Chapter 4. Use-Case Diagrams

This chapter focuses on use-case diagrams, which depict the functionality of a system. First, I introduce use-case diagrams and how they are used. Next, I discuss actors and use cases. Finally, I go over various relationships relating to actors and use cases. Many details that were not fleshed out in Chapter 2 are more fully elaborated here, and throughout the chapter I include suggestions relating to use-case diagrams.

Use-case modeling is a specialized type of structural modeling concerned with modeling the functionality of a system. You usually apply use-case modeling during requirements activities to capture requirements that define what a system should do. Use-case modeling typically starts early in a project and continues throughout a system development process. It is usually done as a series of workshops between users and analysts of a system in which ideas can be explored and requirements may be fleshed out over time.

As a use-case driven process uses use cases to plan and perform iterations, it is important to understand how use cases are related to one another, including what use cases have in common, which use cases are options of other use cases, and which use cases are similar to each other. Given that every project has limited resources, you can use this information about use cases to determine how best to execute a project. Use cases that are common to two or more other use cases need only be implemented once, and then they can be reused. Use cases that are options of other use cases may be implemented at a later time, and use cases that are similar allow us to implement one use case that may be reused. An understanding of how use cases are related allows users and analysts to negotiate and reach agreement concerning the scope and requirements of a system.

A use-case diagram may be considered a “table of contents” for the functional requirements of a system. The details behind each element on the use case diagram may be captured in textual form or using other UML modeling techniques. All the use-case diagrams and their associated details for a specific system form the functional requirements of the system. However, the UML does not provide any explicit guidance on how to capture the textual details, but focuses more on the notation.

Actors

As discussed in Chapter 2, an *actor* is a user or external system with which a system being modeled interacts. For example, our project management system involves various types of users, including project managers, resource managers, human resources, and system administrators. These users are all actors.

Actors follow the type-instance dichotomy first discussed in Chapter 2 and applied to classes and objects in Chapter 3. You can use the UML to talk about classes of actors, as well as specific actors of a class. When speaking of a class of actors, it’s customary to use the terms actor or *actor class*. Thus, while you might think of an actor as a specific thing, in the UML, an

actor really represents a class of things. When speaking of a specific actor of a class, use the term *actor instance*.

An actor is external to a system, interacts with the system, may be a human user or another system, and has goals and responsibilities to satisfy in interacting with the system. Actors address the question of who and what interacts with a system. In the UML, an actor is shown as a "stick figure" icon, or as a class marked with the `actor` keyword and labeled with the name of the actor class.

Figure 4-1 shows various actors associated with the project management system:

A project manager

> Responsible for ensuring that a project delivers a quality product within specified time and cost, and within specified resource constraints

A resource manager

> Responsible for ensuring that trained and skilled human resources are available for projects

A human resource

> Responsible for ensuring that worker skills are maintained, and that quality work is completed for a project

A system administrator

> Responsible for ensuring that a project management system is available for a project

A backup system

> Responsible for housing backup data for a project management system

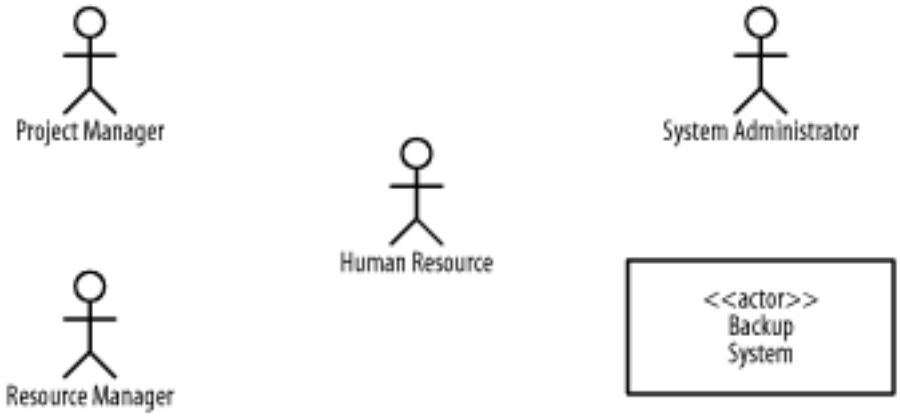

Figure 4-1. Actors

An actor instance is a specific user or system. For example, specific users of the project management system include Jonathan, Andy, Si, Phillip, Nora, and so forth. An actor instance is shown similar to an actor class, but is labeled with the actor instance name followed by a colon followed by the actor class name, all fully underlined. Both names are optional, and the colon is only present if the actor class name is specified. Actor instances address the question of who and what specifically interacts with a system.

Figure 4-2 shows various actor instances of the actor classes in Figure 4-1, including Jonathan and Andy who are project managers, Si who is a human resource, Phillip who is a system administrator, Nora who is an unspecified type of actor instance, a backup system named BackupSys1.0, and other actor instances. Just as it's possible to have an actor of an unspecified type, such as Nora, it is possible to have actors such as `HumanResource` for which a type is specified, but not a name.

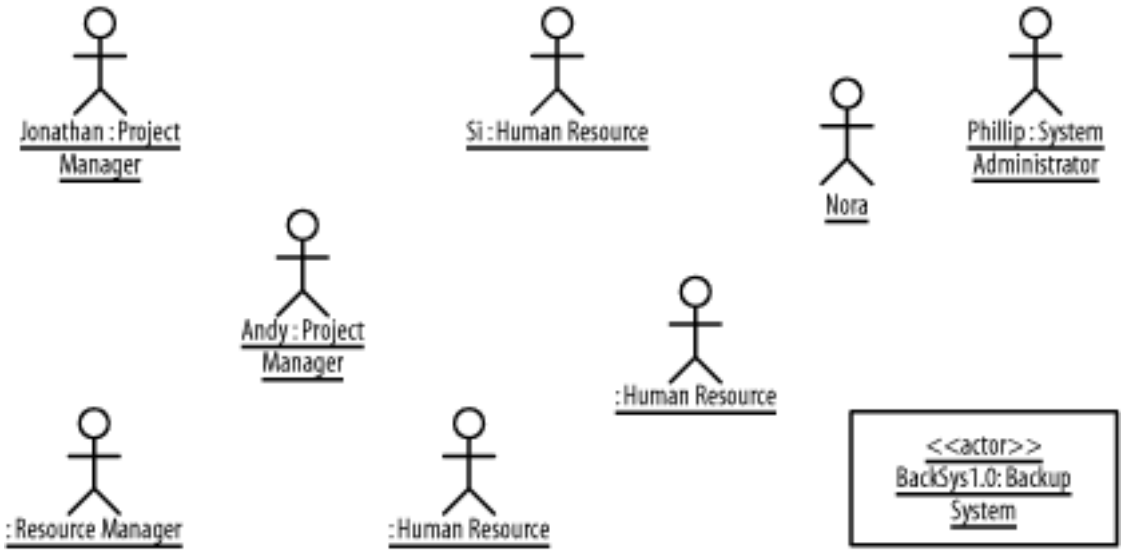

Figure 4-2. Actor instances

Because actors are external to a system and interact with that system, they define the boundary or scope of the system. For example, given the actors shown in Figure 4-1, we know exactly who and what will interact with the project management system. If we don't define our actors, we may fall into the trap of endlessly debating whether we have identified all the users of the system and all the other systems that interact with the system. Consequentially, because every functional requirement should be of interest to at least one user (otherwise, why would we build the system to provide the functionality?), without identifying actors, we have no way of knowing whether we have identified all the functional requirements of the system. An actor may also represent a resource owned by another project or purchased rather than built. For example, the backup system must be provided by another project, and it may be purchased from a vendor or built rather than purchased. Independent of how it is developed, we are interested in interacting with it as a resource.

Use Cases

As discussed in Chapter 2, a use case is a functional requirement that is described from the perspective of the users of a system. For example, functional requirements for the project management system include: security functionality (such as allowing users to log in and out of the system), inputting of data, processing of data, generation of reports, and so forth.

Use cases follow the type-instance dichotomy first discussed in Chapter 2 and applied to classes and object in Chapter 3. You can use the UML to talk about classes of use cases, and you can use the UML to talk about specific use cases of a class. When speaking of a class of use cases, it's customary to use the term use case or *use-case class*. Thus, while you might think of a use case as a specific thing, in the UML, a use case really represents a class of things. When speaking of a specific use case of a class, use the term *use-case instance*.

A use case defines a functional requirement that is described as a sequence of steps, which include actions performed by a system and interactions between the system and actors. Use cases address the question of how actors interact with a system, and describe the actions the system performs.

In the UML, a use case is shown as an ellipse and labeled with the name of the use-case class. Use cases may be enclosed by a rectangle that represents the boundary of the system that provides the functionality. Figure 4-3 shows various use cases associated with the project management system, including functionality to manage projects, manage resources, and administer the system.

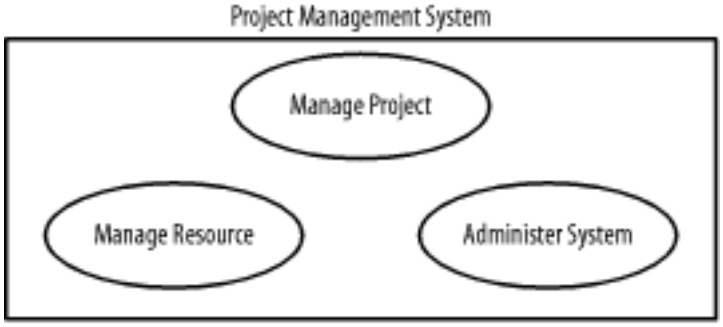

Figure 4-3. Use cases

A use-case instance, often called a *scenario*, is the performance of a specific sequence of actions and interactions. For example, a specific actor instance that uses security functionality to log in or out of the project management system represents a scenario. Other possible scenarios might include a user who enters data, a user who requests processing to be done on the data, or a user who requests that the system generate a report.

A use-case instance is shown similarly to a use case but labeled with the use-case instance name followed by a colon followed by the use-case name, all fully underlined. Both names are

optional, and the colon is present only if the use-case class name is specified. Use-case instances address the questions of how actor instances specifically interact with a system and what specific actions the system performs in return. Use-case instances are not commonly shown on use-case diagrams but rather are simply discussed with users to better understand their requirements.

Because use cases are performed by a system, they define the functional requirements of the system. For example, given the use cases shown in Figure 4-3, we know generally what type of functionality must be provided by the project management system. Other use-case diagrams may refine and decompose these use cases. If we don't define our use cases, we may fall into the trap of endlessly debating whether we have identified all the functionality of the system.

Each use case is composed of one or more behavior sequences. A *behavior sequence* is a sequence of steps in which each step specifies an action or interaction. Each *action* specifies processing performed by the system. Each *interaction* specifies some communication between the system and one of the actors who participate in the use case. For example, a login use case may have the following behavior sequence:

1. The system provides a user interface to capture login information.
2. The user enters her username.
3. The user enters her password.
4. The system validates the username and password.
5. The system responds appropriately by allowing the user to continue, or by rejecting the user, depending on whether her username and password are valid.

Steps 1, 4, and 5 are actions that the system performs, and steps 2 and 3 are interactions between the system and user. It is fairly simple to see that step 4 is an action, but you may be wondering why steps 1 and 5 are actions rather than interactions, when the system is interacting with the user by presenting a user interface to the user or by responding appropriately with a message window or something similar. Steps 1 and 5 are actions rather than interactions because the system is simply taking action to show a user interface or a response message window that the user can easily choose to ignore. However, if this were not a human user but another system that would receive some communication to which it must reply, these would be interactions rather than actions. Quite often, trying to classify each step as either an action or interaction is not necessary; rather, it is more important to consider a use case as a dialog between actors and the system, to understand how actors interact with the system, and to understand what actions are the responsibility of the system.

Actions and interactions may also be repeated, conditional, or optional. For example, the `Manage Project` use case may have a behavior sequence for finding a project on which to work, and the following three succeeding behavior sequences:

- One for managing projects involving employees only
- One for managing projects involving consultants only
- One for managing projects involving both employees and consultants

An instance of the `Manage Project` use case involves actor instances and finding a project and managing it using one of the three available behavior sequences based upon the type of project selected. Behavior sequences, such as the one shown earlier for the login use case, are commonly captured in textual form, but may also be captured using behavioral modeling techniques, as discussed in Part III. However, the UML does not provide any explicit guidance on how to capture behavior sequences.

Because use cases result in some observable value to one or more actors, they must allow actors to achieve their goals. After all, each actor has goals in interacting with a system. Use cases don't represent actor goals, but instead represent functionality that enable actors to achieve their goals. For example, the use cases shown in Figure 4-3 enable the actors shown in Figure 4-1 to achieve their goals in the following ways:

- To ensure that a project delivers a quality product within the specified time, cost, and resource constraints, a project manager may use the `Manage Project` use case.
- To ensure that trained and skilled human resources are available for projects, a resource manager may use the `Manage Resource` use case.
- To ensure that a project management system is available for a project, a system administrator may use the `Administer System` use case, which may involve a backup system.

The project management system does not offer any functionality to enable human resources to achieve their goals. Note that such functionality is outside the scope of the system.

Communicate Associations

Figure 4-1 shows actors associated with the project management system and Figure 4-3 shows use cases associated with the project management system, but how are actors and use cases related? A specialized type of association, called a *communicate association*, addresses the question of how actors and use cases are related and which actors participate in or initiate use cases. (Associations are discussed in Chapter 3.)

As discussed in Chapter 2, a communicate association between an actor and a use case indicates that the actor uses the use case; that is, it indicates that the actor communicates with the system and participates in the use case. A use case may have associations with multiple actors, and an actor may have associations with multiple use cases. A communicate association is shown as a solid-line between an actor and a use case.

Figure 4-4 shows that a project manager participates in managing projects, a resource manager participates in managing resources, and a system administrator and backup system participates in administering the system.

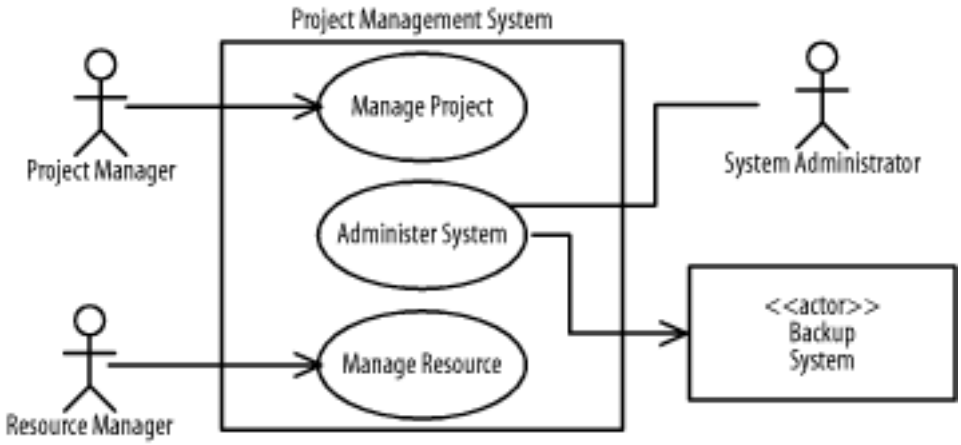

Figure 4-4. Actors and use cases

A navigation arrow on an association pointing toward a use case indicates that the actor initiates the interaction with the system. Figure 4-4 shows that a project manager initiates the interaction with the project management system to manage projects, and a resource manager initiates the interaction with the project management system to manage resources.

A navigation arrow on an association pointing toward an actor indicates that the system initiates the interaction with the actor. Figure 4-4 shows that the project management system initiates the interaction with the backup system to back up project management data.

Rather than use two arrows when either the system or the actor may initiate an interaction, navigation arrows on both ends of such an association are dropped. Figure 4-4 shows that either a system administrator or the system may initiate an interaction to administer the system. The system administrator might initiate an interaction with the system to back up the data, or, for example, the system might initiate an interaction with the system administrator informing the actor that system resources are low.

Be aware, however, that a lack of navigation arrows may simply result from a modeler choosing not to specify anything about the initiation of an interaction. Thus, with respect to Figure 4-4, you can't be absolutely certain that either actor can initiate a system administration interaction. It could be that the system administrator only can initiate the interaction, and the UML modeler simply chose not to specify initiation in this one case. It simply depends on the modeling guidelines the modeler is using.

Dependencies

A model may have many use cases, so how do we organize the use cases that define what a system should do? And how do we use this information about use cases to determine how best to execute a project while considering how use cases are related to one another, including what some use cases might have in common, and also taking into account use cases that are options of other use cases? Specialized types of dependencies, called include and extend dependencies,

address these questions; dependencies are discussed in Chapter 3. The next few sections discuss these specialized types of dependencies.

Include Dependencies

Perhaps we wish to log the activities of project managers, resources managers, and system administrators as they interact with the project management system. Figure 4-5 elaborates on the use cases in Figure 4-4 to show that the activities of the project manager, resource managers, and system administrators are logged when they are performing the use cases shown in the diagram. Thus, logging activities are common to these three use cases. We can use an include dependency to address this type of situation by factoring out and reusing common behavior from multiple use cases.

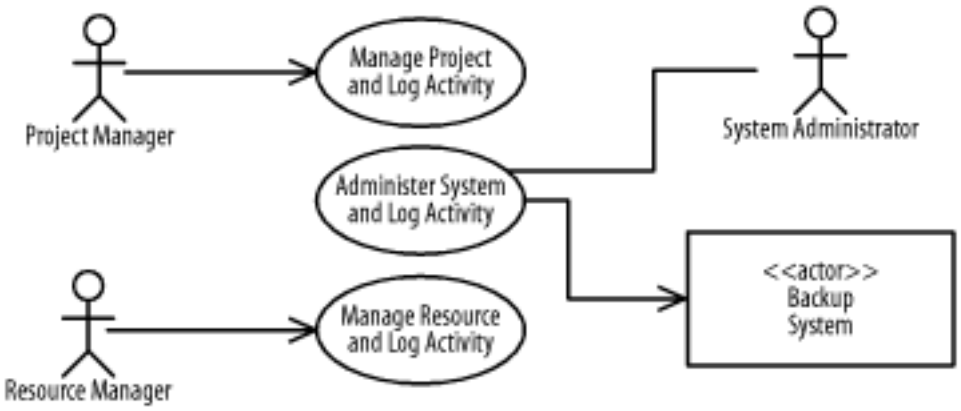

Figure 4-5. Use cases with common behavior

An *include dependency* from one use case (called the *base use case*) to another use case (called the *inclusion use case*) indicates that the base use case will include or call the inclusion use case. A use case may include multiple use cases, and it may be included in multiple use cases. An include dependency is shown as a dashed arrow from the base use case to the inclusion use case marked with the `include` keyword. The base use case is responsible for identifying where in its behavior sequence or at which step to include the inclusion use case. This identification is not done in the UML diagram, but rather in the textual description of the base use case.

Figure 4-6 refines Figure 4-5 using include dependencies. The `Log Activity` use case is common to the `Manage Project`, `Manage Resource`, and `Administer System` use cases, so it is factored out and included by these use cases.

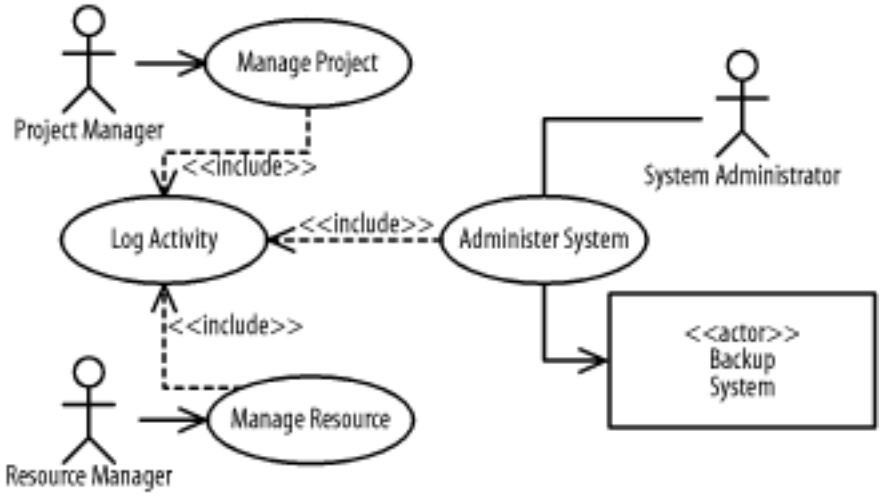

Figure 4-6. Include dependencies

You can use an include dependency when a use case may be common to multiple other use cases and is therefore factored out of the different use cases so that it may be reused. The `Log Activity` use case in Figure 4-6 is included in the `Manage Project`, `Manage Resource`, and `Administer System` use cases. Consequently, you must analyze and develop that use case before you develop the three use cases that depend on it.

Extend Dependencies

Projects are made of activities, and activities are made of tasks. Figure 4-7 elaborates the `Manage Project` use case in Figure 4-4, and shows that a project manager may manage projects by maintaining the project itself, its activities, or its tasks. Thus, maintaining the project, its activities, and its tasks are options of managing a project. You can use an extend dependency to address this situation by factoring out optional behavior from a use case.

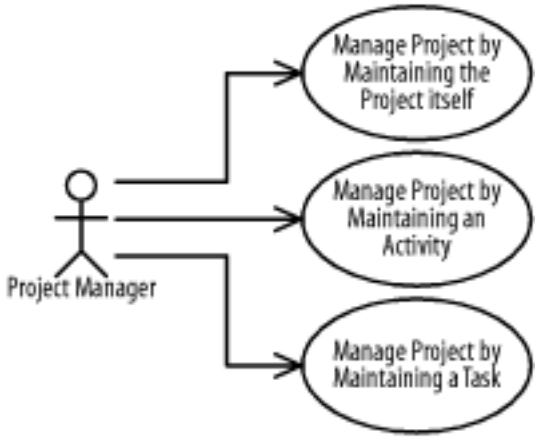

Figure 4-7. Use cases with optional behavior

An *extend dependency* from one use case (called the *extension use case*) to another use case (called the *base use case*) indicates that the extension use case will extend (or be inserted into) and augment the base use case. A use case may extend multiple use cases, and a use case may be extended by multiple use cases. An extend dependency is shown as a dashed arrow from the extension use case to the base use case marked with the `extend` keyword. The base use case is responsible for identifying at which steps in its behavior sequence the extending use cases may be inserted.

Figure 4-8 refines Figure 4-7 using extend dependencies.
The `Maintain Project`, `Maintain Activity`, and `Maintain Task` use cases are options of the `Manage Project` use case, so `Manage Project` is factored out and extends those three use cases.

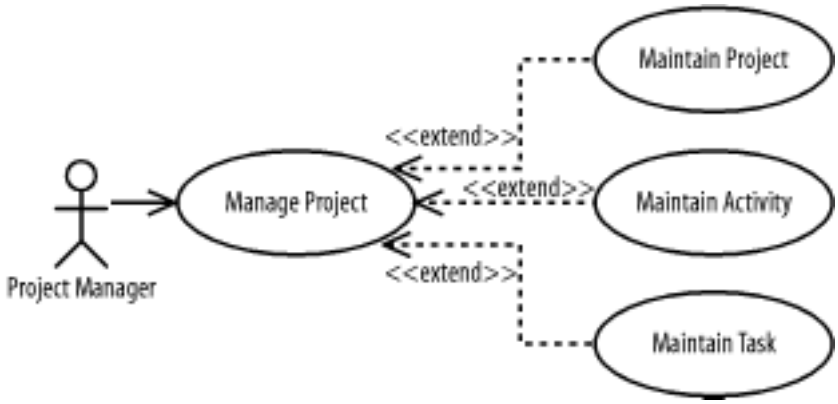

Figure 4-8. Simple extend dependencies

The location in a base use case at which another behavior sequence may be inserted is called an *extension point*. Extension points for a use case may be listed in a compartment labeled "Extension Points" where each extension point is shown inside the compartment with an extension-point name followed by a colon followed by a suitable description of the location of the extension point in the use case's behavior sequence. Locations may be described as being before, after, or in-the-place-of a step in the base use case's behavior sequence. For example, the `Manage Project` use case may have a behavior sequence for finding a project on which to work followed by an extension point named `Project Selected` followed by another behavior. The `Project Selected` extension point may be described as occurring after a project is found but before it is actually worked on.

An extend dependency is responsible for defining when an extension use case is inserted into the base use case by specifying a condition that must be satisfied for the insertion to occur. The condition may be shown following the `extend` keyword enclosed within square brackets followed by the extension point name enclosed in parentheses. For example, other use cases may be inserted into the `Project Selected` extension point just described for the `Manage Project` use case. Such behavior sequences may include reviewing and updating project information, or selecting a specific version of a project before managing the details of the project in the succeeding behavior sequences.

Figure 4-9 elaborates on the `Administer System` use case in Figure 4-4 using extend dependencies. It shows that a system administrator is offered two options — starting up the system or shutting down the system — at the extension point named `Administration Functions`, which is described as being available on the administration menu of the user interface. Figure 4-9 further shows the following:

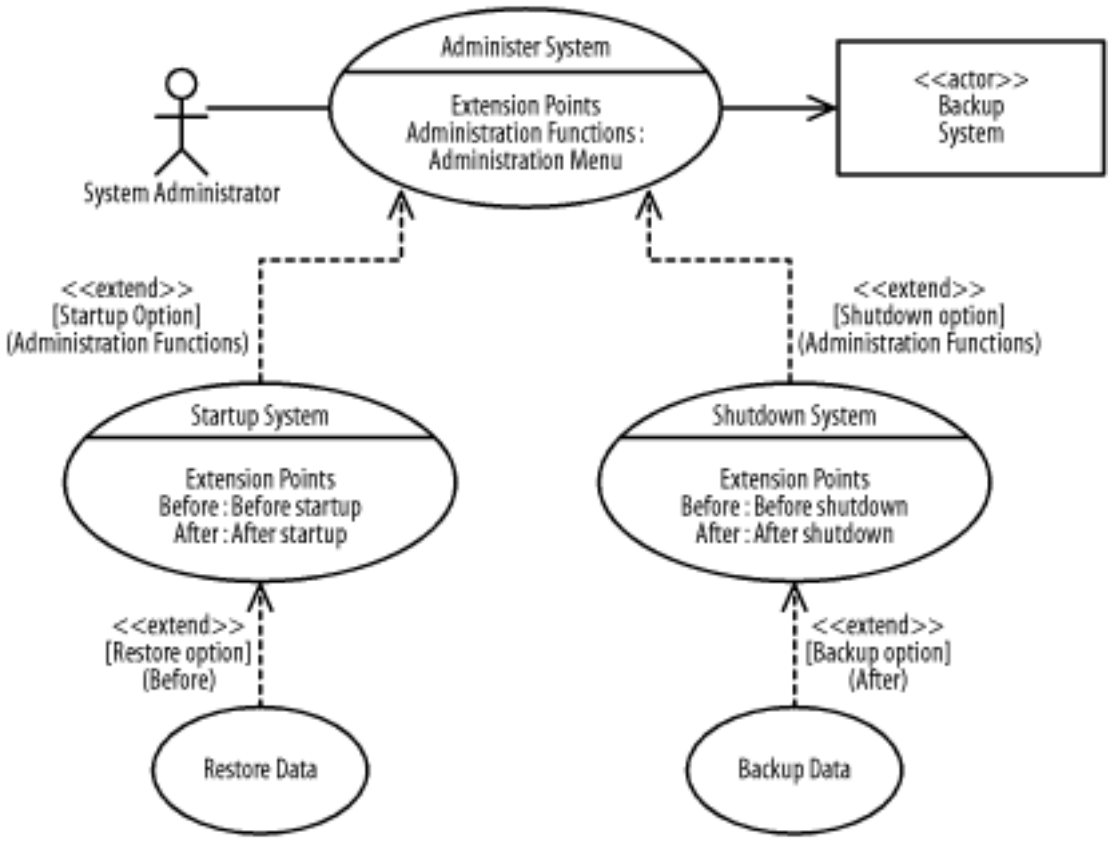

Figure 4-9. Extension points and extend dependencies

- The `Startup System` use case is available as an option at the `Administration Functions` extension point of the `Administer System` use case. The `Startup System` use case has two extension points named `Before` and `After`. The `Before` extension point is described as occurring before the startup functionality is performed by the system, and the `After` extension point is described as occurring after the startup functionality is performed by the system. These extension points are used as follows:
    - The `Restore Data` use case is available as an option at the `Before` extension point of the `Startup System` use case. Before starting up the system, the system administrator may restore data from the backup system to the project management system's database that was previously archived.
    - There are no options described for the `After` extension point of the `Startup System` use case.
- The `Shutdown System` use case is available as an option at the `Administration Functions` extension point of the `Administer System` use case.

The Shutdown System use case has two extension points, named Before and After. The Before extension point is described as occurring before the shutdown functionality is performed by the system, and the After extension point is described as occurring after the shutdown functionality is performed by the system. These extension points are used as follows:

- The Backup Data use case is available as an option at the After extension point of the Shutdown System use case. After shutting down the system, the system administrator may back up data from the project management system's database to the backup system for later retrieval.
- There are no options described for the Before extension point of the Shutdown System use case.

The extension points just described allow us to insert behavior into the Startup System and Shutdown System use cases before or after they perform startup or shutdown processing for the project management system. The extend dependencies reference these extension points to indicate where use cases may be inserted inside one another, and also to indicate the conditions that must be satisfied for such an insertion to occur. Naturally, data is restored before the system is started up and data is backed up after the system is shut down.

Use an extend dependency when a use case is optional to another use case. Because the Maintain Project, Maintain Activity, and Maintain Task use cases extend the Manage Project use case, the Manage Project use case must be developed before the others; otherwise, the other use cases won't have a use case to extend. Likewise, the Administer System use case must be developed before the Startup System and Shutdown System use cases, Startup System must be developed before Restore Data, and Shutdown System must be developed before Backup Data. However, once Administer System is developed, Startup System and Shutdown System may be developed in parallel or concurrently, because they are not directly related.