- **State machine diagrams** represent the behavior of nontrivial objects (Section 2.4.4).
- **Activity diagrams** are flow diagrams used to represent the data flow or the control flow through a system (Section 2.4.5).

2.4.1 Use Case Diagrams

Use cases and actors

Actors are external entities that interact with the system. Examples of actors include a user role (e.g., a system administrator, a bank customer, a bank teller) or another system (e.g., a central database, a fabrication line). Actors have unique names and descriptions.

Use cases describe the behavior of the system as seen from an actor's point of view. Behavior described by use cases is also called **external behavior**. A use case describes a function provided by the system as a set of events that yields a visible result for the actors. Actors initiate a use case to access system functionality. The use case can then initiate other use cases and gather more information from the actors. When actors and use cases exchange information, they are said to **communicate**. We will see later that we represent these exchanges with communication relationships.

For example, in an accident management system, field officers (such as a police officer or a fire fighter) have access to a wireless computer that enables them to interact with a dispatcher. The dispatcher in turn can visualize the current status of all its resources, such as police cars or trucks, on a computer screen and dispatch a resource by issuing commands from a workstation. In this example, field officer and dispatcher can be modeled as actors.

Figure 2-13 depicts the actor `FieldOfficer` who invokes the use case `ReportEmergency` to notify the actor `Dispatcher` of a new emergency. As a response, the `Dispatcher` invokes the

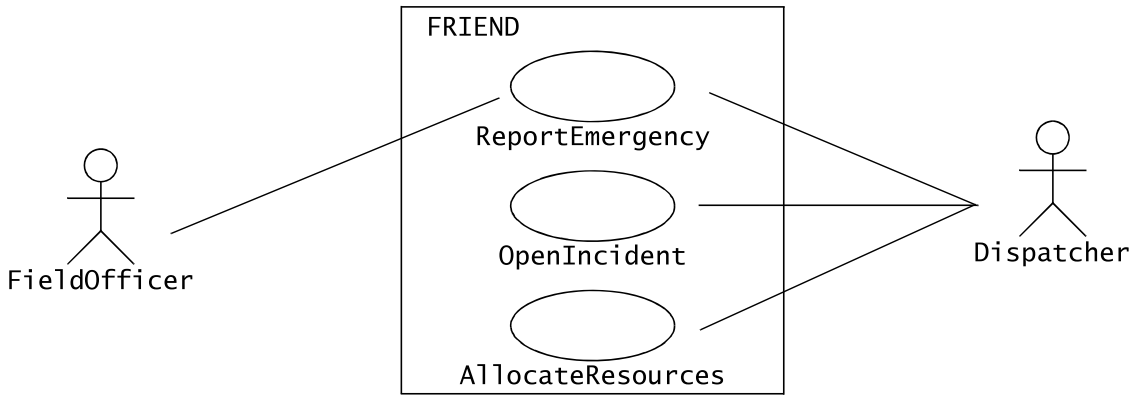

Figure 2-13 An example of a UML use case diagram for First Responder Interactive Emergency Navigational Database (`FRIEND`), an accident management system. Associations between actors and use cases denote information flows. These associations are bidirectional: they can represent the actor initiating a use case (`FieldOfficer` initiates `ReportEmergency`) or a use case providing information to an actor (`ReportEmergency` notifies `Dispatcher`). The box around the use cases represents the system boundary.

OpenIncident use case to create an incident report and initiate the incident handling. The Dispatcher enters preliminary information from the FieldOfficer in the incident database (FRIEND) and orders additional units to the scene with the AllocateResources use case.

For the textual description of a use case, we use a template composed of six fields (see Figure 2-14) adapted from [Constantine & Lockwood, 2001]:

- The **name** of the use case is unique across the system so that developers (and project participants) can unambiguously refer to the use case.
- **Participating actors** are actors interacting with the use case.
- **Entry conditions** describe the conditions that need to be satisfied before the use case is initiated.

Use case name	ReportEmergency
Participating actors	Initiated by FieldOfficer Communicates with Dispatcher
Flow of events	1. The FieldOfficer activates the "Report Emergency" function of her terminal. 2. FRIEND responds by presenting a form to the FieldOfficer. 3. The FieldOfficer fills out the form by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form. 4. FRIEND receives the form and notifies the Dispatcher. 5. The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the report. 6. FRIEND displays the acknowledgment and the selected response to the FieldOfficer.
Entry condition	• The FieldOfficer is logged into FRIEND.
Exit condition	• The FieldOfficer has received an acknowledgment and the selected response from the Dispatcher, OR • The FieldOfficer has received an explanation indicating why the transaction could not be processed.
Quality requirements	• The FieldOfficer's report is acknowledged within 30 seconds. • The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.

Figure 2-14 An example of a use case, ReportEmergency.

- The **flow of events** describes the sequence of interactions of the use case, which are to be numbered for reference. The common case (i.e., cases that are expected by the user) and the exceptional cases (i.e., cases unexpected by the user, such as errors and unusual conditions) are described separately in different use cases for clarity. We organize the steps in the flow of events in two columns, the left column representing steps accomplished by the actor, the right column representing steps accomplished by the system. Each pair of actor–system steps represents an interaction.
- **Exit conditions** describe the conditions satisfied after the completion of the use case.
- **Quality requirements** are requirements that are not related to the functionality of the system. These include constraints on the performance of the system, its implementation, the hardware platforms it runs on, and so on. Quality requirements are described in detail in Chapter 4, *Requirements Elicitation*.

Use cases are written in natural language. This enables developers to use them for communicating with the client and the users, who generally do not have an extensive knowledge of software engineering notations. The use of natural language also enables participants from other disciplines to understand the requirements of the system. The use of the natural language allows developers to capture things, in particular special requirements, that cannot easily be captured in diagrams.

Use case diagrams can include four types of relationships: communication, inclusion, extension, and inheritance. We describe these relationships in detail next.

Communication relationships

Actors and use cases communicate when information is exchanged between them. **Communication relationships** are depicted by a solid line between the actor and use case symbol. In Figure 2-13, the actors `FieldOfficer` and `Dispatcher` communicate with the `ReportEmergency` use case. Only the actor `Dispatcher` communicates with the use cases `OpenIncident` and `AllocateResources`. Communication relationships between actors and use cases can be used to denote access to functionality. In the case of our example, a `FieldOfficer` and a `Dispatcher` are provided with different interfaces to the system and have access to different functionality.

Include relationships

When describing a complex system, its use case model can become quite complex and can contain redundancy. We reduce the complexity of the model by identifying commonalities in different use cases. For example, assume that the `Dispatcher` can press at any time a key to access a street map. This can be modeled by a use case `ViewMap` that is included by the use cases `OpenIncident` and `AllocateResources` (and any other use cases accessible by the `Dispatcher`). The resulting model only describes the `ViewMap` functionality once, thus reducing complexity of

the overall use case model. Two use cases are related by an include relationship if one of them includes the second one in its flow of events. In use case diagrams, **include relationships** are depicted by a dashed open arrow originating from the including use case (see Figure 2-15). Include relationships are labeled with the string «include».

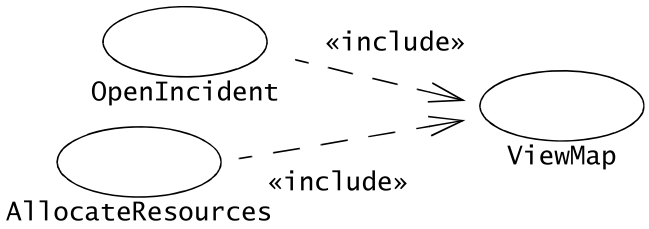

Figure 2-15 An example of an «include» relationship (UML use case diagram).

Use case name	AllocateResources
Participating actor	Initiated by Dispatcher
Flow of events	. . .
Entry condition	• The Dispatcher opens an Incident.
Exit condition	• Additional Resources are assigned to the Incident. • Resources receives notice about their new assignment. • FieldOfficer in charge of the Incident receives notice about the new Resources.
Quality requirements	At any point during the flow of events, this use case can **include** the ViewMap use case. The ViewMap use case is initiated when the Dispatcher invokes the map function. When invoked within this use case, the system scrolls the map so that location of the current Incident is visible to the Dispatcher.

Figure 2-16 Textual representation of include relationships of Figure 2-15. "Include" in **bold** for clarity.

We represent include relationships in the textual description of the use case with one of two ways. If the included use case can be included at any point in the flow of events (e.g., the ViewMap use case), we indicate the inclusion in the *Quality requirements* section of the use case (Figure 2-16). If the included use case is invoked during an event, we indicate the inclusion in the flow of events.

Extend relationships

Extend relationships are an alternate means for reducing complexity in the use case model. A use case can extend another use case by adding events. An extend relationship indicates that an instance of an extended use case may include (under certain conditions) the

behavior specified by the extending use case. A typical application of extend relationships is the specification of exceptional behavior. For example (Figure 2-17), assume that the network connection between the `Dispatcher` and the `FieldOfficer` can be interrupted at any time. (e.g., if the `FieldOfficer` enters a tunnel). The use case `ConnectionDown` describes the set of events taken by the system and the actors while the connection is lost. `ConnectionDown` extends the use cases `OpenIncident` and `AllocateResources`. Separating exceptional behavior from common behavior enables us to write shorter and more focused use cases. In the textual representation of a use case, we represent extend relationships as entry conditions of the extending use case. For example, the extend relationships depicted in Figure 2-17 are represented as an entry condition of the `ConnectionDown` use case (Figure 2-18).

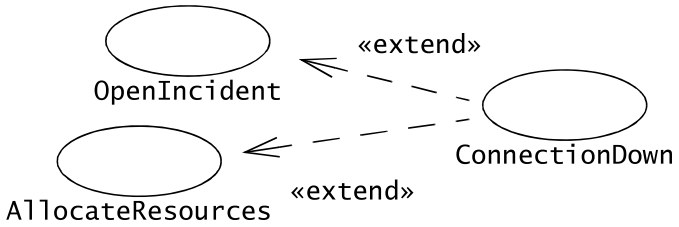

Figure 2-17 An example of an «`extend`» relationship (UML use case diagram).

Use case name	`ConnectionDown`
Participating actor	Communicates with `FieldOfficer` and `Dispatcher`.
Flow of events	. . .
Entry condition	This use case **extends** the `OpenIncident` and the `AllocateResources` use cases. It is initiated by the system whenever the network connection between the `FieldOfficer` and `Dispatcher` is lost.
Exit condition	. . .

Figure 2-18 Textual representation of extend relationships of Figure 2-17. "Extends" in **bold** for clarity.

The difference between the include and extend relationships is the location of the dependency. Assume that we add several new use cases for the actor `Dispatcher`, such as `UpdateIncident` and `ReallocateResources`. If we modeled the `ConnectionDown` use case with include relationships, the authors of `UpdateIncident` and `ReallocateResources` use cases need to know about and include the `ConnectionDown` use case. If we used extend relationships instead, only the `ConnectionDown` use case needs to be modified to extend the additional use cases. In general exception cases (such as help, errors, and other unexpected conditions) are modeled with extend relationships. Use cases that describe behavior commonly shared by a limited set of use cases are modeled with include relationships.

Inheritance relationships

An **inheritance relationship** is a third mechanism for reducing the complexity of a model. One use case can specialize another more general one by adding more detail. For example, `FieldOfficers` are required to authenticate before they can use `FRIEND`. During early stages of requirements elicitation, authentication is modeled as a high-level `Authenticate` use case. Later, developers describe the `Authenticate` use case in more detail and allow for several different hardware platforms. This refinement activity results in two more use cases: `AuthenticateWithPassword` which enables `FieldOfficers` to login without any specific hardware, and `AuthenticateWithCard` which enables `FieldOfficers` to login using a smart card. The two new use cases are represented as specializations of the `Authenticate` use case (Figure 2-19). In the textual representation, specialized use cases inherit the initiating actor and the entry and exit conditions from the general use case (Figure 2-20).

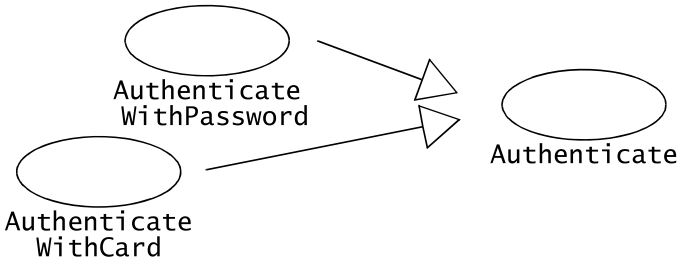

Figure 2-19 An example of an inheritance relationship (UML use case diagram). The `Authenticate` use case is a high-level use case describing, in general terms, the process of authentication. `AuthenticateWithPassword` and `AuthenticateWithCard` are two specializations of `Authenticate`.

Use case name	`AuthenticateWithCard`
Participating actor	**Inherited** from Authenticate use case.
Flow of events	1. The `FieldOfficer` inserts her card into the field terminal. 2. The field terminal acknowledges the card and prompts the actor for her personal identification number (PIN). 3. The `FieldOfficer` enters her PIN with the numeric keypad. 4. The field terminal checks the entered PIN against the PIN stored on the card. If the PINs match, the `FieldOfficer` is authenticated. Otherwise, the field terminal rejects the authentication attempt.
Entry condition	**Inherited** from `Authenticate` use case.
Exit condition	**Inherited** from `Authenticate` use case.

Figure 2-20 Textual representation of inheritance relationships of Figure 2-19.

Note that extend relationships and inheritance relationships are different. In an extend relationship, each use case describes a different flow of events to accomplish a different task. In Figure 2-17, the `OpenIncident` use cases describes the actions that occur when the `Dispatcher` creates a new `Incident`, whereas the `ConnectionDown` use case describes the actions that occur during network outages. In Figure 2-19, `AuthenticateWithPassword` and `Authenticate` both describe the same task, each at different abstraction levels.

Scenarios

A use case is an abstraction that describes all possible scenarios involving the described functionality. A **scenario** is an instance of a use case describing a concrete set of actions. Scenarios are used as examples for illustrating common cases; their focus is on understandability. Use cases are used to describe all possible cases; their focus is on completeness. We describe a scenario using a template with three fields:

- The **name** of the scenario enables us to refer to it unambiguously. The name of a scenario is underlined to indicate that it is an instance.
- The **participating actor instances** field indicates which actor instances are involved in this scenario. Actor instances also have underlined names.
- The **flow of events** of a scenario describes the sequence of events step by step.

Note that there is no need for entry or exit conditions in scenarios. Entry and exit conditions are abstractions that enable developers to describe a range of conditions under which a use case is invoked. Given that a scenario only describes one specific situation, such conditions are unnecessary (Figure 2-21).

2.4.2 Class Diagrams

Classes and objects

Class diagrams describe the structure of the system in terms of classes and objects. **Classes** are abstractions that specify the attributes and behavior of a set of objects. A class is a collection of objects that share a set of attributes that distinguish the objects as members of the collection. **Objects** are entities that encapsulate state and behavior. Each object has an identity: it can be referred individually and is distinguishable from other objects.

In UML, classes and objects are depicted by boxes composed of three compartments. The top compartment displays the name of the class or object. The center compartment displays its attributes, and the bottom compartment displays its operations. The attribute and operation compartments can be omitted for clarity. Object names are underlined to indicate that they are instances. By convention, class names start with an uppercase letter. Objects in object diagrams may be given names (followed by their class) for ease of reference. In that case, their name starts