2

Modeling with UML

Every mechanic is familiar with the problem of the part you can't buy because you can't find it because the manufacturer considers it a part of something else.

—Robert Pirsig, in *Zen and the Art of Motorcycle Maintenance*

Notations enable us to articulate complex ideas succinctly and precisely. In projects involving many participants, often of different technical and cultural backgrounds, accuracy and clarity are critical as the cost of miscommunication increases rapidly.

For a notation to enable accurate communication, it must come with a *well-defined* semantics, it must be *well suited* for representing a given aspect of a system, and it must be *well understood* among project participants. In the latter lies the strength of standards and conventions: when a notation is used by a large number of participants, there is little room for misinterpretation and ambiguity. Conversely, when many dialects of a notation exists, or when a very specialized notation is used, the notation users are prone to misunderstandings as each user imposes its own interpretation. We selected UML (Unified Modeling Language, [OMG, 2009]) as a primary notation for this book because it provides a spectrum of notations for representing different aspects of a system and has been accepted as a standard notation in the industry.

In this chapter, we first describe the concepts of modeling in general and object-oriented modeling in particular. We then describe five fundamental notations of UML that we use throughout the book: use case diagrams, class diagrams, interaction diagrams, state machine diagrams, and activity diagrams. For each of these notations, we describe its basic semantics and provide examples. We revisit these notations in detail in later chapters as we describe the activities that use them. Specialized notations that we use in only one chapter are introduced later, such as UML

and deployment diagrams in Chapter 6, *System Design: Decomposing the System*, and PERT charts in Chapter 14, *Project Management*.

2.1 Introduction

UML is a notation that resulted from the unification of OMT (Object Modeling Technique [Rumbaugh et al., 1991]), Booch [Booch, 1994], and OOSE (Object-Oriented Software Engineering [Jacobson et al., 1992]). UML has also been influenced by other object-oriented notations, such as those introduced by Shlaer and Mellor [Mellor & Shlaer, 1998], Coad and Yourdon [Coad et al., 1995], Wirfs-Brock [Wirfs-Brock et al., 1990], and Martin and Odell [Martin & Odell, 1992].

The goal of UML is to provide a standard notation that can be used by all object-oriented methods and to select and integrate the best elements of precursor notations. For example, UML includes the use case diagrams introduced by OOSE and uses many features of the OMT class diagrams. UML also includes new concepts that were not present in other major methods at the time, such as extension mechanisms and a constraint language. UML has been designed for a broad range of applications. Hence, it provides constructs for a broad range of systems and activities (e.g., distributed systems, analysis, system design, deployment). System development focuses on three different models of the system (see Figure 1-2):

- The **functional model**, represented in UML with use case diagrams, describes the functionality of the system from the user's point of view.
- The **object model**, represented in UML with class diagrams, describes the structure of the system in terms of objects, attributes, associations, and operations. During requirements and analysis, the object model starts as the *analysis object model* and describes the application concepts relevant to the system. During system design, the object model is refined into the *system design object model* and includes descriptions of the subsystem interfaces. During object design, the object model is refined into the *object design model* and includes detailed descriptions of solution objects.
- The **dynamic model**, represented in UML with interaction diagrams, state machine diagrams, and activity diagrams, describes the internal behavior of the system. Interaction diagrams describe behavior as a sequence of messages exchanged among a *set of objects*, whereas state machine diagrams describe behavior in terms of states of an *individual object* and the possible transitions between states. Activity diagrams describe behavior in terms control and data flows.

In this chapter, we describe UML diagrams for representing these models. Introducing these notations represents an interesting challenge: understanding the purpose of a notation requires some familiarity with the activities that use it. However, it is necessary to understand the notation before describing the activities. To address this issue, we introduce UML iteratively. In the next section, we first provide an overview of the five basic notations of UML. In Section 2.3, we introduce the fundamental ideas of modeling. In Section 2.4, we revisit the five basic notations of UML in light of modeling concepts. In subsequent chapters, we discuss these notations in even greater detail when we introduce the activities that use them.

2.2 An Overview of UML

In this section, we briefly introduce five UML notations:

- Use Case Diagrams (Section 2.2.1)
- Class Diagrams (Section 2.2.2)
- Interaction Diagrams (Section 2.2.3)
- State Machine Diagrams (Section 2.2.4)
- Activity Diagrams (Section 2.2.5).

2.2.1 Use Case Diagrams

Use cases are used during requirements elicitation and analysis to represent the functionality of the system. Use cases focus on the behavior of the system from an external point of view. A use case describes a function provided by the system that yields a visible result for an actor. An actor describes any entity that interacts with the system (e.g., a user, another system, the system's physical environment). The identification of actors and use cases results in the definition of the boundary of the system, that is, in differentiating the tasks accomplished by the system and the tasks accomplished by its environment. The actors are outside the boundary of the system, whereas the use cases are inside the boundary of the system.

For example, Figure 2-1 depicts a use case diagram for a simple watch. The `WatchUser` actor may either consult the time on their watch (with the `ReadTime` use case) or set the time (with the `SetTime` use case). However, only the `WatchRepairPerson` actor can change the battery of the watch (with the `ChangeBattery` use case).

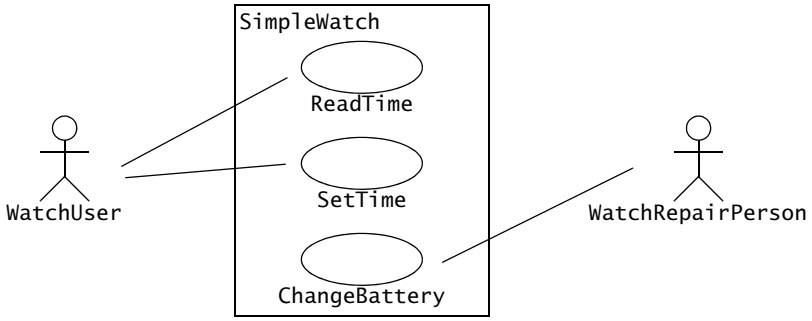

Figure 2-1 A UML use case diagram describing the functionality of a simple watch. The `WatchUser` actor may either consult the time on her watch (with the `ReadTime` use case) or set the time (with the `SetTime` use case). However, only the `WatchRepairPerson` actor can change the battery of the watch (with the `ChangeBattery` use case). Actors are represented with stick figures, use cases with ovals, and the boundary of the system with a box enclosing the use cases.

2.2.2 Class Diagrams

Class diagrams are used to describe the structure of the system. Classes are abstractions that specify the common structure and behavior of a set of objects. Objects are instances of classes that are created, modified, and destroyed during the execution of the system. An object has state that includes the values of its attributes and its links with other objects.

Class diagrams describe the system in terms of objects, classes, attributes, operations, and their associations. For example, Figure 2-2 is a class diagram describing the elements of all the watches of the `SimpleWatch` class. These watch objects all have an association to an object of the `PushButton` class, an object of the `Display` class, an object of the `Time` class, and an object of the `Battery` class. The numbers on the ends of associations denote the number of links each `SimpleWatch` object can have with an object of a given class. For example, a `SimpleWatch` has exactly two `PushButtons`, one `Display`, two `Batteries`, and one `Time`. Similarly, all `PushButton`, `Display`, `Time`, and `Battery` objects are associated with exactly one `SimpleWatch` object.

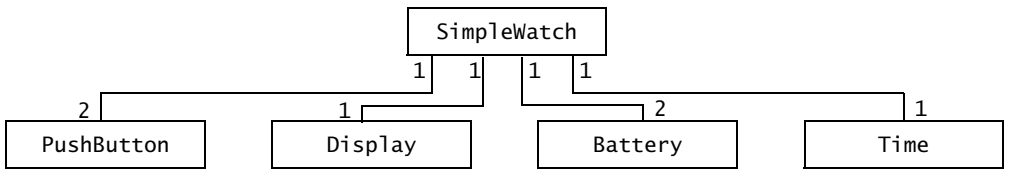

Figure 2-2 A UML class diagram describing the elements of a simple watch.

At the analysis level, associations represent existence relationships. For example, a `SimpleWatch` requires the correct number of `PushButtons`, `Displays`, `Batteries`, and `Time`. In this example, the association is symmetrical: `PushButton` cannot perform its function without a `SimpleWatch`. UML also allows for one-directional relationships, which we describe in Section 2.4.2. At the implementation level, associations are realized as references (i.e., pointers) to objects.

2.2.3 Interaction Diagrams

Interaction diagrams are used to formalize the dynamic behavior of the system and to visualize the communication among objects. They are useful for identifying additional objects that participate in the use cases. We call objects involved in a use case **participating objects**. An interaction diagram represents the interactions that take place among these objects. For example, Figure 2-3 is a special form of interaction diagram, called a **sequence diagram**, for the `SetTime` use case of our simple watch. The left-most column represents the `WatchUser` actor who initiates the use case. Labeled arrows represent stimuli that an actor or an object sends to other objects. In this case, the `WatchUser` presses button 1 twice and button 2 once to set her watch a minute

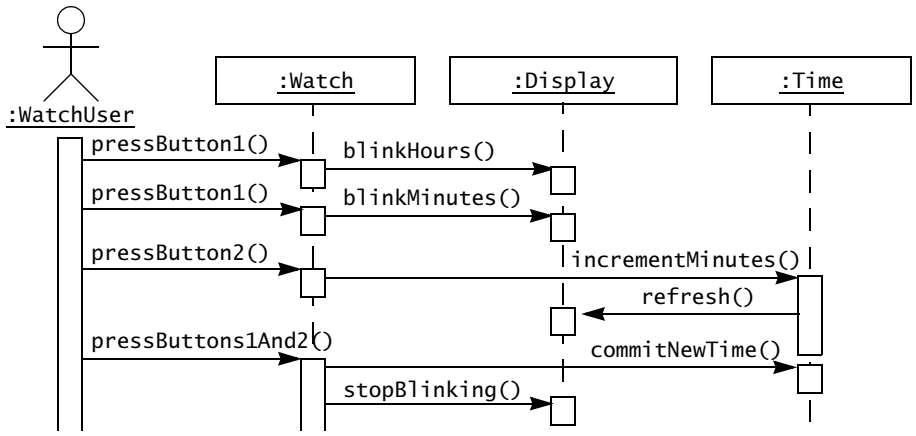

Figure 2-3 A UML sequence diagram for the `Watch`. The left-most column represents the timeline of the `WatchUser` actor who initiates the use case. The other columns represent the timeline of the objects that participate in this use case. Object names are underlined to denote that they are instances (as opposed to classes). Labeled arrows are stimuli that an actor or an object sends to other objects.

ahead. The `SetTime` use case terminates when the `WatchUser` presses both buttons simultaneously.

2.2.4 State Machine Diagrams

State machine diagrams describe the dynamic behavior of an individual object as a number of states and transitions between these states. A state represents a particular set of values for an object. Given a state, a transition represents a future state the object can move to and the conditions associated with the change of state. For example, Figure 2-4 is a state machine diagram for the `Watch`. Note that this diagram represents different information than the sequence diagram of Figure 2-3. The sequence diagram focuses on the messages exchanged between objects as a result of external events created by actors. The state machine diagram focuses on the transitions between states as a result of external events for an individual object.

2.2.5 Activity Diagrams

An activity diagram describes the behavior of a system in terms of activities. Activities are modeling elements that represent the execution of a set of operations. The execution of an activity can be triggered by the completion of other activities, by the availability of objects, or by external events. Activity diagrams are similar to flowchart diagrams in that they can be used to represent control flow (i.e., the order in which operations occur) and data flow (i.e., the objects that are exchanged among operations). For example, Figure 2-5 is an activity diagram representing activities related to managing an `Incident`. Rounded rectangles represent activities; arrows between activities represent control flow; thick bars represent the

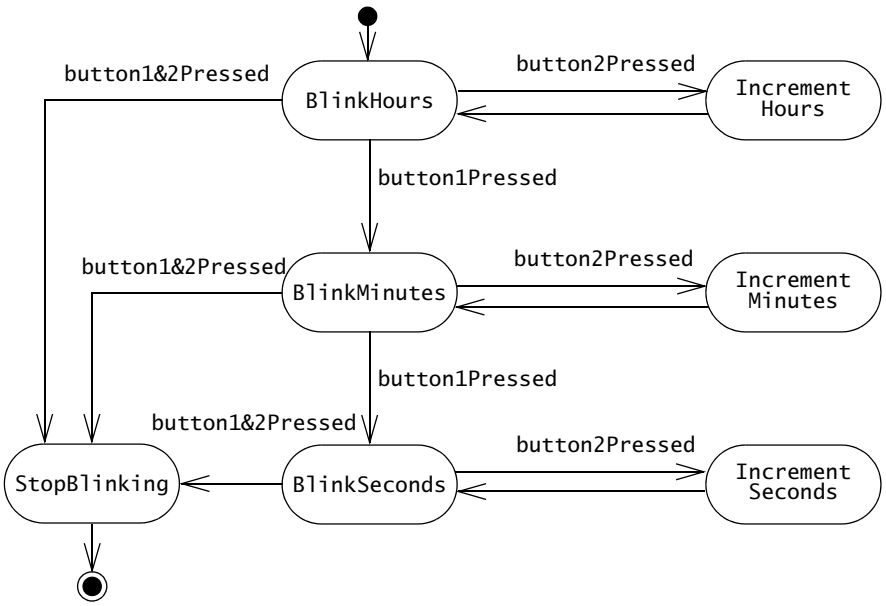

Figure 2-4 A UML state machine diagram for `SetTime` use case of the `Watch`.

synchronization of the control flow. The activity diagram of Figure 2-5 depicts that the `AllocateResources`, `CoordinateResources`, and `DocumentIncident` can be initiated only after the `OpenIncident` activity has been completed. Similarly, the `ArchiveIncident` activity can be initiated only after the completion of `AllocateResources`, `Coordinate-Resources`, and `DocumentIncident`. These latter three activities, however, can occur concurrently.

This concludes our first walkthrough of the five basic notations of UML. Now, we go into more detail: In Section 2.3, we introduce basic modeling concepts, including the definition of

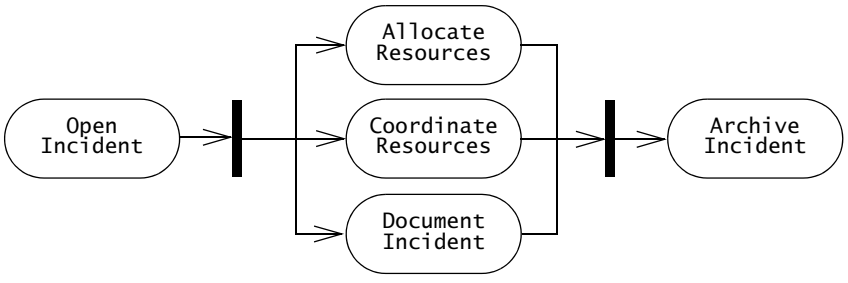

Figure 2-5 An example of a UML activity diagram. Activity diagrams represent behavior in terms of activities and their precedence constraints. The completion of an activity triggers an outgoing transition, which in turn may initiate another activity.

systems, models, types, and instances, abstraction, and falsification. In Sections 2.4.1–2.4.5, we describe in detail use case diagrams, class diagrams, sequence diagrams, state machine diagrams, and activity diagrams. We illustrate their use with a simple example. Section 2.4.6 describes miscellaneous constructs, such as packages and notes, that are used in all types of diagrams. We use these five notations throughout the book to describe software systems, work products, activities, and organizations. By the consistent and systematic use of a small set of notations, we hope to provide the reader with an operational knowledge of UML.

2.3 Modeling Concepts

In this section, we describe the basic concepts of modeling. We first define the terms **system** and **model** and discuss the purpose of **modeling**. We explain their relationship to programming languages and terms such as **data types**, **classes**, **instances**, and **objects**. Finally, we describe how object-oriented modeling focuses on building an abstraction of the system environment as a basis for the **system model**.

2.3.1 Systems, Models, and Views

A **system** is an organized set of communicating parts. We focus here on engineered systems, which are designed for a specific purpose, as opposed to natural systems, such as a planetary system, whose ultimate purpose we may not know. A car, composed of four wheels, a chassis, a body, and an engine, is designed to transport people. A watch, composed of a battery, a circuit, wheels, and hands, is designed to measure time. A payroll system, composed of a mainframe computer, printers, disks, software, and the payroll staff, is designed to issue salary checks for employees of a company. Parts of a system can in turn be considered as simpler systems called **subsystems**. The engine of a car, composed of cylinders, pistons, an injection module, and many other parts, is a subsystem of the car. Similarly, the integrated circuit of a watch and the mainframe computer of the payroll system are subsystems. This subsystem decomposition can be recursively applied to subsystems. Objects represent the end of this recursion, when each piece is simple enough that we can fully comprehend it without further decomposition.

Many systems are made of numerous subsystems interconnected in complicated ways, often so complex that no single developer can manage its entirety. **Modeling** is a means for dealing with this complexity. Complex systems are generally described by more than one model, each focusing on a different aspect or level of accuracy. Modeling means constructing an abstraction of a system that focuses on interesting aspects and ignores irrelevant details. What is interesting or irrelevant varies with the task at hand. For example, assume we want to build an airplane. Even with the help of field experts, we cannot build an airplane from scratch and hope that it will function correctly on its maiden flight. Instead, we first build a scale model of the air frame to test its aerodynamic properties. In this scale model, we only need to represent the exterior surface of the airplane. We can ignore details such as the instrument panel or the engine. In order to train pilots for this new airplane, we also build a flight simulator. The flight simulator

needs to accurately represent the layout and behavior of flight instruments. In this case, however, details about the exterior of the plane can be ignored. Both the flight simulator and the scale model are much less complex than the airplane they represent. Modeling allows us to deal with complexity through a divide-and-conquer approach: For each type of problem we want to solve (e.g., testing aerodynamic properties, training pilots), we build a model that only focuses on the issues relevant to the problem. Generally, modeling focuses on building a model that is simple enough for a person to grasp completely. A rule of thumb is that each entity should contain at most 7 ± 2 parts [Miller, 1956].

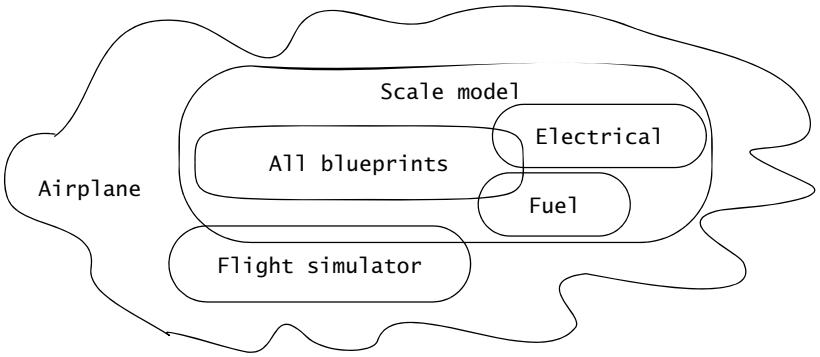

Figure 2-6 A model is an abstraction describing a subset of a system. A view depicts selected aspects of a model. Views and models of a single system may overlap each other.

Modeling also helps us deal with complexity by enabling us to incrementally refine simple models into more detailed ones that are closer to reality. In software engineering, as in all engineering disciplines, the model usually precedes the system. During analysis, we first build a model of the environment and of the common functionality that the system must provide, at a level that is understandable by the client. Then we refine this model, adding more details about the forms that the system should display, the layout of the user interface, and the response of the system to exceptional cases. The set of all models built during development is called the **system model**. If we did not use models, but instead started coding the system right away, we would have to specify all the details of the user interface before the client could provide us with feedback (and, thus, lose much time and resources when the client then introduces changes).

Unfortunately, even a model may become so complex that it is not easily understandable. We can continue to use the divide-and-conquer method to refine a complex model into simpler models. A **view** focuses on a subset of a model to make it understandable (Figure 2-6). For example, all the blueprints necessary to construct an airplane constitute a model. Excerpts necessary to explain the functioning of the fuel system constitute the fuel system view. Views may overlap: a view of the airplane representing the electrical wiring also includes the wiring for the fuel system.

Notations are graphical or textual rules for representing views. A UML class diagram is a graphical view of the object model. In wiring diagrams, each connected line represents a different wire or bundle of wires. In UML class diagrams, a rectangle with a title represents a class. A line between two rectangles represents a relationship between the two corresponding classes. Note that different notations can be used to represent the same view (Figure 2-7).

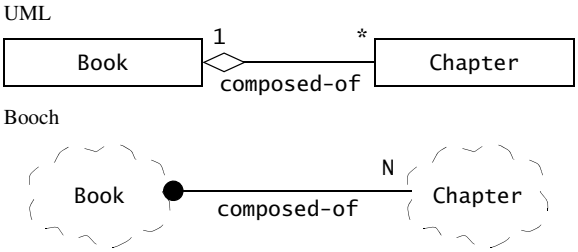

Figure 2-7 Example of describing a model with two different notations. The model includes two classes, `Book` and `Chapter`, with the relationship, `Book is composed of Chapters`. In UML, classes are depicted by rectangles and aggregation associations by a line terminated with a diamond. In the Booch notation, classes are depicted by clouds, and aggregation associations are depicted with a line terminated with a solid circle.

In software engineering, there are many other notations for modeling systems. UML describes a system in terms of classes, events, states, interactions, and activities. Data flow diagrams [De Marco, 1978] depict how data is retrieved, processed, and stored. Z Schemes [Spivey, 1992] represent the system in terms of invariants (conditions that never change) and in terms of what is true before and after the execution of an operation. Each notation is tailored for a different problem.

In the next sections, we focus in more detail on the process of modeling.

2.3.2 Data Types, Abstract Data Types, and Instances

A **data type** is an abstraction in the context of a programming language. A data type has a unique name that distinguishes it from other data types. It denotes a set of values that are members of the data type (i.e., the **instances** of the data type) and defines the structure and the operations valid in all instances of the data type. Data types are used in typed languages to ensure that only valid operations are applied to specific instances.

For example, the name `int` in Java corresponds to all the signed integers between -2^{32} and $2^{32} - 1$. The valid operations on this type are all the integer arithmetic operations (e.g., addition, subtraction, multiplication, division) and all the functions and methods that have parameters of type `int` (e.g., `mod`). The Java run-time environment throws an exception if a floating point operation is applied to an instance of the `int` data type (e.g., `trunc` or `floor`).

An **abstract data type** is a data type defined by an implementation-independent specification. Abstract data types enable developers to reason about a set of instances without looking at a specific implementation of the abstract data type. Examples of abstract data types are sets and sequences, which can be mathematically defined. A system may provide different implementations of the set abstract data type, each optimizing different criteria (e.g., memory consumption, insertion time). However, a developer using a set only needs to understand its semantics and need not be aware of the internal representation of the set. For example, the abstract data type `Person` may define the operations `getName()`,[1] `getSocialSecurityNumber()`, and `getAddress()`. The fact that the social security number of the person is stored as a number or as a string is not visible to the rest of the system. Such decisions are called **implementation decisions**.

2.3.3 Classes, Abstract Classes, and Objects

A **class** is an abstraction in object-oriented modeling and in object-oriented programming languages. Like abstract data types, a class encapsulates both structure and behavior. Unlike abstract data types, classes can be defined in terms of other classes by using inheritance. Assume we have a watch that also can function as a calculator. The class `CalculatorWatch` can then be seen as a refinement of the class `Watch`. This type of relationship between a base class and a refined class is called **inheritance**. The generalization class (e.g., `Watch`) is called the **superclass**, the specialized class (e.g., `CalculatorWatch`) is called the **subclass**. In an inheritance relationship, the subclass refines the superclass by defining new attributes and operations. In Figure 2-8, `CalculatorWatch` defines functionality for performing simple arithmetic that regular `Watches` do not have. Superclass and subclass are relative terms. The same class can be a subclass with respect to one class and a superclass with respect to another class.

When an inheritance relationship serves only to model shared attributes and operations, that is, if the generalization is not meant to be instantiated, the resulting class is called an **abstract class**. Abstract classes often represent generalized concepts in the application domain, and their names are italicized. For example, in chemistry, `Benzene` can be considered a class of molecules that belongs to the abstract class *`OrganicCompound`* (Figure 2-9). *`OrganicCompound`* is a generalization and does not correspond to any one molecule; that is, it does not have any instances. In Java, `Collection` is an abstract class providing a generalization for all collection classes. However, there are no instances of the class `Collection`. Rather, all collection objects are instances of one of the subclasses of Collection, such as `LinkedList`, `ArrayList`, or `HashMap`. Note that not all generalizations are abstract classes. For example, in Figure 2-8 the `Watch` class is not an abstract class as it has instances. When modeling software systems,

1. We refer to an operation by its name followed by its arguments in parentheses. If the arguments are not specified, we suffix the name of the operation by a pair of empty parentheses. We describe operations in detail in the next section.

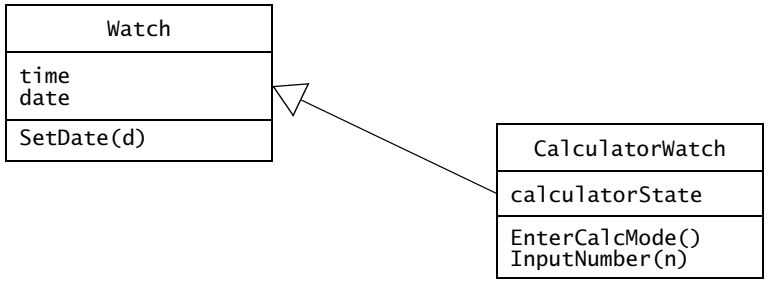

Figure 2-8 A UML class diagram depicting two classes, `Watch` and `CalculatorWatch`. `CalculatorWatch` is a refinement of `Watch`, providing calculator functionality not found in normal watches. In a UML class diagram, classes and objects are represented as boxes with three compartments: the first compartment depicts the name of the class, the second depicts its attributes, the third its operations. The second and third compartments can be omitted for brevity. An inheritance relationship is displayed by a line with a triangle. The triangle points to the superclass, and the other end is attached to the subclass.

abstract classes sometimes do not correspond to an existing application domain concept, but rather are introduced to reduce complexity in the model or to promote reuse.

A class defines the **operations** that can be applied to its instances. Operations of a superclass can be inherited and applied to the objects of the subclass as well. For example, in Figure 2-8, the operation `SetDate(d)`, setting the current date of a `Watch`, is also applicable to `CalculatorWatches`. The operation `EnterCalcMode()`, however, defined in the `CalculatorWatch` class, is not applicable in the `Watch` class.

A class defines the **attributes** that apply to all its instances. An attribute is a named slot in the instance where a value is stored. Attributes have a unique name within the class and the type. `Watches` have a `time` and a `date` attribute. `CalculatorWatches` have a `calculatorState` attribute.

An **object** is an instance of a class. An object has an identity and stores attribute values. Each object belongs to exactly one class. In UML, an instance is depicted by a rectangle with its name underlined. This convention is used throughout UML to distinguish between instances and

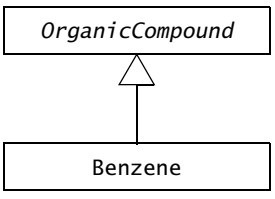

Figure 2-9 An example of abstract class (UML class diagram). *`OrganicCompound`* is never instantiated and only serves as a generalization class. The names of abstract classes are italicized.

classes.[2] In Figure 2-10, `simpleWatch1291` is an instance of `Watch`, and `calculatorWatch1515` is an instance of `CalculatorWatch`. Note that, although the operations of `Watch` are applicable to `calculatorWatch1515`, `calculatorWatch1515` is not an instance of the class `Watch`. The attributes of an object can be visible to other parts of the system in some programming languages. For example, Java allows the implementor to specify in great detail which attributes are visible and which are not.

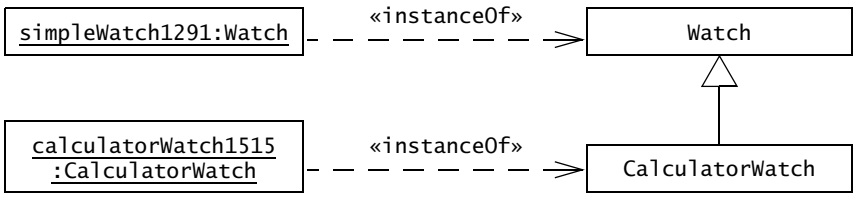

Figure 2-10 A UML class diagram depicting instances of two classes. `simpleWatch1291` is an instance of `Watch`. `calculatorWatch1515` is an instance of `CalculatorWatch`. Although the operations of `Watch` are also applicable to `calculatorWatch1515`, the latter is not an instance of the former.

2.3.4 Event Classes, Events, and Messages

Event classes are abstractions representing a kind of event for which the system has a common response. An **event**, an instance of an event class, is a relevant occurrence in the system. For example, an event can be a stimuli from an actor (e.g., "the `WatchUser` presses the left button"), a time-out (e.g., "after 2 minutes"), or the sending of a message between two objects. Sending a **message** is the mechanism by which the sending object requests the execution of an operation in the receiving object. The message is composed of a name and a number of arguments. The receiving object matches the name of the message to one of its operations and passes the arguments to the operation. Any results are returned to the sender.

For example, in Figure 2-11, the `:Watch` object computes the current time by getting the Greenwich time from the `:Time` object and the time difference from the `:TimeZone` object by sending the `getTime()` and the `getTimeDelta()` messages, respectively. Note that `:Watch` denotes an undesignated object of class `Watch`.

Events and messages are instances: they represent concrete occurrences in the system. Event classes are abstractions describing groups of events for which the system has a common response. In practice, the term "event" can refer to instances or classes. This ambiguity is resolved by examining the context in which the term is used.

2. Underlined strings are also used for representing Uniform Resource Locators (URLs). To improve readability, we do not use an underlined font in the text, but rather, we use the same font to denote instances and classes. In general, this ambiguity can be resolved by examining the context. In UML diagrams, however, we always use an underlined font to distinguish instances from classes.

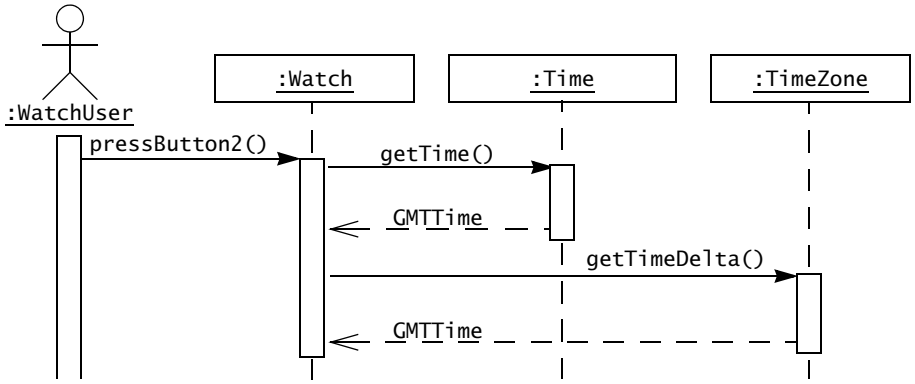

Figure 2-11 Examples of message sends (UML sequence diagram). The `Watch` object sends the `getTime()` message to a `Time` object to query the current Greenwich time. It then sends the `getTimeDelta()` message to a `TimeZone` object to query the difference to add to the Greenwich time. The dashed arrows represent the replies (i.e., message results that are sent back to the sender).

2.3.5 Object-Oriented Modeling

The **application domain** represents all aspects of the user's problem. This includes the physical environment, the users and other people, their work processes, and so on. It is critical for analysts and developers to understand the application domain for a system to accomplish its intended task effectively. Note that the application domain changes over time, as work processes and people change.[3]

The **solution domain** is the modeling space of all possible systems. Modeling in the solution domain represents the system design and object design activities of the development process. The solution domain model is much richer and more volatile than the application domain model. This is because the system is usually modeled in much more detail than the application domain. Emerging technologies (also called technology enablers), deeper understanding of implementation technology by the developers, and changes in requirements trigger changes to the solution domain models. Note that the deployment of the system can change the application domain as users develop new work processes to accommodate the system.

Object-oriented analysis is concerned with modeling the application domain. **Object-oriented design** is concerned with modeling the solution domain. Both modeling activities use the same representations (i.e., classes and objects). In object-oriented analysis and

3. The application domain is sometimes further divided into a user domain and a client domain. The client domain includes the issues relevant to the client, such as, operation cost of the system, impact of the system on the rest of the organization. The user domain includes the issues relevant to the end user, such as, functionality, ease of learning and of use.

design, the application domain model is also part of the system model. For example, an air traffic control system has a `TrafficController` class to represent individual users, their preferences, and log information. The system also has an `Aircraft` class to represent information associated with the tracked aircraft. `TrafficController` and `Aircraft` are application domain concepts that are encoded into the system (Figure 2-12).

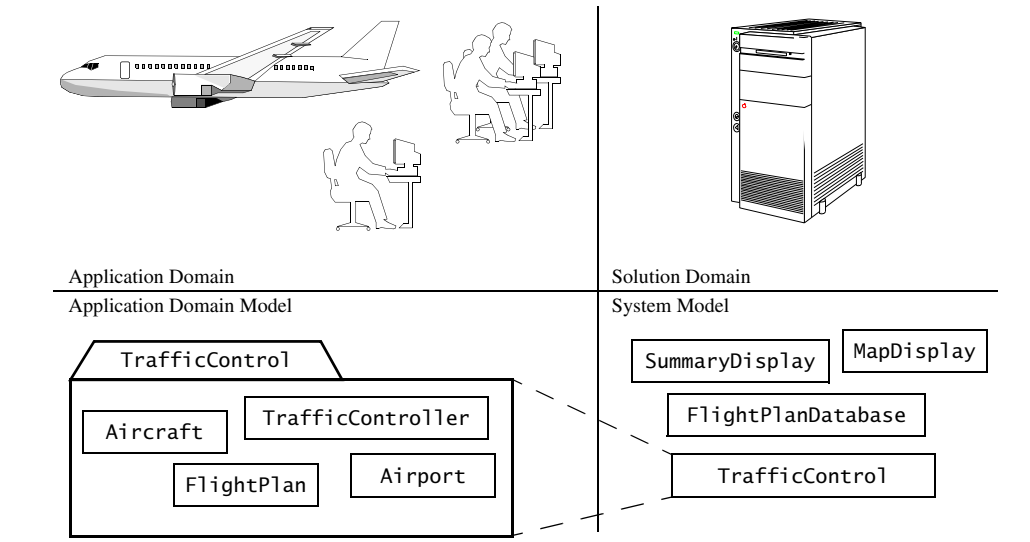

Figure 2-12 The application domain model represents entities of the environment that are relevant to an air traffic control system (e.g., aircraft, traffic controllers). The system model represents entities that are part of the system (e.g., map display, flight plan database). Note that in object-oriented analysis and design, the application domain model is part of the system model. The system model refines the application domain model to include solution domain concepts, such as `SummaryDisplay`, `MapDisplay`, and `FightPlan-Database`. (For more details, see Chapter 5, *Analysis*.)

Modeling the application domain and the solution domain with a single notation has advantages and disadvantages. On the one hand, it can be powerful: solution domain classes that represent application concepts can be traced back to the application domain. Moreover, these classes can be encapsulated into subsystems independent of other implementation concepts (e.g., user interface and database technology) and be packaged into a reusable toolkit of domain classes. On the other hand, using a single notation can introduce confusion because it removes the distinction between the real world and the model of it. The solution domain is bound to be simpler and biased toward the solution. To address this issue, we use a single notation and, in cases of ambiguity, we distinguish between the two domains. In most cases, we are referring to the model (e.g., "an `Aircraft` is associated with a `FlightPlan`" is a statement about the model).

2.3.6 Falsification and Prototyping

A model is a simplification of reality in the sense that irrelevant details are ignored. Relevant details, however, need to be represented. **Falsification** [Popper, 1992] is the process of demonstrating that relevant details have been incorrectly represented or not represented at all; that is, the model does not correspond to the reality it is supposed to represent.

The process of falsification is well known in other sciences: researchers propose different models of a reality, which are gradually accepted as an increasing amount of data supports the model, then rejected once a counterexample is found. Near the end of the 18th century, for example, it was discovered that the orbit of the planet Mercury did not exactly match the orbit predicted by Newton's theory of gravity. Later, Einstein's general theory of relativity predicted a slightly different orbit that better matched the results. In other words, Newton's theory was falsified in favor of Einstein's. Note, however, that we still use Newton's theory for practical applications on Earth, because the differences predicted by both theories are small in these situations and Newton's theory is much simpler. In other words, the details ignored by Newton's theory are not relevant for the scales we are accustomed to.

We can apply falsification to software system development as well. For example, a technique for developing a system is **prototyping**: when designing the user interface, developers construct a prototype that only simulates the user interface of a system. The prototype is then presented to potential users for evaluation—that is, falsification—and modified subsequently. In the first iterations of this process, developers are likely to throw away the initial prototype as a result of feedback from the users. In other terms, users falsify the initial prototype, a model of the future system, because it does not accurately represent relevant details.

Note that it is only possible to demonstrate that a model is incorrect. Although in some cases, it is possible to show mathematically that two models are equivalent, it is not possible to show that either of them correctly represents reality. For example, formal verification techniques can enable developers to show that a specific software implementation is consistent with a formal specification. However, only field testing and extended use can indicate that a system meets the needs of the client. At any time, system models can be falsified due to changes in the requirements, in the implementation technology, or in the environment.

2.4 A Deeper View into UML

We now describe in detail the five main UML diagrams we use in this book.

- **Use case diagrams** represent the functionality of the system from a user's point of view. They define the boundaries of the system (Section 2.4.1).
- **Class diagrams** represent the static structure of a system in terms of objects, their attributes, operations, and relationships (Section 2.4.2).
- **Interaction diagrams** represent the system's behavior in terms of interactions among a set of objects. They are used to identify objects in the application and implementation domains (Section 2.4.3).

- **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

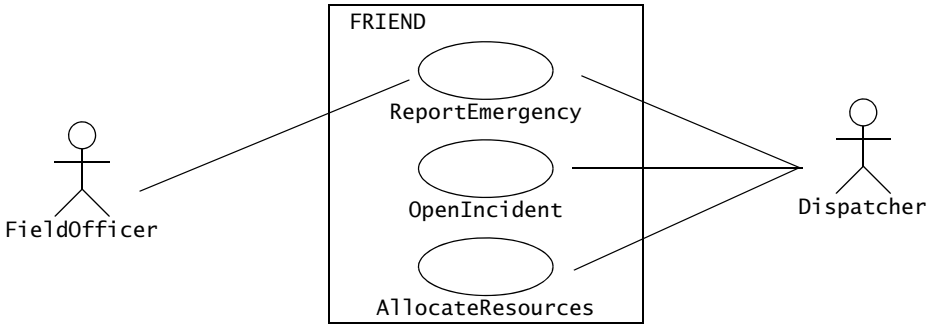

Figure 2-13 An example of a UML use case diagram for First Responder Interactive Emergency Navigational Database (`FRIEND`), an accident management system. Associations between actors and use cases denote information flows. These associations are bidirectional: they can represent the actor initiating 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 and orders additional units to the scene with the AllocateResources use case.

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.

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.

- 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».

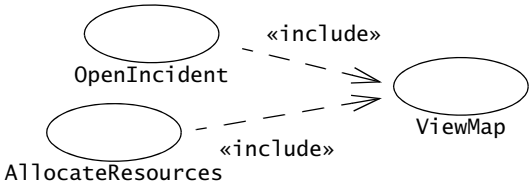

Figure 2-15 An example of an «include» relationship (UML use case diagram).

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).

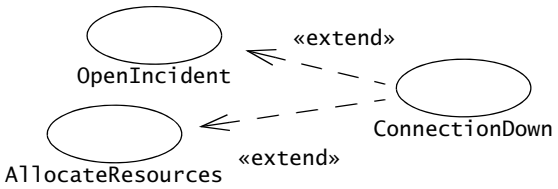

Figure 2-17 An example of an «extend» relationship (UML use case diagram).

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).

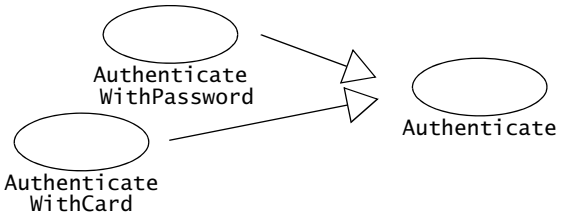

Figure 2-19 An example of an inheritance relationship (UML use case diagram). The `Authenticate` use case is a high-level use case describing, in general terms, the process of authentication. `AuthenticateWithPassword` and `AuthenticateWithCard` are two specializations of `Authenticate`.

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

Scenario name	warehouseOnFire
Participating actor instances	bob, alice:FieldOfficer john:Dispatcher
Flow of events	1. Bob, driving down main street in his patrol car, notices smoke coming out of a warehouse. His partner, Alice, activates the "Report Emergency" function from her FRIEND laptop. 2. Alice enters the address of the building, a brief description of its location (i.e., northwest corner), and an emergency level. In addition to a fire unit, she requests several paramedic units on the scene given that area appears to be relatively busy. She confirms her input and waits for an acknowledgment. 3. John, the Dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and two paramedic units to the Incident site and sends their estimated arrival time (ETA) to Alice. 4. Alice receives the acknowledgment and the ETA.

Figure 2-21 The warehouseOnFire scenario for the ReportEmergency use case.

with a lowercase letter. In the FRIEND example (Figures 2-22 and 2-23), Bob and Alice are field officers, represented in the system as FieldOfficer objects called bob:FieldOfficer and alice:FieldOfficer. FieldOfficer is a class describing all FieldOfficer objects, whereas Bob and Alice are represented by two individual FieldOfficer objects.

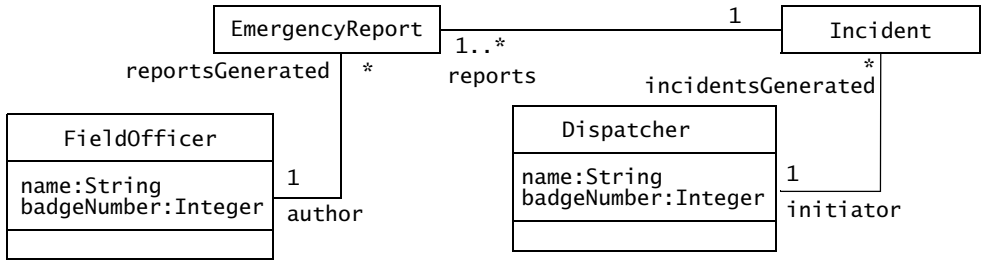

Figure 2-22 An example of a UML class diagram: classes that participate in the ReportEmergency use case. Detailed type information is usually omitted until object design (see Chapter 9, *Object Design: Specifying Interfaces*).

In Figure 2-22, the FieldOfficer class has two attributes: a name and a badgeNumber. This indicates that all FieldOfficer objects have these two attributes. In Figure 2-23, the bob:FieldOfficer and alice:FieldOfficer objects have specific values for these attributes: "Bob. D." and "Alice W.", respectively. In Figure 2-22, the FieldOfficer.name attribute is of type String, which indicates that only instances of String can be assigned to the

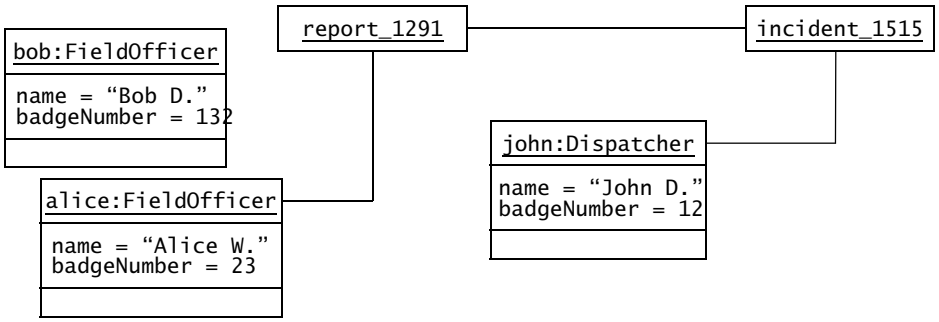

Figure 2-23 An example of a UML object diagram: objects that participate in `warehouseOnFire`.

`FieldOfficer.name` attribute. The type of an attribute is used to specify the valid range of values the attribute can take. Note that when attribute types are not essential to the definition of the system, attribute type decisions can be delayed until object design. This allows the developers to concentrate on the functionality of the system and to minimize the number of trivial changes when the functionality of the system is revised.

Associations and links

A **link** represents a connection between two objects. **Associations** are relationships between classes and represent groups of links. Each `FieldOfficer` object also has a list of `EmergencyReports` that has been written by the `FieldOfficer`. In Figure 2-22, the line between the `FieldOfficer` class and the `EmergencyReport` class is an association. In Figure 2-23, the line between the `alice:FieldOfficer` object and the `report_1291:EmergencyReport` object is a link. This link represents a state that is kept in the system to denote that `alice:FieldOfficer` generated `report_1291:EmergencyReport`.

In UML, associations can be symmetrical (bidirectional) or asymmetrical (unidirectional). All the associations in Figure 2-22 are symmetrical. Figure 2-24 depicts an example of one-directional association between `Polygon` and `Point`. The **navigation** arrow at the `Point` end of the association indicates that the system only supports navigation from the `Polygon` to the `Point`. In other words, given a specific `Polygon`, it is possible to query all `Points` that make up

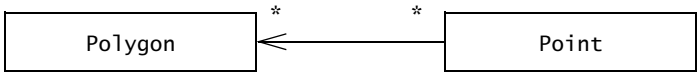

Figure 2-24 Example of a one-directional association. Developers usually omit navigation during analysis and add navigation information during object design, when they make such decisions (see Chapter 8, *Object Design: Reusing Pattern Solutions*, and Chapter 9, *Object Design: Specifying Interfaces*).

the `Polygon`. However, the navigation arrow indicates that, given a specific `Point`, it is not possible to find which `Polygons` the `Point` is part of. UML allows navigation arrows to be displayed on both ends of an association. By convention, however, an association without arrows indicates that navigation is supported in both directions.

Association class

Associations are similar to classes, in that they can have attributes and operations attached to them. Such an association is called an **association class** and is depicted by a class symbol that contains the attributes and operations and is connected to the association symbol with a dashed line. For example, in Figure 2-25, the allocation of `FieldOfficers` to an `Incident` is modeled as an association class with attributes `role` and `notificationTime`.

Any association class can be transformed into a class and simple associations as shown in Figure 2-26. Although this representation is similar to Figure 2-25, the association class

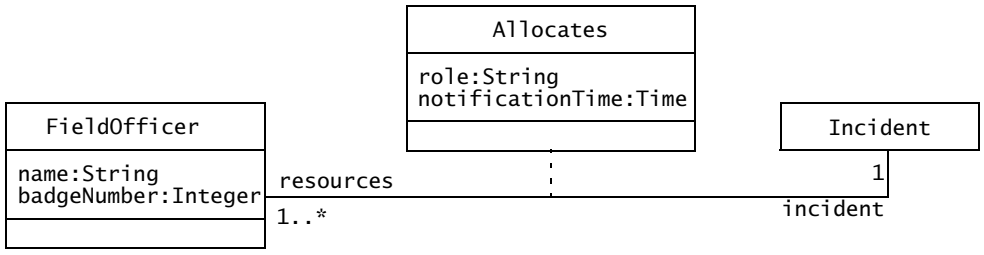

Figure 2-25 An example of an association class (UML class diagram).

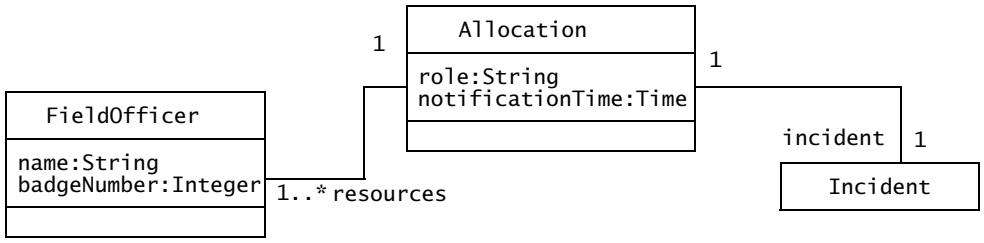

Figure 2-26 Alternative model for `Allocation` (UML class diagram).

representation is clearer in Figure 2-25: an association cannot exist without the classes it links. Similarly, the `Allocation` object cannot exist without a `FieldOfficer` and an `Incident`. Although Figure 2-26 carries the same information, this diagram requires careful examination of the association multiplicity. We examine such modeling trade-offs in Chapter 5, *Analysis*.

Roles

Each end of an association can be labeled by a **role**. In Figure 2-22, the roles of the association between `EmergencyReport` and `FieldOfficer` are `author` and `reportsGenerated`. Labeling the end of associations with roles allows us to distinguish among the multiple associations originating from a class. Moreover, roles clarify the purpose of the association.

Multiplicity

Each end of an association can be labeled by a set of integers indicating the number of links that can legitimately originate from an instance of the class connected to the association end. This set of integers is called the **multiplicity** of the association end. In Figure 2-22, the association end `author` has a multiplicity of `1`. This means that all `EmergencyReports` are written by exactly one `FieldOfficer`. In other terms, each `EmergencyReport` object has exactly one link to an object of class `FieldOfficer`. The multiplicity of the association end `reportsGenerated` role is "many," shown as a star. The "many" multiplicity is shorthand for `0..n`. This means that any given `FieldOfficer` may be the author of zero or more `EmergencyReports`.

In UML, an association end can have an arbitrary set of integers as a multiplicity. For example, an association could allow only a prime number of links and, thus, would have a multiplicity 1, 2, 3, 5, 7, 11, 13, and so forth. In practice, however, most of the associations we encounter belong to one of the following three types (see Figure 2-27).

- A **one-to-one association** has a multiplicity `1` on each end. A one-to-one association between two classes (e.g., `PoliceOfficer` and `BadgeNumber`) means that exactly one link exists between instances of each class (e.g., a `PoliceOfficer` has exactly one `BadgeNumber`, and a `BadgeNumber` denotes exactly one `PoliceOfficer`).
- A **one-to-many association** has a multiplicity `1` on one end and `0..n` (also represented by a star) or `1..n` on the other. A one-to-many association between two classes (e.g., `FireUnit` and `FireTruck`) denotes composition (e.g., a `FireUnit` owns one or more `FireTrucks`, a `FireTruck` is owned exactly by one `FireUnit`).
- A **many-to-many association** has a multiplicity `0..n` or `1..n` on both ends. A many-to-many association between two classes (e.g., `FieldOfficer` and `IncidentReport`) denotes that an arbitrary number of links can exist between instances of the two classes (e.g., a `FieldOfficer` can write many `IncidentReports`, an `IncidentReport` can be written by many `FieldOfficers`). This is the most complex type of association.

Adding multiplicity to associations increases the amount of information we capture from the application or the solution domain. Specifying the multiplicity of an association becomes critical when we determine which use cases are needed to manipulate the application domain objects. For example, consider a file system made of `Directories` and `Files`. A `Directory` can

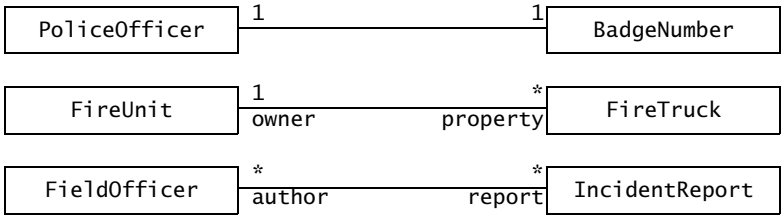

Figure 2-27 Examples of multiplicity (UML class diagram). The association between `PoliceOfficer` and `BadgeNumber` is one-to-one. The association between `FireUnit` and `FireTruck` is one-to-many. The association between `FieldOfficer` and `IncidentReport` is many-to-many.

contain any number of *`FileSystemElements`*. A *`FileSystemElement`* is a concept that denotes either a `Directory` or a `File`. In case of a strictly hierarchical system, a *`FileSystemElement`* is part of exactly one `Directory`, which we denote with a one-to-many multiplicity (Figure 2-28).

If, however, a `File` or a `Directory` can be simultaneously part of more than one `Directory`, we need to represent the aggregation of *`FileSystemElement`* into `Directories` as a many-to-many association (see Figure 2-29).

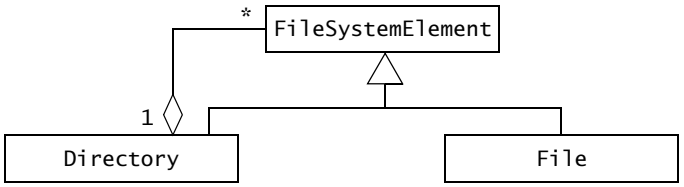

Figure 2-28 Example of a hierarchical file system. A `Directory` can contain any number of *`FileSystemElements`* (a *`FileSystemElement`* is either a `File` or a `Directory`). A given *`FileSystemElement`*, however, is part of exactly one `Directory`.

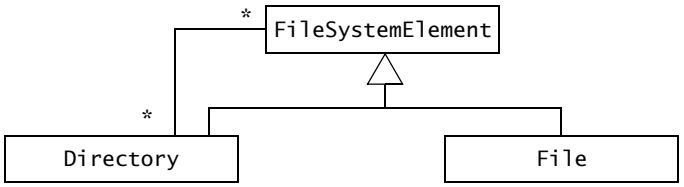

Figure 2-29 Example of a nonhierarchical file system. A `Directory` can contain any number of *`FileSystemElements`* (a *`FileSystemElement`* is either a `File` or a `Directory`). A given *`FileSystemElement`* can be part of many `Directories`.

This discussion may seem to be considering detailed issues that could be left for later activities in the development process. The difference between a hierarchical file system and a nonhierarchical one, however, is also in the functionality it offers. If a system allows a given `File` to be part of multiple `Directories`, we need to define a use case describing how a user adds an existing `File` to existing `Directories` (e.g., the Unix `link` command or the Macintosh `MakeAlias` menu item). Moreover, use cases removing a `File` from a `Directory` must specify whether the `File` is removed from one `Directory` only or from all `Directories` that reference it. Note that a many-to-many association can result in a substantially more complex system.

Aggregation

Associations are used to represent a wide range of connections among a set of objects. In practice, a special case of association occurs frequently: **aggregation**. For example, a `State` contains many `Counties`, which in turn contain many `Townships`. A `PoliceStation` is composed of `PoliceOfficers`. A `Directory` contains a number of `Files`. Such relationships could be modeled using a one-to-many association. Instead, UML provides the concept of an aggregation, denoted by a simple line with a diamond at the container end of the association (see Figures 2-28 and 2-30). One-to-many associations and aggregations, although similar, cannot be used interchangeably: aggregations denote hierarchical aspects of the relationship and can have either one-to-many or many-to-many multiplicity, whereas one-to-many associations imply a peer relationship. For example, in Figure 2-30, the `PoliceOfficers` are part of the `PoliceStation`. In Figure 2-22, a `FieldOfficer` writes zero or many `EmergencyReports`. However, the `FieldOfficer` is not *composed* `EmergencyReports`. Consequently, we use an association in the later case and an aggregation in the former case.

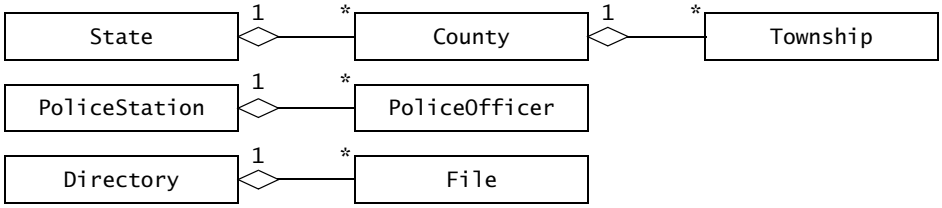

Figure 2-30 Examples of aggregations (UML class diagram). A `State` contains many `Counties`, which in turn contains many `Townships`. A `PoliceStation` has many `PoliceOfficers`. A file system `Directory` contains many `Files`.

Qualification

Qualification is a technique for reducing multiplicity by using keys. Associations with a `0..1` or `1` multiplicity are easier to understand than associations with a `0..n` or `1..n` multiplicity. Often, in the case of a one-to-many association, objects on the "many" side can be distinguished

from one another using a name. For example, in a hierarchical file system, each file belongs to exactly one directory. Each file is uniquely identified by a name in the context of a directory. Many files can have the same name in the context of the file system; however, two files cannot share the same name within the same directory. Without qualification (see top of Figure 2-31), the association between `Directory` and `File` has a one multiplicity on the `Directory` side and a zero-to-many multiplicity on the `File` side. We reduce the multiplicity on the `File` side by using the `filename` attribute as a key, also called a **qualifier** (see top of Figure 2-31). The relationship between `Directory` and `File` is called a **qualified association**.

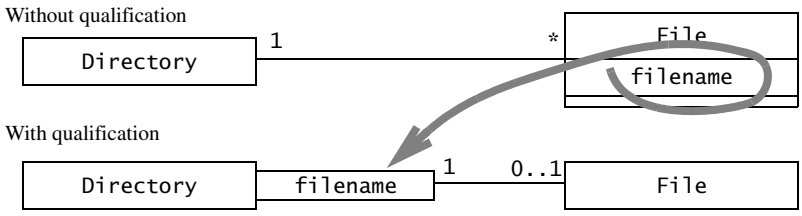

Figure 2-31 Example of how a qualified association reduces multiplicity (UML class diagram). Adding a qualifier clarifies the class diagram and increases the conveyed information. In this case, the model including the qualification denotes that the name of a file is unique within a directory.

Reducing multiplicity is always preferable, as the model becomes clearer and fewer cases have to be taken into account. Developers should examine each association that has a one-to-many or many-to-many multiplicity to see if a qualifier can be added. Often, these associations can be qualified with an attribute of the target class (e.g., `filename` in Figure 2-31).

Inheritance

Inheritance is the relationship between a general class and one or more specialized classes. Inheritance enables us to describe all the attributes and operations that are common to a set of classes. For example, `FieldOfficer` and `Dispatcher` both have `name` and `badgeNumber` attributes. However, `FieldOfficer` has an association with `EmergencyReport`, whereas `Dispatcher` has an association with `Incident`. The common attributes of `FieldOfficer` and `Dispatcher` can be modeled by introducing a *`PoliceOfficer`* class that is specialized by the `FieldOfficer` and the `Dispatcher` classes (see Figure 2-32). *`PoliceOfficer`*, the generalization, is called a **superclass**. `FieldOfficer` and `Dispatcher`, the specializations, are called the **subclasses**. The subclasses **inherit** the attributes and operations from their parent class. Abstract classes (defined in Section 2.3.3) are distinguished from concrete classes by *`italicizing`* the name of abstract classes. In Figure 2-32, *`PoliceOfficer`* is an abstract class. Abstract classes are used in object-oriented modeling to classify related concepts, thus reducing the overall complexity of the model.

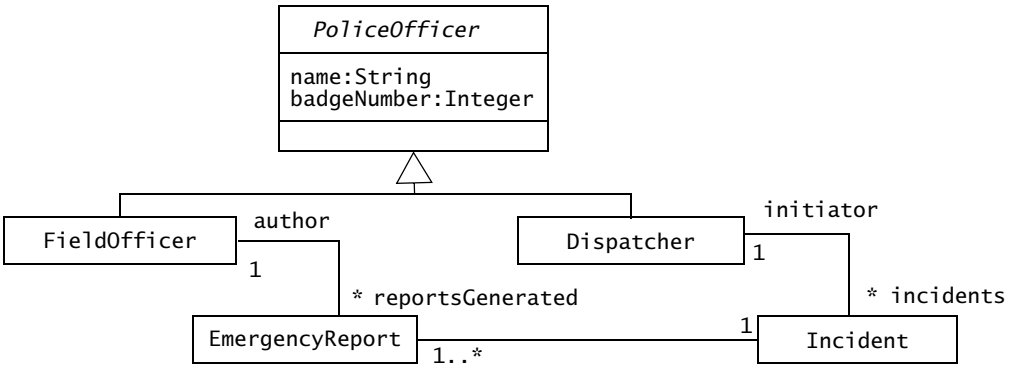

Figure 2-32 An example of an inheritance (UML class diagram). *PoliceOfficer* is an abstract class which defines the common attributes and operations of the `FieldOfficer` and `Dispatcher` classes.

Object behavior is specified by **operations**. An object requests the execution of an operation from another object by sending it a **message**. The message is matched up with a **method** defined by the class to which the receiving object belongs or by any of its superclasses. The methods of a class in an object-oriented programming language are the implementations of these operations.

The distinction between operations and methods allows us to distinguish between the specification of behavior (i.e., an operation) and its implementation (i.e., a set of methods that are possibly defined in different classes in the inheritance hierarchy). For example, the class `Incident` in Figure 2-33 defines an operation, called `assignResource()`, which, given a `FieldOfficer`, creates an association between the receiving `Incident` and the specified `Resource`. The `assignResource()` operation may also have a side effect such as sending a notification to the newly assigned `Resource`. The `close()` operation of `Incident` is responsible

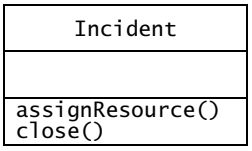

Figure 2-33 Examples of operations provided by the `Incident` class (UML class diagram).

for closing the `Incident`. This includes going over all the resources that have been assigned to the incident over time and collecting their reports. Although UML distinguishes operations from methods, in practice, developers usually do not and simply refer to methods.

Applying class diagrams

Class diagrams are used for describing the structure of a system. During analysis, software engineers build class diagrams to formalize application domain knowledge. Classes represent participating objects found in use cases and interaction diagrams, and describe their attributes and operations. The purpose of analysis models is to describe the scope of the system and discover its boundaries. For example, using the class diagram pictured in Figure 2-22, an analyst could examine the multiplicity of the association between `FieldOfficer` and `EmergencyReport` (i.e., one `FieldOfficer` can write zero or more `EmergencyReports`, but each `EmergencyReport` is written by exactly one `FieldOfficer`) and ask the user whether this is correct. Can there be more than one author of an `EmergencyReport`? Can there be anonymous reports? Depending on the answer from the user, the analyst would then change the model to reflect the application domain. The development of analysis models is described in Chapter 5, *Analysis*.

Analysis models do not focus on implementation. Concepts such as interface details, network communication, and database storage are not represented. Class diagrams are refined during system design and object design to include classes representing the solution domain. For example, the developer adds classes representing databases, user interface windows, adapters around legacy code, optimizations, and so on. The classes are also grouped into subsystems with well-defined interfaces. The development of design models is described in Chapter 6, *System Design: Decomposing the System*, Chapter 8, *Object Design: Reusing Pattern Solutions*, Chapter 9, *Object Design: Specifying Interfaces*, and Chapter 10, *Mapping Models to Code*.

2.4.3 Interaction Diagrams

Interaction diagrams describe patterns of communication among a set of interacting objects. An object interacts with another object by sending **messages**. The reception of a message by an object triggers the execution of a method, which in turn may send messages to other objects. **Arguments** may be passed along with a message and are bound to the parameters of the executing method in the receiving object. In UML, interaction diagrams can take one of two forms: sequence diagrams or communication diagrams.

Sequence diagrams represent the objects participating in the interaction horizontally and time vertically. For example, consider a watch with two buttons (hereafter called `2Bwatch`). Setting the time on `2Bwatch` requires the actor `2BWatchOwner` to first press both buttons simultaneously, after which `2Bwatch` enters the set time mode. In the set time mode, `2Bwatch` blinks the number being changed (e.g., the hours, minutes, seconds, day, month, or year). Initially, when the `2BWatchOwner` enters the set time mode, the hours blink. If the actor presses the first button, the next number blinks (e.g, if the hours are blinking and the actor presses the first button, the hours stop blinking and the minutes start blinking). If the actor presses the second button, the blinking number is incremented by one unit. If the blinking number reaches the end of its range, it is reset to the beginning of its range (e.g., assume the minutes are blinking and its current value is `59`, its new value is set to `0` if the actor presses the second button). The actor exits the set time mode by pressing both buttons simultaneously. Figure 2-34 depicts a

sequence diagram for an actor setting his `2Bwatch` one minute ahead. Each column represents an object that participates in the interaction. Messages are shown by solid arrows. Labels on solid arrows represent message names and may contain arguments. Activations (i.e., executing methods) are depicted by vertical rectangles. The actor who initiates the interaction is shown in the left-most column. The messages coming from the actor represent the interactions described in the use case diagrams. If other actors communicate with the system during the use case, these actors are represented on the right-hand side and can receive messages. Although for simplicity, interactions among objects and actors are uniformly represented as messages, the modeler should keep in mind that interactions between actors and the system are of a different nature than interactions among objects.

Sequence diagrams can be used to describe either an abstract sequence (i.e., all possible interactions) or concrete sequences (i.e., one possible interaction, as in Figure 2-34). When describing all possible interactions, sequence diagrams provide notations for iterations and conditionals. An iteration is denoted by a combined fragment labeled with the `loop` operator (see Figure 2-35). An alternative is denoted by a combined fragment containing a partition for each alternative. The alternatives are selected by guards on the first message of the partition (`[i>0]` and `[else]` in Figure 2-35). If `i` is positive, the top alternative of the `alt` combined fragment is executed and the `op1()` message is sent. Otherwise, the bottom alternative is executed and the `op2()` message is sent.

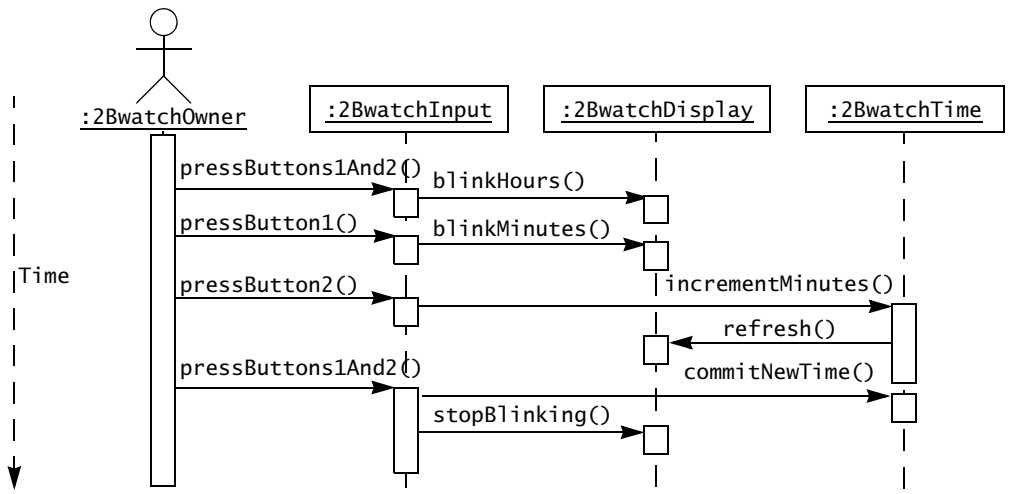

Figure 2-34 Example of a sequence diagram: setting the time on `2Bwatch`.

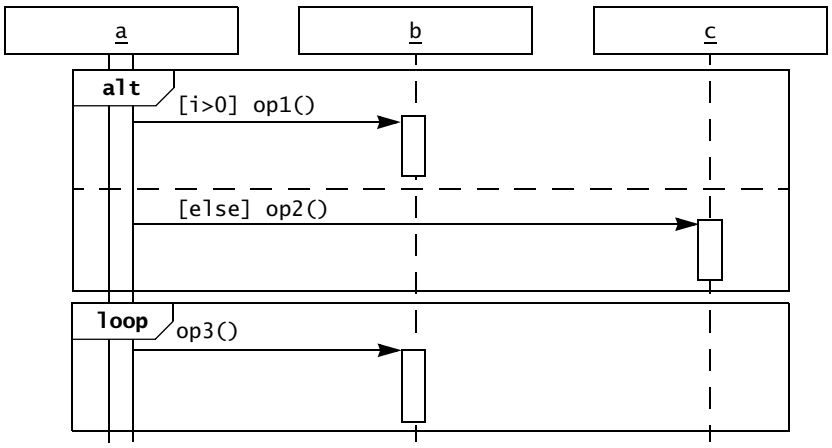

Figure 2-35 Examples of conditions and iterators in sequence diagrams.

Communication diagrams depict the same information as sequence diagrams. Communication diagrams represent the sequence of messages by numbering the interactions. On one hand, this removes the need for geometrical constraints on the objects and results in a more compact diagram. On the other hand, the sequence of messages becomes more difficult to follow. Figure 2-36 depicts the communication diagram that is equivalent to the sequence diagram of Figure 2-34.

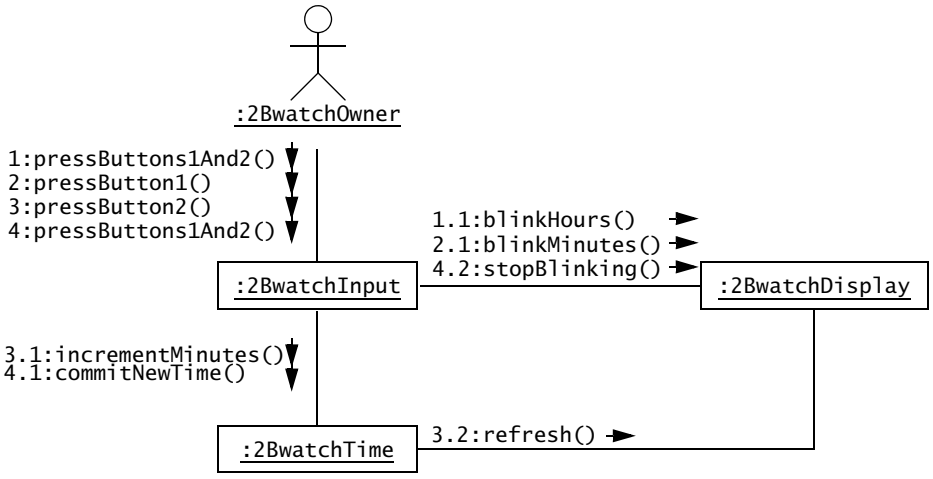

Figure 2-36 Example of a communication diagram: setting the time on `2Bwatch`. This diagram represents the same use case as the sequence diagram of Figure 2-34.

Applying interaction diagrams

Interaction diagrams describe interactions among several objects. One of the main reasons for constructing an interaction diagram is to uncover the responsibilities of the classes in the class diagrams and to discover even new classes. In other words, the interaction diagram helps the developer in deciding which objects require particular operations. Typically, there is an interaction diagram for every use case with focus on the event flow. The developer identifies the objects that participate in the use case, and assigns pieces of the use case behavior to the objects in the form of operations.

The class diagram and the associated interaction diagrams are usually constructed in tandem after the initial class diagram has been defined. This process often also leads to refinements in the use case (e.g., correcting ambiguous descriptions, adding missing behavior) and, consequently, the discovery of more objects and more services. We describe in detail the use of interaction diagrams in Chapter 5, *Analysis*.

2.4.4 State Machine Diagrams

A UML **state machine** is a notation for describing the sequence of states an object goes through in response to external events. UML state machines are extensions of the finite state machine model. On one hand, state machines provide notation for nesting states and state machines (i.e., a state can be described by a state machine). On the other hand, state machines provide notation for binding transitions with message sends and conditions on objects. UML state machines are largely based on Harel's statecharts [Harel, 1987] and have been adapted for use with object models [Douglass, 1999]. UML state machines can be used to represent any Mealy or Moore state machine.

A **state** is a condition satisfied by the attributes of an object. For example, an `Incident` object in `FRIEND` can exist in four states: `Active`, `Inactive`, `Closed`, and `Archived` (see Figure 2-37). An active `Incident` denotes a situation that requires a response (e.g., an ongoing fire, a traffic accident). An inactive `Incident` denotes a situation that was handled, but for which reports are yet to be written (e.g., the fire has been put out, but damage estimates have not yet been completed). A closed `Incident` denotes a situation that has been handled and documented. An archived `Incident` is a closed `Incident` whose documentation has been moved to off-site storage. In this example, we can represent these four states with a single attribute in the `Incident` class—a `status` attribute that can take any of four values: `Active`, `Inactive`, `Closed`, and `Archived`. In general, a state can be computed from the values of several attributes.

A **transition** represents a change of state triggered by events, conditions, or time. For example, Figure 2-37 depicts three transitions: from the `Active` state into the `Inactive` state, from the `Inactive` state to the `Closed` state, and from the `Closed` state to the `Archived` state.

A state is depicted by a rounded rectangle. A transition is depicted by an open arrow connecting two states. States are labeled with their name. A small solid black circle indicates the initial state. A circle surrounding a small solid black circle indicates a final state.

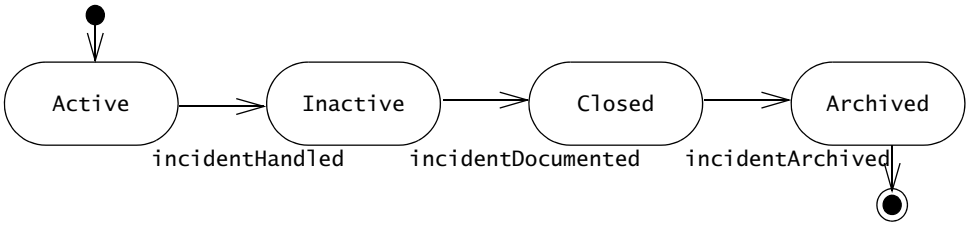

Figure 2-37 A UML state machine diagram for the `Incident` class.

Figure 2-38 displays another example, a state machine for the `2Bwatch` (for which we constructed a sequence diagram in Figure 2-34). At the highest level of abstraction, `2Bwatch` has two states, `MeasureTime` and `SetTime`. `2Bwatch` changes states when the user presses and releases both buttons simultaneously. During the transition from the `SetTime` state to the `MeasureTime` state, `2Bwatch` beeps. This is indicated by the action `/beep` on the transition. When `2Bwatch` is first powered on, it is in the `SetTime` state. This is modeled by making `SetTime` the initial state. When the battery of the watch runs out, the `2Bwatch` is permanently out of order. This is indicated with a final state. In this example, transitions can be triggered by an event (e.g., `pressBothButtons`) or by the passage of time (e.g., `after 2 min`.).

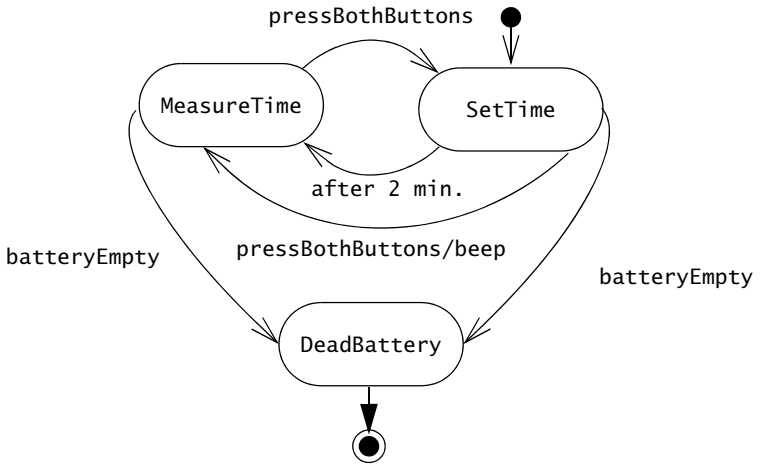

Figure 2-38 State machine diagram for `2Bwatch` set time function.

Figure 2-39 depicts a refined state machine diagram for the `2Bwatch` depicted in Figure 2-38 using actions to denote the behavior within the states. **Actions** are fundamental units of processing that can take a set of inputs, produce a set of outputs, and can change the state of the system. Actions normally take a short amount of time to execute and are not interruptable.

For example, an action can be realized by an operation call. Actions can occur in three places in a state machine:

- when a transition is taken (e.g., `beep` when the transition between `SetTime` and `MeasureTime` is fired on the `pressBothButtons` event)
- when a state is entered (e.g., `blink hours` in the `SetTime` state in Figure 2-39)
- when a state is exited (e.g., `stop blinking` in the `SetTime` state in Figure 2-39).

During a transition, the exit actions of the source state are executed first, then the actions associated with the transition are executed, then the entry actions of the destination state are executed. The exit and entry actions are always executed when a state is exited or entered, respectively. They do not depend on the specific transition that was used to exit or enter the state.

An **internal transition** is a transition that does not leave the state. Internal transitions are triggered by events and can have actions associated with them. However, the firing of an internal transition does not result in the execution of any exit or entry actions. For example, in Figure 2-39, the `SetTime` state has two internal transitions, one associated with pressing button 1 and one associated with pressing the right button.

An **activity** is a coordinated set of actions. A state can be associated an activity that is executed as long as an object resides in this state. While an action is short and non-interruptable, an activity can take a substantial amount of time and is interrupted when a transition exiting the state is fired. Activities are associated with state using the `do` label and are placed inside the state where they executed. For example, in Figure 2-39, `count ticks` is an activity associated with the `MeasureTime` state.

Nested state machines reduce complexity. They can be used instead of internal transitions. In Figure 2-40, the current number is modeled as a **nested state**, whereas actions corresponding to modifying the current number are modeled using internal transitions. Note that each state could be modeled as a nested state machine. For example, the `BlinkHours` state machine would have 24 substates that correspond to the hours in the day; transitions between these states would correspond to pressing the second button.

Applying state machine diagrams

State machine diagrams are used to represent nontrivial behavior of a subsystem or an object. Unlike interaction diagrams that focus on the events impacting the behavior of a set of objects, state machine diagrams make explicit which attribute or set of attributes have an impact on the behavior of a single object. State machines are used to identify object attributes and to refine the behavior description of an object, and interaction diagrams are used to identify participating objects and the services they provide. State machine diagrams can also be used during system and object design to describe solution domain objects with interesting behavior. We describe the use of state machine diagrams in detail in Chapter 5, *Analysis*, and Chapter 6, *System Design: Decomposing the System*.

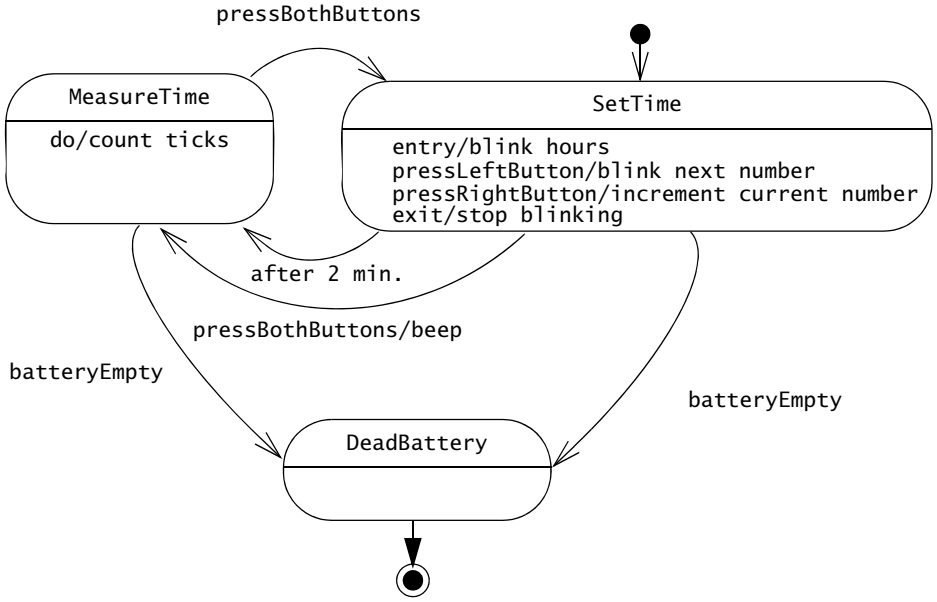

Figure 2-39 Internal transitions associated with the `SetTime` state (UML state machine diagram).

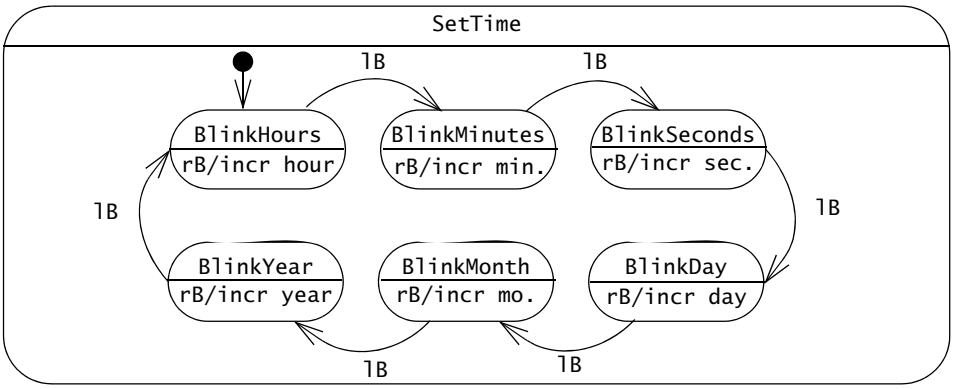

Figure 2-40 Refined state machine associated with the `SetTime` state (UML state machine diagram). `lB` and `rB` correspond to pressing the left and right button, respectively.

2.4.5 Activity Diagrams

UML activity diagrams represent the sequencing and coordination of lower level behaviors. An activity diagram denotes how a behavior is realized in terms of one or several sequences of activities and the object flows needed for coordinating the activities. Activity

diagrams are hierarchical: an **activity** is made out of either an action or a graph of subactivities and their associated object flow. Figure 2-41 is an activity diagram corresponding to the state diagram in Figure 2-37. Rounded rectangles represent actions and activities. Edges between activities represent control flow. An activity can be executed only after all predecessor activities completed.

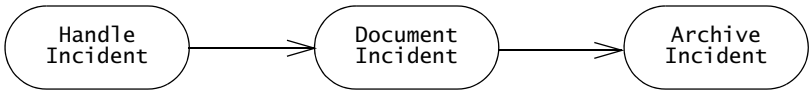

Figure 2-41 A UML activity diagram for `Incident`. During the action `HandleIncident`, the `Dispatcher` receives reports and allocates resources. Once the `Incident` is closed, the `Incident` moves to the `DocumentIncident` activity during which all participating `FieldOfficers` and `Dispatchers` document the `Incident`. Finally, the `ArchiveIncident` activity represents the archival of the `Incident` related information onto slow access medium.

Control nodes coordinate control flows in an activity diagram, providing mechanisms for representing decisions, concurrency, and synchronization. The main control nodes we use are decisions, fork nodes, and join nodes.

Decisions are branches in the control flow. They denote alternatives based on a condition of the state of an object or a set of objects. Decisions are depicted by a diamond with one or more incoming open arrows and two or more outgoing arrows. The outgoing edges are labeled with the conditions that select a branch in the control flow. The set of all outgoing edges from a decision represents the set of all possible outcomes. In Figure 2-42, a decision after the `OpenIncident` action selects between three branches: If the incident is of high priority and if it is a fire, the `FireChief` is notified. If the incident is of high priority and is not a fire, the `PoliceChief` is notified. Finally, if neither condition is satisfied, that is, if the `Incident` is of low priority, no superior is notified and the resource allocation proceeds.

Fork nodes and **join nodes** represent concurrency. Fork nodes denotes the splitting of the flow of control into multiple threads while join nodes denotes the synchronization of multiple threads and their merging of the flow of control into a single thread. For example, in Figure 2-43, the actions `AllocateResources`, `Coordinate-Resources`, and `DocumentIncident` may all occur in parallel. However, they can only be initiated after the `OpenIncident` action, and the `ArchiveIncident` action may only be initiated after all other activities have been completed.

Activities may be grouped into **swimlanes** (also called **activity partitions**) to denote the object or subsystem that implements the actions. Swimlanes are represented as rectangles enclosing a group of actions. Transitions may cross swimlanes. In Figure 2-44, the `Dispatcher` swimlane groups all the actions that are performed by the `Dispatcher` object. The `FieldOfficer` swimlane denotes that the `FieldOfficer` object is responsible for the `DocumentIncident` action.

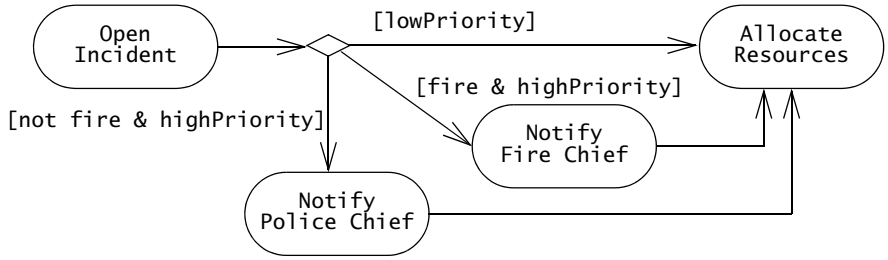

Figure 2-42 Example of decision in the `OpenIncident` process. If the `Incident` is a fire and is of high priority, the `Dispatcher` notifies the `FireChief`. If it is a high-priority `Incident` that is not a fire, the `PoliceChief` is notified. In all cases, the `Dispatcher` allocates resources to deal with the `Incident`.

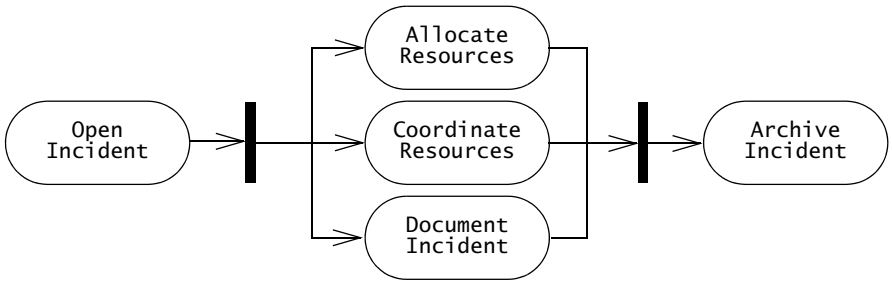

Figure 2-43 An example of fork and join nodes in a UML activity diagram.

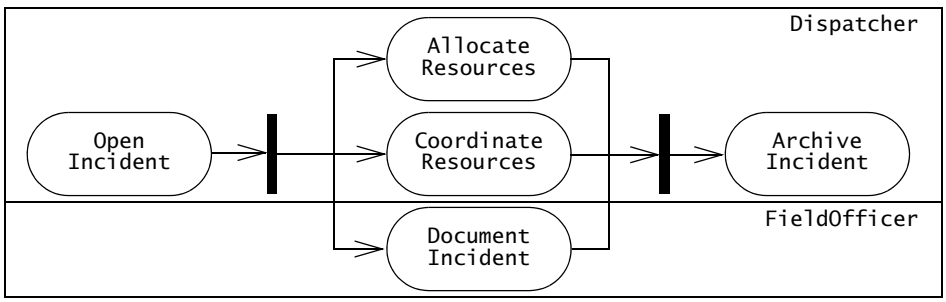

Figure 2-44 An example of swimlanes in a UML activity diagram.

Applying activity diagrams

Activity diagrams provide a task-centric view of the behavior of a set of objects. They can be used, for example, to describe sequencing constraints among use cases, sequential activities among a group of objects, or the tasks of a project. In this book, we use activity diagrams to describe the activities of software development in Chapter 14, *Project Management*, and Chapter 15, *Software Life Cycle*.

2.4.6 Diagram Organization

Models of complex systems quickly become complex as developers refine them. The complexity of models can be dealt with by grouping related elements into **packages**. A package is a grouping of model elements, such as use cases or classes, defining scopes of understanding.

For example, Figure 2-45 depicts use cases of the `FRIEND` system, grouped by actors. Packages are displayed as rectangles with a tab attached to their upper-left corner. Use cases dealing with incident management (e.g., creating, resource allocation, documentation) are grouped in the `IncidentManagement` package. Use cases dealing with incident archive (e.g., archiving an incident, generating reports from archived incidents) are grouped in the `IncidentArchive` package. Use cases dealing with system administration (e.g., adding users, registering end stations) are grouped in the `SysAdministration` package. This enables the client and the developers to organize use cases into related groups and to focus on only a limited set of use cases at a time.

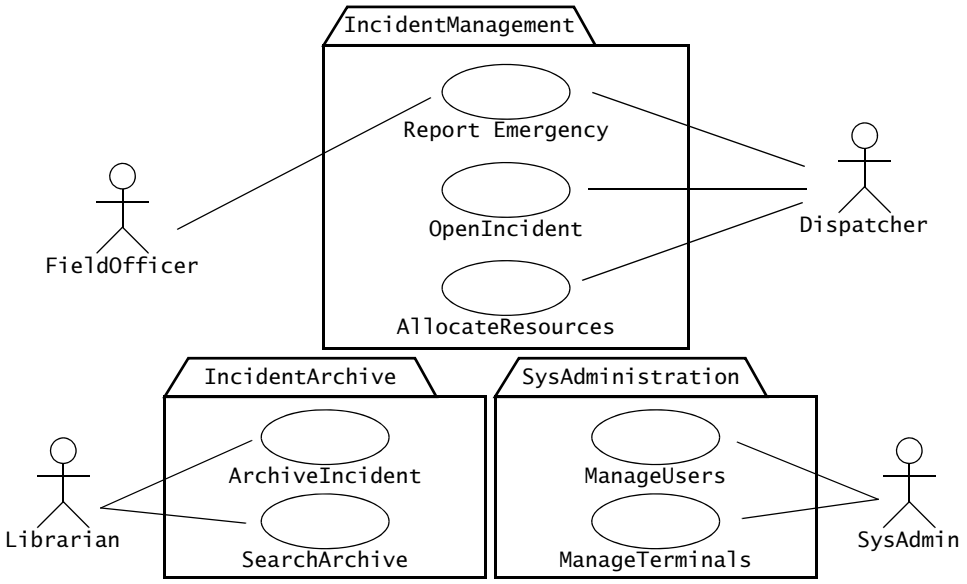

Figure 2-45 Example of packages: use cases of `FRIEND` organized by actors (UML use case diagram).

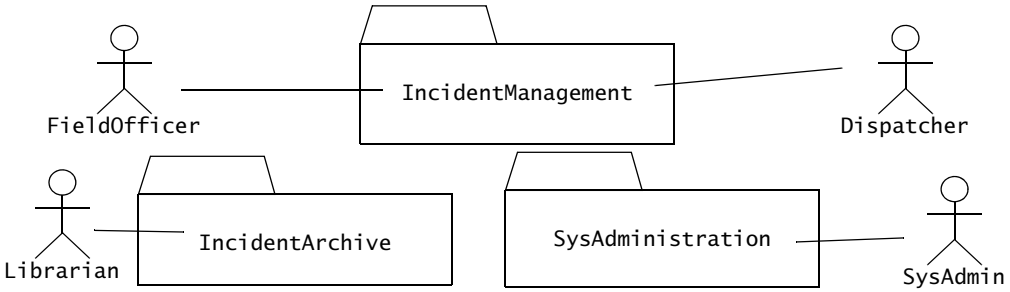

Figure 2-46 Example of packages. This figure displays the same packages as Figure 2-45 except that the details of each packages are suppressed (UML use case diagram).

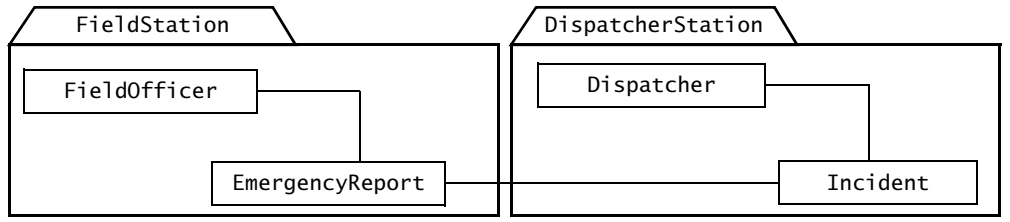

Figure 2-47 Example of packages. The `FieldOfficer` and `EmergencyReport` classes are located in the `FieldStation` package, and the `Dispatcher` and `Incident` classes are located on the `DispatcherStation` package.

Figures 2-46 and 2-47 are examples of class diagrams using packages. Classes from the `ReportEmergency` use case are organized according to the site where objects are created. `FieldOfficer` and `EmergencyReport` are part of the `FieldStation` package, and `Dispatcher` and `Incident` are part of the `DispatcherStation`. Figure 2-47 displays the packages with the model elements they contain, and Figure 2-46 displays the same information without the contents of each package. Figure 2-46 is a higher-level picture of the system and can be used for discussing system-level issues, whereas Figure 2-47 is a more detailed view that can be used to discuss the content of specific packages.

Packages are used to deal with complexity in the same way a user organizes files and subdirectories into directories. However, packages are not necessarily hierarchical: the same class may appear in more than one package. To reduce inconsistencies, classes (more generally model elements) are owned by exactly one package, whereas the other packages are said to refer to the modeling element. Note that packages are organizing constructs, not objects. They have no associated behavior and cannot send and receive messages.

A **note** is a comment attached to a diagram. Notes are used by developers for attaching information to models and model elements. This is an ideal mechanism for recording

outstanding issues relevant to a model, clarifying a complex point, or recording to-dos or reminders. Although notes have no semantics per se, they are sometimes used to express constraints that cannot otherwise be expressed in UML. Figure 2-48 is an example of a note.

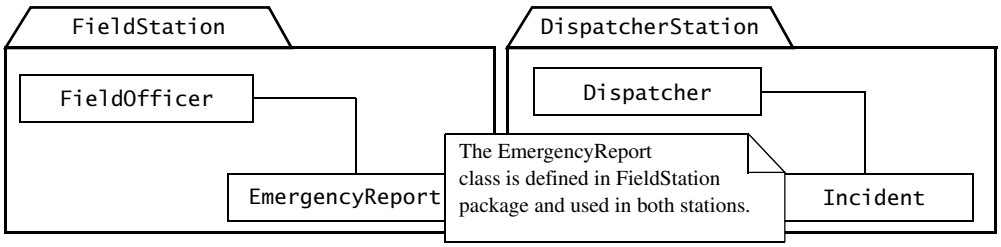

Figure 2-48 An example of a note. Notes can be attached to a specific element in a diagram.

2.4.7 Diagram Extensions

The goal of the UML designers was to provide a set of notations to model a broad class of software systems. They also recognized that a fixed set of notations could not achieve this goal, because it is impossible to anticipate the needs encountered in all application and solution domains. For this reason, UML provides a number of extension mechanisms enabling the modeler to extend the language. In this section, we describe two such mechanisms, **stereotypes** and **constraints**.

A **stereotype** is an extension mechanism that allows developers to classify model elements in UML. A stereotype is represented by string enclosed by guillemets (e.g., «boundary») and attached to the model element to which it applies, such as a class or an association. Formally, attaching a stereotype to a model element is semantically equivalent to creating a new class in the UML meta-model (i.e., the model that represents the constructs of UML). This enables modelers to create new kinds of building blocks that are needed in their domain. For example, during analysis, we classify objects into three types: entity, boundary, and control. Entity, boundary, and control objects have the same structure (i.e., they have attributes, operations, and associations), but serve different purposes. The base UML language only includes one type of object. To represent these three types, we use the stereotypes «entity», «boundary», and «control» (Figure 2-49). The «entity», «boundary», and «control» stereotypes are described in Chapter 5, *Analysis*. Another example is the relationships among use cases. As we saw in Section 2.4.1, include relationships in use case diagrams are denoted with a dashed open arrow and the «include» stereotype.

A **constraint** is a rule that is attached to a UML model element restricting its semantics. This allows us to represent phenomena that cannot otherwise be expressed with UML. For example, in Figure 2-50, an `Incident` may be associated with one or more `EmergencyReports` from the field. However, it is important that the `Dispatchers` are able to view the reports chronologically. We represent the chronological ordering of `EmergencyReport` to `Incident` with

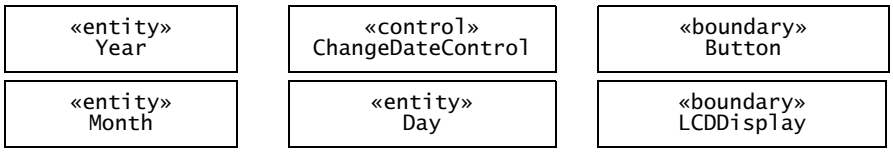

Figure 2-49 Examples of stereotypes (UML class diagram).

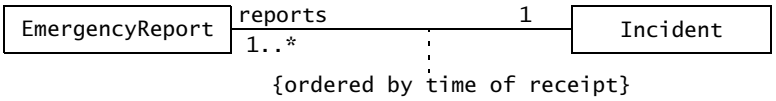

Figure 2-50 An example of constraint (UML class diagram).

the constraint `{ordered by time of receipt}`. Constraints can be expressed as an informal string or by using a formal language such as OCL (Object Constraint Language, [OMG, 2009]). We describe OCL and the use of constraints in Chapter 9, *Object Design: Specifying Interfaces*.

2.5 Further Readings

The historic roots of modeling notations can be traced back to structured analysis [De Marco, 1978] and structured design [Yourdon & Constantine, 1975], which is based on functional decomposition. These methods were based data flow diagrams [De Marco, 1978]. Data flow diagrams are quite important for software engineers who need to maintain legacy systems designed with structured analysis techniques.

UML came out of the teachings and efforts of many researchers and practitioners, some of whom we cited earlier in this chapter. The efforts of Booch, Jacobson, and Rumbaugh enabled a broadly accepted unified notation. Their earlier works [Booch, 1994], [Jacobson et al., 1992], [Rumbaugh et al., 1991] give much insight into the roots of object-oriented analysis and design and still provide valuable knowledge about object-oriented modeling.

Because it was designed to address a broad range of systems and concern, UML is a complex standard. In this chapter, we focused on the basic elements of UML that you need to understand before proceeding with the next chapters. For further information on UML, refer to the following books:

UML Distilled [Fowler, 2003] is a brief introduction to UML and illustrated with many examples. For readers without any knowledge of UML, this book is a useful overview to get into the notation quickly.

The *Unified Modeling Language User Guide* [Booch et al., 2005] is a comprehensive presentation of UML by its principal designers. It covers much more material than *UML Distilled* and is more appropriate for the advanced modeler. As the *UML User Guide* has fewer examples, *UML Distilled* is more appropriate for novices.

The *OMG Unified Modeling Language Superstructure* [OMG, 2009] is the official specification of UML. It is continuously maintained by a revision task force that is responsible for clarifying ambiguities, correcting errors, and resolving inconsistencies found by the UML community.

2.6 Exercises

2-1 Consider an ATM system. Identify at least three different actors that interact with this system.

2-2 Can the system under consideration be represented as an actor? Justify your answer.

2-3 What is the difference between a scenario and a use case? When do you use each construct?

2-4 Draw a use case diagram for a ticket distributor for a train system. The system includes two actors: a traveler who purchases different types of tickets, and a central computer system that maintains a reference database for the tariff. Use cases should include `BuyOneWayTicket`, `BuyWeeklyCard`, `BuyMonthlyCard`, and `UpdateTariff`. Also include the following exceptional cases: `TimeOut` (i.e., traveler took too long to insert the right amount), `TransactionAborted` (i.e., traveler selected the cancel button without completing the transaction), `DistributorOutOfChange`, and `DistributorOutOfPaper`.

2-5 Write the flow of events and specify all fields for the use case `UpdateTariff` that you drew in Exercise 2-4. Do not forget to specify any relationships.

2-6 Draw a class diagram representing a book defined by the following statement: "A book is composed of a number of parts, which in turn are composed of a number of chapters. Chapters are composed of sections." Focus only on classes and relationships.

2-7 Add multiplicity to the class diagram you produced in Exercise 2-6.

2-8 Draw an object diagram representing the first part of this book (i.e., Part I, *Getting Started*). Make sure that the object diagram you draw is consistent with the class diagram of Exercise 2-6.

2-9 Extend the class diagram of Exercise 2-6 to include the following attributes:

- a book includes a publisher, publication date, and an ISBN
- a part includes a title and a number
- a chapter includes a title, a number, and an abstract
- a section includes a title and a number.

2-10 Consider the class diagram of Exercise 2-9. Note that the `Part`, `Chapter`, and `Section` classes all include title and number attributes. Add an abstract class and an inheritance relationship to factor out these two attributes into the abstract class.

2-11 Draw a class diagram representing the relationship between parents and children. Take into account that a person can have both a parent and a child. Annotate associations with roles and multiplicities.

2-12 Draw a class diagram for bibliographic references. Use the references in Appendix C, *Bibliography*, to test your class diagram. Your class diagram should be as detailed as possible.

2-13 Draw a sequence diagram for the `warehouseOnFire` scenario of Figure 2-21. Include the objects `bob`, `alice`, `john`, `FRIEND`, and instances of other classes you may need. Draw only the first five message sends.

2-14 Draw a sequence diagram for the `ReportIncident` use case of Figure 2-14. Draw only the first five message sends. Make sure it is consistent with the sequence diagram of Exercise 2-13.

2-15 Consider the process of ordering a pizza over the phone. Draw an activity diagram representing each step of the process, from the moment you pick up the phone to the point where you start eating the pizza. Do not represent any exceptions. Include activities that others need to perform.

2-16 Add exception handling to the activity diagram you developed in Exercise 2-15. Consider at least three exceptions (e.g., delivery person wrote down wrong address, delivery person brings wrong pizza, store out of anchovies).

2-17 Consider the software development activities which we described in Section 1.4 in Chapter 1, *Introduction to Software Engineering*. Draw an activity diagram depicting these activities, assuming they are executed strictly sequentially. Draw a second activity diagram depicting the same activities occurring incrementally (i.e., one part of the system is analyzed, designed, implemented, and tested completely before the next part of the system is developed). Draw a third activity diagram depicting the same activities occurring concurrently.

References

[Booch, 1994] G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1994.

[Booch et al., 2005] G. Booch, J. Rumbaugh, & I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 2005.

[Coad et al., 1995] P. Coad, D. North, & M. Mayfield, *Object Models: Strategies, Patterns, & Applications*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[Constantine & Lockwood, 2001] L.L. Constantine & L.A.D. Lockwood, "Structure and style in use cases for user interface design," in M. van Harmelen (ed.), *Object-Oriented User Interface Design*, 2001.

[De Marco, 1978] T. De Marco, *Structured Analysis and System Specification*, Yourdon, New York, 1978.

[Douglass, 1999] B.P. Douglass, *Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time Applications*, Addison-Wesley, Reading, MA, 1999.

[Fowler, 2003] M. Fowler, *UML Distilled: A Brief Guide To The Standard Object Modeling Language*, 3rd ed., Addison-Wesley, Reading, MA, 2003.

[Harel, 1987] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, pp. 231–274, 1987.

[Jacobson et al., 1992] I. Jacobson, M. Christerson, P. Jonsson, & G. Overgaard, *Object-Oriented Software Engineering—A Use Case Driven Approach,* Addison-Wesley, Reading, MA, 1992.

[Martin & Odell, 1992] J. Martin & J. J. Odell, *Object-Oriented Analysis and Design*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[Mellor & Shlaer, 1998] S. Mellor & S. Shlaer, *Recursive Design Approach*, Prentice Hall, Upper Saddle River, NJ, 1998.

[Miller, 1956] G.A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *Psychological Review*, Vol. 63, pp. 81–97, 1956.

[OMG, 2009] Object Management Group, *OMG Unified Modeling Language Superstructure. Version 2.2*, http://www.omg.org.

[Popper, 1992] K. Popper, *Objective Knowledge: An Evolutionary Approach*, Clarendon, Oxford, 1992.

[Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Spivey, 1992] J. M. Spivey, *The Z Notation, A Reference Manual.* 2nd ed., Prentice Hall International, Hertfordshire, U.K., 1992.

[Wirfs-Brock et al., 1990] R. Wirfs-Brock, B. Wilkerson, & L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[Yourdon & Constantine, 1975] E. Yourdon & L. Constantine, *Structured Design*, Prentice Hall, Englewood Cliffs, NJ, 1975.