone implies that the clock is part of the system, but is it really?

The system clock comes into play when it invokes some behavior within your system. It is hard to determine whether the system clock is an actor because the clock is not clearly outside of your system. As it turns out, the system clock is often best described as an actor because it is not something that you can influence. Additionally, describing the clock as an actor will help when demonstrating that your system needs to perform a task based on the current time.

Figure 2-3. Here are a couple of questions to ask yourself when trying to identify an actor



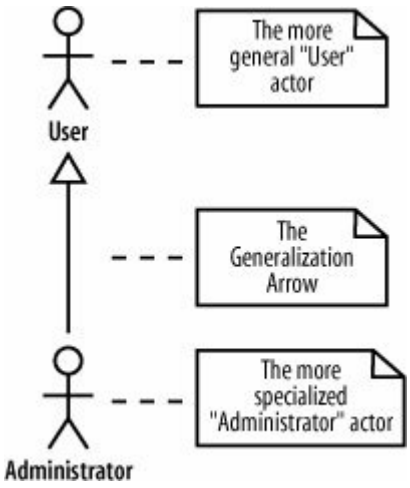
It is also tempting to focus on just the users of your systems as the actors in your model, but don't forget about other people, such as auditors, installers, maintainers, upgraders, and so on. If you focus on only the obvious users of your system, then you might forget about some of these other stakeholders, and that can be very dangerous! Those actors may have a veto ("We can't certify this system without proof that the data has not been tampered with") or they may have to enforce important nonfunctional requirements, such as an upgrade in a 10-minute system downtime window and an upgrade without shutting the system down, etc. If these actors are ignored, these important functions of your system won't be documented, and you risk ending up with a worthless system.

2.1.1.2. Refining actors

When going through the process of capturing all of the actors that interact with your system, you will find that some actors are related to each other, as shown in Figure 2-4.

The `Administrator` actor is really a special kind of system user. To show that an administrator can do whatever a regular user can do (with some extra additions), a generalization arrow is used. For more on generalization and the generalization arrow, see Chapter 5.

Figure 2-4. Showing that an administrator is a special kind of user



2.1.2. Use Cases

Once you have captured an initial set of actors that interact with your system, you can assemble the exact model of those interactions. The next step is to find cases where the system is being used to complete a specific job for an actoruse cases, in fact. Use cases can be identified from your user's requirements. This is where those wordy, blurry definitions in the user requirements document should be distilled into a clear set of jobs for your system.

Remember, if use cases are truly requirements, then they must have very clear pass/fail criteria. The developer, the tester, the technical writer, and the user must explicitly know whether the system fulfils the use case or not.

A use case, or job, might be as simple as allowing the user to log in or as complex as executing a distributed transaction across multiple global databases. The important thing to remember is that a use casefrom the user's perspectiveis a complete use of the system; there is some interaction with the system, as well as some output from that interaction. For example, Requirement A.1 describes one main use of the CMS: to create a new blog account. Figure 2-5 shows how this interaction is captured as a use case.

Figure 2-5. A use case in UML is drawn as an oval with a name that describes the interaction that it represents

After all that build-up, you might have expected a use case to be a complex piece of notation. Instead, all you get is an oval! The notation for a use case is very simple and often hides its importance in capturing system concerns. Don't be deceived; the use case is probably the single most powerful construct in UML to make sure your system does what it is supposed to.

What Makes a Good Use Case?

Experience will help you determine when you have a good use case, but there is a rule of thumb that can be used to specify a use case:

A use case is something that provides some measurable result to the user or an external system.

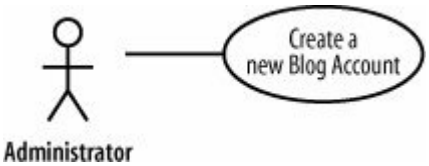
Any piece of system behavior that meets this simple test is likely to be a good candidate for a use case.

2.1.3. Communication Lines

At this point, we've identified a use case and an actor, but how do we show that the `Administrator` actor participates in the `Create a new Blog Account` use case? The answer is by using communication lines .

A communication line connects an actor and a use case to show the actor participating in the use case. In this example, the `Administrator` actor is involved in the `Create` a `new Blog Account` use case; this is shown in Figure 2-6 by adding a communication line.

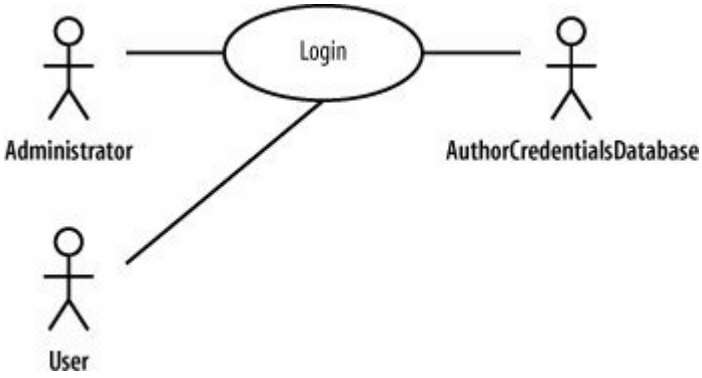
Figure 2-6. A communication line joins the Administrator actor to the "Create a new Blog Account" use case; the Administrator is involved in the interaction that the use case represents



This simple example shows a communication line between only one actor and only one use case. There is potential to have any number of actors involved in a use case. There is no theoretical limit to the number of actors that can participate in a use case.

To show a collection of actors participating in a use case, all you have to do is draw a communication line from each of the participating actors to the use case oval, as shown in Figure 2-7.

Figure 2-7. The login use case interacts with three actors during its execution



Sometimes UML diagrams will have communication lines with navigability; for example, a diagram with an arrow at one end will show the flow of information between the actor and the use case, or show who starts the use case. Although this notation is not really a crime in UML terms, it's not a very good use of communication lines.

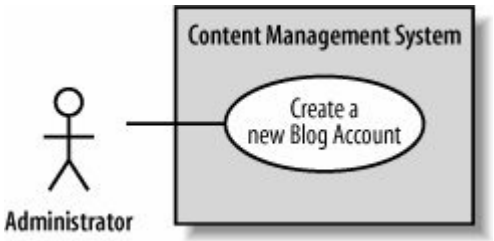
The purpose of a communication line is to show that an actor is simply involved in a use case, not to imply an information exchange in any particular direction or that the actor starts the use case. That type of information is contained within a use case's detailed description, therefore it doesn't make sense to apply navigation to communication lines. For more on use cases and descriptions, see "Use Case Descriptions," later in this chapter.

2.1.4. System Boundaries

Although there is an implicit separation between actors (external to your system) and use cases (internal to your system) that marks your system's boundary, UML does provide another small piece of notation if you want to make things crystal clear.

To show your system's boundary on a use case diagram, draw a box around all of the use cases but keep the actors outside of the box. It's also good practice to name your box after the system you are developing, as shown for the CMS in Figure 2-8.

Figure 2-8. The Administrator actor is located outside of the CMS, explicitly showning that the system boundary box use cases must fall within the system boundary box, since it doesn't make sense to have a use case outside of your system's boundary



2.1.5. Use Case Descriptions

A diagram showing your use cases and actors may be a nice starting point, but it does not provide enough detail for your system designers to actually understand exactly how the system's concerns will be met. How can a system designer understand who the most important actor is from the use case notation alone? What steps are involved in the use case? The best way to express this important information is in the form of a text-based descriptionevery use case should be accompanied by one.

There are no hard and fast rules as to what exactly goes into a use case description according to UML, but some example types of information are shown in Table 2-1.

Table 2-1. Some types of information that you can include in your use case descriptions

Use case description detail	What the detail means and why it is useful
Related Requirements	Some indication as to which requirements this use case partially or completely fulfills.
Goal In Context	The use case's place within the system and why this use case is important.
Preconditions	What needs to happen before the use case can be executed.
Successful End Condition	What the system's condition should be if the use case executes successfully.
Failed End Condition	What the system's condition should be if the use case fails to execute successfully.
Primary Actors	The main actors that participate in the use case. Often includes the actors that trigger or directly receive information from a use case's execution.
Secondary Actors	Actors that participate but are not the main players in a use case's execution.
Trigger	The event triggered by an actor that causes the use case to execute.
Main Flow	The place to describe each of the important steps in a use case's normal execution.
Extensions	A description of any alternative steps from the ones described in the Main Flow.

Table 2-2 shows an example use case description for the `Create a new Blog Account` use case and provides a handy template for your own descriptions.

Table 2-2. A complete use case description for the "Create a new Blog Account" use case

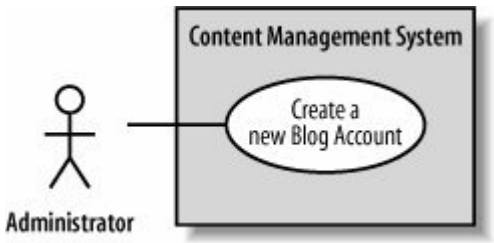
Use case name	Create a new Blog Account	
Related Requirements	Requirement A.1.	
Goal In Context	A new or existing author requests a new blog account from the `Administrator`.	
Preconditions	The system is limited to recognized authors and so the author needs to have appropriate proof of identity.	
Successful End Condition	A new blog account is created for the author.	
Failed End Condition	The application for a new blog account is rejected.	
Primary Actors	Administrator.	
	Secondary Actors	Author Credentials Database.

	Trigger	The `Administrator` asks the CMS to create `a new blog account`.
Main Flow	Step	Action
	1	The `Administrator` asks the system to create a new blog account.
	2	The `Administrator` selects an account type.
	3	The `Administrator` enters the author's details.
	4	The author's details are verified using the Author Credentials Database.
	5	The new blog account is created.
	6	A summary of the new blog account's details are emailed to the author.
Extensions	Step	Branching Action
	4.1	The Author Credentials `Database` does not verify the author's details.
	4.2	The author's new blog account application is rejected.

The format and content in Table 2-2 is only an example, but it's worth remembering that use case descriptions and the information that they contain are more than just extra information to accompany the use case diagrams. In fact, a use case's description completes the use case; without a description a use case is, well, not very useful.

The description in Table 2-2 was reasonably straightforward, but something's not quite right when you compare the description to the original use case diagram (shown in Figure 2-9; although the use case description mentions two actors, this use case diagram shows only one).

Figure 2-9. Ensuring that your use case diagrams match the more detailed use case descriptions is critical



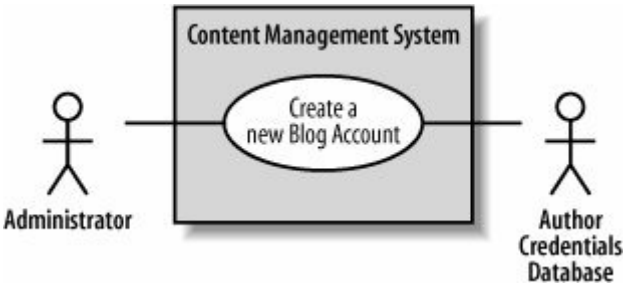
The use case description has identified a new actor, the `Author Credentials Database`. By creating a complete description of the `Create a new Blog Account` use case, it becomes clear that this actor is missing.

If you can, it's worth reviewing your use case model with your users as much as possible to ensure that you have captured all of the key uses of your system and that nothing has been missed.

You will often find that items are missing from your diagrams as more detail goes into your use case descriptions. The same goes for any aspect of your model: the more detail you put in, the more you might have to go back and correct what you did before. This is what iterative system development is all about. Don't be too worried though, this refinement of your model is a good thing. With each iteration of development you will (hopefully!) get a better and more accurate model of your system.

Figure 2-10 shows the corrected use case diagram incorporating the new `Author Credentials Database` actor.

Figure 2-10. Bring the use case diagram in sync with the use case's description by adding the Author Credentials Database actor



How Many Use Cases Should Your Model Have?

There is no set rule for the number of use cases that your use case model should contain for a given system. The number of use cases depends on the of the jobs that your system has to do according to the requirements. This means that for a particular system, you might only need two use cases or you might need hundreds.

It is more important that you have the right use cases, rather than worrying about the amount you have. As with most things in system modeling, the best way to get your use cases right is to get used to applying them; experience will teach you what is right for your own systems.

2.2. Use Case Relationships

A use case describes the way your system behaves to meet a requirement. When filling out your use case descriptions, you will notice that there is some similarity between steps in different use cases. You may also find that some use cases work in several different modes or special cases. Finally, you may also find a use case with multiple flows throughout its execution, and it would be good to show those important optional cases on your use case diagrams.

Wouldn't it be great if you could get rid of the repetition between use case descriptions and show important optional flows right on your use case diagrams? OK, so that was a loaded question. You can show reusable, optional, and even specialized use case behavior between use cases.

2.2.1. The <<include>> Relationship

So far, you have seen that use cases typically work with actors to capture a requirement. Relationships between use cases are more about breaking your system's behavior into manageable chunks than adding anything new to your system. The purpose of use case relationships is to provide your system's designers with some architectural guidance so they can efficiently break down the system's concerns into manageable pieces within the detailed system design.

In addition to blogs, a CMS can have any number of means for working with its content. One popular mechanism for maintaining documents is by creating a Wiki. Wikis allow online authors to create, edit, and link together web pages to create a web of related content, or a Wiki-web. A great example of a Wiki is available at http://www.Wikipedia.org.

Take another look at the `Create a new Blog Account` use case description shown in Table 2-2. The description seems simple enough, but suppose another requirement is added to the `Content Management System`.

Requirement A.2

The content management system shall allow an administrator to create a new personal Wiki, provided the personal details of the applying author are verified using the Author Credentials Database.

To capture Requirement A.2 a new use case needs to be added to the `Content Management System`, as shown in Figure 2-11.

Now that we have added the new use case to our model, it's time to fill out a detailed use case description (shown in Table 2-3). See Table 2-1 if you need to refresh your memory about the meaning of each of the details within a use case description.

Figure 2-11. A new requirement can often mean a new use case for the system, although it's not always a one-to-one mapping

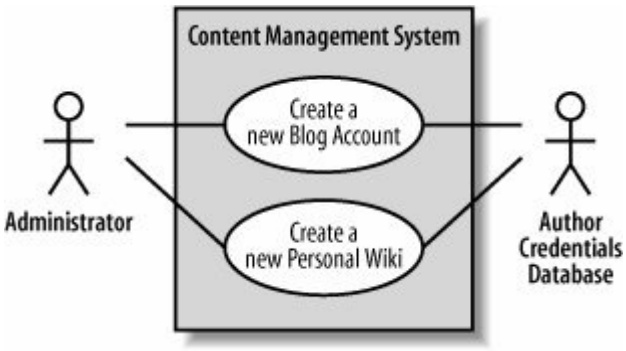


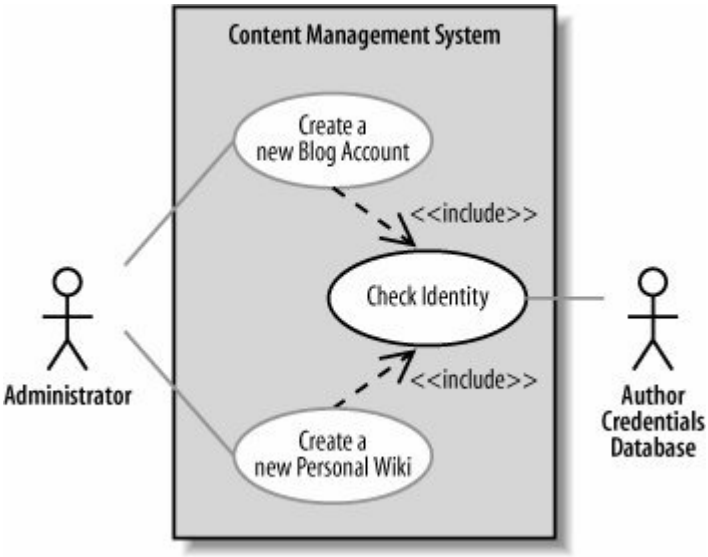
Table 2-3. The detailed description for the "Create a new Personal Wiki" use case

Use case name	**Create a new Personal Wiki**	
Related Requirements	Requirement A.2.	
Goal In Context	A new or existing author requests a new personal Wiki from the Administrator.	
Preconditions	The author has appropriate proof of identity.	
Successful End Condition	A new personal Wiki is created for the author.	
Failed End Condition	The application for a new personal Wiki is rejected.	
Primary Actors	Administrator.	
Secondary Actors	Author Credentials Database.	
Trigger	The Administrator asks the CMS to create a new personal Wiki.	
Main Flow	Step	Action
	1	The Administrator asks the system to create a new personal Wiki.
	2	The Administrator enters the author's details.
	3	The author's details are verified using the Author Credentials Database.
	4	The new personal Wiki is created.
	5	A summary of the new personal Wiki's details are emailed to the author.
Extensions	Step	Branching Action
	3.1	The Author Credentials Database does not verify the author's details.
	3.2	The author's new personal Wiki application is rejected.

The first thing to notice is that we have some redundancy between the two use case descriptions (Tables 2-2 and 2-3). Both `Create a new Blog Account` and `Create a new Personal Wiki` need to check the applicant's credentials. Currently, this behavior is simply repeated between the two use case descriptions.

This repetitive behavior shared between two use cases is best separated and captured within a totally new use case. This new use case can then be reused by the `Create a new Blog Account` and `Create a new Personal Wiki` use cases using the `<<include>>` relationship (as shown in Figure 2-12).

Figure 2-12. The <<include>> relationship supports reuse between use cases



The `<<include>>` relationship declares that the use case at the head of the dotted arrow completely reuses all of the steps from the use case being included. In Figure 2-12, the `Create a new Blog Account` and `Create a new Personal Wiki` completely reuse all of the steps declared in the `Check Identity` use case.

You can also see in Figure 2-12 that the `Check Identity` use case is not directly connected to the `Administrator` actor; it picks this connection up from the use cases that include it. However, the connection to the `Author Credentials Database` is now solely owned by the `Check Identity` use case. A benefit of this change is that it emphasizes that the `Check Identity` use case is the only one that relies directly on a connection to the `Author Contact Details Database` actor.

To show the `<<include>>` relationship in your use case descriptions, you need to remove the redundant steps from the `Create a new Blog Account` and `Create new Personal Wiki` use case descriptions and instead use the `Included Cases` field and `include::<use case name>` syntax to indicate the use case where the reused steps reside, as shown in Tables 2-4 and 2-5.

Table 2-4. Showing <<include>> in a use case description using Included Cases and include::<use case name>

Use case name	**Create a new Blog Account**	
Related Requirements	Requirement A.1.	
Goal In Context	A new or existing author requests a new blog account from the Administrator.	
Preconditions	The author has appropriate proof of identity.	
Successful End Condition	A new blog account is created for the author.	
Failed End Condition	The application for a new blog account is rejected.	
Primary Actors	Administrator	
Secondary Actors	None	
Trigger	The Administrator asks the CMS to create a new blog account.	
Included Cases	Check Identity	
Main Flow	Step	Action

	1	The Administrator asks the system to create a new blog account.
	2	The Administrator selects an account type.
	3	The Administrator enters the author's details.
	4 include::Check Identity	The author's details are checked.
	5	The new account is created.
	6	A summary of the new blog account's details are emailed to the author.

Table 2-5. The Create a new Personal Wiki use case description also gets a makeover

Use case name	**Create a new Personal Wiki**	
Related Requirements	Requirement A.2	
Goal In Context	A new or existing author requests a new personal Wiki from the Administrator.	
Preconditions	The author has appropriate proof of identity.	
Successful End Condition	A new personal Wiki is created for the author.	
Failed End Condition	The application for a new personal Wiki is rejected.	
Primary Actors	Administrator	
Secondary Actors	None	
Trigger	The Administrator asks the CMS to create a new personal Wiki.	
Included Cases	Check Identity	
Main Flow	Step	Action
	1	The Administrator asks the system to create a new personal Wiki.
	2	The Administrator enters the author's details.
	3 include::Check Identity	The author's details are checked.
	5	The new personal Wiki is created.
	6	A summary of the new personal Wiki's details are emailed to the author.

Now you can create a use case description for the reusable steps within the `Check Identity` use case, as shown in Table 2-6.

Table 2-6. The Check Identity use case description contains the reusable steps

Use case name	**Check Identity**	
Related Requirements	Requirement A.1, Requirement A.2.	
Goal In Context	An author's details need to be checked and verified as accurate.	
Preconditions	The author being checked has appropriate proof of identity.	
Successful End Condition	The details are verified.	
Failed End Condition	The details are not verified.	
Primary Actors	Author Credentials Database.	
Secondary Actors	None.	
Trigger	An author's credentials are provided to the system for verification.	
Main Flow	Step	Action
	1	The details are provided to the system.
	2	The Author Credentials Database verifies the details.
	3	The details are returned as verified by the Author Credentials Database.
Extensions	Step	Branching Action
	2.1	The Author Credentials Database does not verify the details.
	2.2	The details are returned as unverified.

Why bother with all this hassle with reuse between use cases? Why not just have two use cases and maintain the similar steps separately? All this reuse has two important benefits:

- Reuse using `<<include>>` removes the need for tedious cut-and-paste operations between use case descriptions, since updates are made in only one place instead of every use case.

- The `<<include>>` relationship gives you a good indication at system design time that the implementation of `Check Identity` will need to be a reusable part of your system.

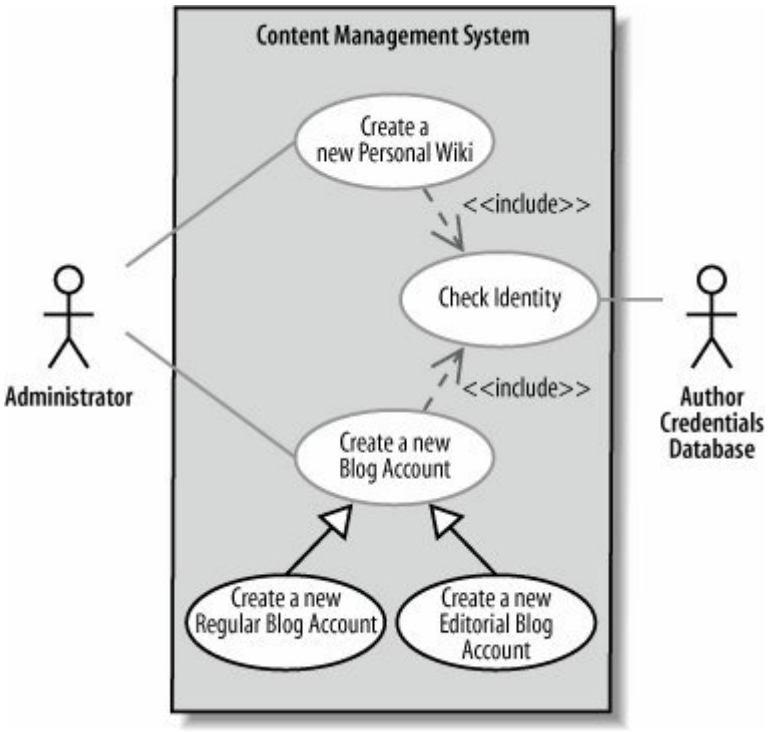
2.2.2. Special Cases

Sometimes you'll come across a use case whose behavior, when you start to analyze it more carefully, can be applied to several different cases, but with small changes. Unlike the `<<include>>` relationship, which allows you to reuse a small subset of behavior, this is applying a use case with small changes for a collection of specific situations. In object-oriented terms, you potentially have a number of specialized cases of a generalized use case.

Let's take a look at an example. Currently, the `Content Management System` contains a single `Create a new Blog Account` use case that describes the steps required to create an account. But what if the CMS supports several different types of blog accounts, and the steps required to create each of these accounts differs ever so slightly from the original use case? You want to describe the general behavior for creating a blog accountcaptured in the `Create a new Blog Account` use caseand then define specialized use cases in which the account being created is a specific type, such as a regular account with one blog or an editorial account that can make changes to entries in a set of blogs.

This is where use case generalization comes in. A more common way of referring to generalization is using the term inheritance . Use case inheritance is useful when you want to show that one use case is a special type of another use case. To show use case inheritance , use the generalization arrow to connect the more general, or parent, use case to the more specific use case. Figure 2-13 shows how you could extend the CMS's use cases to show that two different types of blog accounts can be created.

Figure 2-13. Two types of blog account, regular and editorial, can be created by the Management System



Taking a closer look at the `Create a new Editorial Blog Account` specialized use case description, you can see how most of the behavior from the more general `Create a new Blog Account` use case is reused. Only the details that are specific to creating a new editorial account need to be added (see Table 2-7).

Table 2-7. You can show that a use case is a special case of a more general use case within the detailed description using the Base Use Cases field

Use case name	**Create a new Editorial Blog Account**
Related Requirements	Requirement A.1.
Goal In Context	A new or existing author requests a new editorial blog account from the Administrator .
Preconditions	The author has appropriate proof of identity.
Successful End Condition	A new editorial blog account is created for the author.
Failed End Condition	The application for a new editorial blog account is rejected.

Primary Actors	Administrator.	
Secondary Actors	None.	
Trigger	The Administrator asks the CMS to create a new editorial account that will allow an author to edit entries in a set of blogs.	
Base Use Cases	Create a new Blog Account	
Main Flow	Step	Action
	1	The Administrator asks the system to create a new blog account.
	2	The Administrator selects the editorial account type.
	3	The Administrator enters the author's details.
	4	The Administrator selects the blogs that the account is to have editorial rights over.
	5 include::Check Identity	The author's details are checked.
	6	The new editorial account is created.
	7	A summary of the new editorial account's details are emailed to the author.
Extensions	Step	Branching Action
	5.1	The author is not allowed to edit the indicated blogs.
	5.2	The editorial blog account application is rejected.
	5.3	The application rejection is recorded as part of the author's history.

Use case inheritance is a powerful way of reusing a use case so that you only have to specify the extra steps that are needed in the more specific use cases. See Chapter 5 for more information on inheritance between classes.

But be carefulby using inheritance, you are effectively saying that every step in the general use case must occur in the specialized use cases. Also, every relationship that the general use case has with external actors or use cases, as shown with the `<<include>>` relationship between `Create a new Blog Account` and `Check Identity`, must also make sense in the more specialized use cases, such as `Create a new Editorial Blog Account`.

If you really don't want your more specific use case to do everything that the general use case describes, then don't use generalization. Instead, you might want to consider using either the `<<include>>` relationship shown in the previous section or the `<<extend>>` relationship coming up in the next section.

2.2.3. The <<extend>> Relationship

Any explanation of the `<<extend>>` stereotype should be preceded by a warning that it is the most heavily debated type of use case relationship. Almost nothing is less understood or harder to

accurately communicate within the UML modeling community than the <<extend>> use case relationship, and this presents a bit of a problem when you are trying to learn about it. Figure 2-14 shows you how <<extend>> works; take a look, and then let's dive into some UML concept and theory.

Figure 2-14. The <<extend>> use case relationship looks a bit like the <<include>> relationship, but that's where the similarities end

At first glanceparticularly if you are a Java programmer<<extend>> seems very similar to inheritance between classes. In Java, a class can extend from a base class. Similarly, in C++ and C#, you can declare inheritance between classes, and you would often say that a class extends another class. In both these cases, the extend relationship between classes means inheritance. So, for a programmer, it follows that <<extend>> should mean something like inheritance, right?

Alarm bells should definitely be going off now. You already saw in the previous section how use cases declare inheritance using a generalization arrow, so why would you need yet another type of arrow with an <<extend>> stereotype? Does the generalization arrow mean the same thing as the <<extend>> stereotype? Unfortunately, the <<extend>> stereotype has very little in common with inheritance, and so the two definitely do not mean the same thing.

The designers of UML 2.0 took a very different view as to the meaning of <<extend>> between use cases. They wanted a means for you to show that a use case might completely reuse another use case's behavior, similar to the <<include>> relationship, but that this reuse was optional and dependent either on a runtime or system implementation decision.

From the CMS example, the `Create a new Blog Account` use case might want to record that a new author applied for an account and was rejected, adding this information to the author's application history. Extra steps can be added to the `Create a new Blog Account` use case's description to show this optional behavior, as shown in Step 4.3 in Table 2-8.

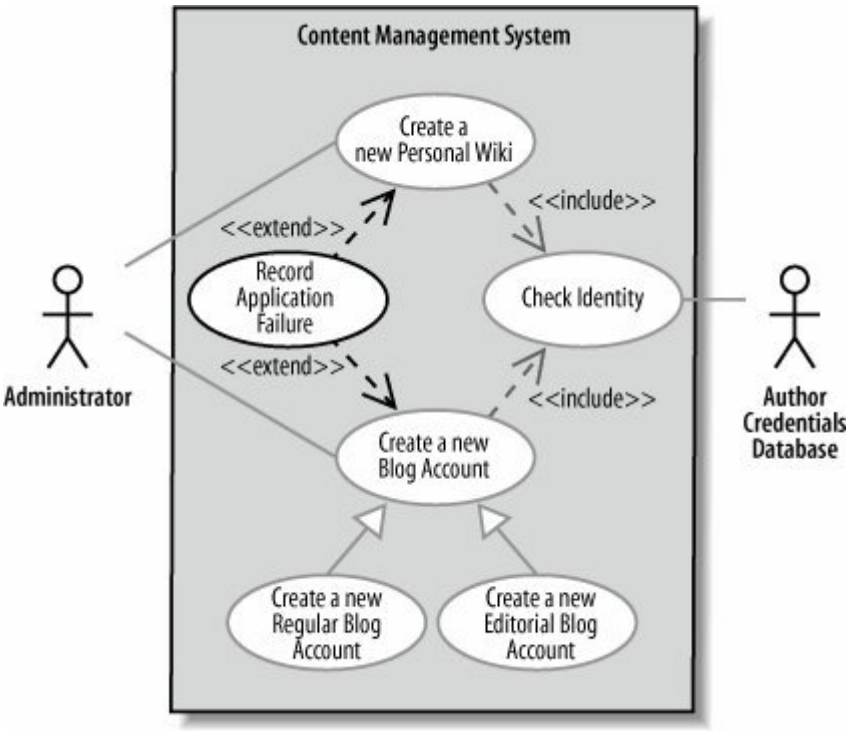
Table 2-8. Behavior that is a candidate for <<extend>> relationship reuse can usually be found in the Extensions section of a use case description

Use case name	**Create a new Blog Account**
Related Requirements	Requirement A.1.
Goal In Context	A new or existing author requests a new blog account from the Administrator.
Preconditions	The author has appropriate proof of identity.
Successful End Condition	A new blog account is created for the author.
Failed End Condition	The application for a new blog account is rejected.
Primary Actors	Administrator.
Secondary Actors	None.
Trigger	The Administrator asks the CMS to create a new blog account.
Included Cases	Check Identity

Main Flow	Step	Action
	1	The Administrator asks the system to create a new blog account.
	2	The Administrator selects an account type.
	3	The Administrator enters the author's details.
	4 include::Check Identity	The author's details are checked.
	5	The new account is created.
	6	A summary of the new blog account's details are emailed to the author.
Extensions	Step	Branching Action
	4.1	The author is not allowed to create a new blog.
	4.2	The blog account application is rejected.
	4.3	The application rejection is recorded as part of the author's history.

The same behavior captured in Step 4.3 would also be useful if the customer was refused an account for some reason during the `Create a new Personal Wiki` use case's execution. According to the requirements, this reusable behavior is optional in both cases; you don't want to record a rejection if the application for a blog account or a personal Wiki was accepted. The `<<extend>>` relationship is ideal in this sort of reuse situation, as shown in Figure 2-15.

Figure 2-15. The <<extend>> relationship comes into play to show that both the "Create a new Personal Wiki" and "Create a new Blog Account" use cases might occasionally share the application rejection recording behavior

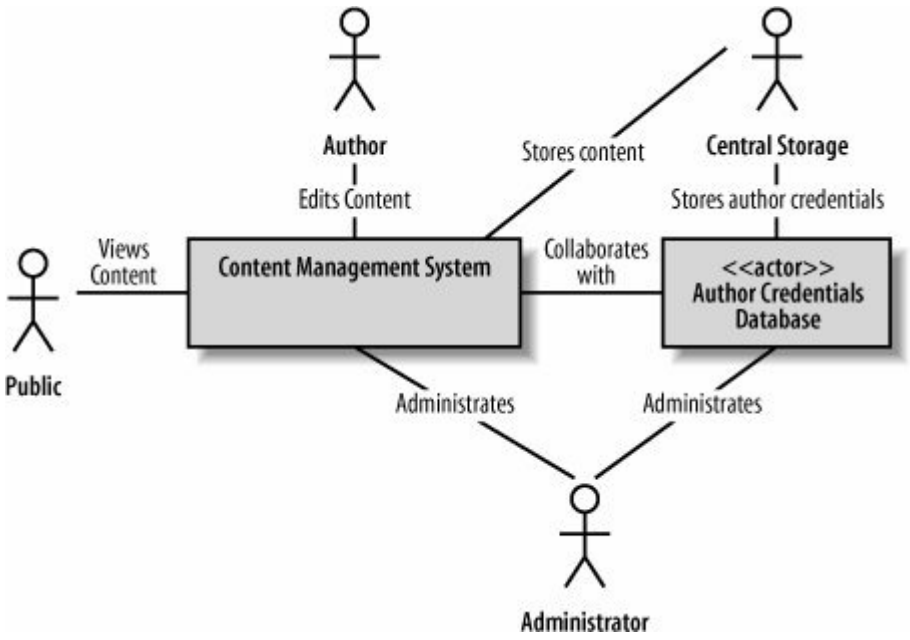


The new `Record Application Failure` use case, as the name implies, captures all of the behavior associated with recording an author's application failure whether it be for a personal Wiki or for a specific type of blog account. Using the `<<extend>>` relationship, the `Record Application Failure` use case's behavior is optionally reused by the `Create a new Blog Account` and `Create a new Personal Wiki` use cases if an application is rejected.

2.3. Use Case Overview Diagrams

When you are trying to understand a system, it is sometimes useful to get a glimpse of the context within which it sits. For this purpose, UML provides the Use Case Overview diagram. Use Case Overview diagrams give you an opportunity to paint a broad picture of your system's context or domain (see Figure 2-16 for an example).

Figure 2-16. The CMS's context as shown on a Use Case Overview diagram



Unfortunately, Use Case Overviews are badly named as they don't usually contain any use cases. The use cases are not shown because the overview is designed to provide a context to your system; the system's internalscaptured by use casesare not normally visible.

Use Case Overviews are a useful place to show any extra snippets of information when understanding your system's place within the world. Those snippets often include relationships and communication lines between actors. These contextual pieces of information do not usually contain a great deal of detail, they are more a placeholder and starting point to for the rest of your model's detail.

2.4. What's Next?

Although this book, like UML, does not push any particular system development process, there are some common steps that are taken after the first cut of use cases are captured.

With your use case model in hand, it is often a good time to start delving into the high-level activities that your system will have to execute to fulfill its use cases. See Chapter 3 for information on activity diagrams.

Once you have a good grip on the high-level activities, look at the classes and components that will actually make up the parts of your system. You already might have some idea of what those classes contain, and so the next stop naturally would be to create a few rudimentary class diagrams. See Chapter 4 for information on class diagrams.

Regardless of your next step, just because you have a use case model does not necessarily mean that you are finished with use cases altogether. The only constant in life is change, and this certainly applies to your system's requirements. As a requirement changeseither because some new system constraint has been found or because a user has changed his mindyou need to go back and refine your use cases to make sure you are still developing the system that the users want.