# 2. Modeling with UML

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 become critical as the cost of miscommunication increases rapidly.

For a notation to enable accurate communication, it must come with a *well defined* semantics, *well suited* for representing a given aspect of a system, and *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 will be prone to misunderstandings as each user imposes its own interpretation. We selected UML (Unified Modeling Language, [10]) as a notation for this book, given that it has a well defined semantics, 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 describes the concepts of modeling in general and object-oriented modeling in particular. We then describe four fundamental UML notations that we use throughout the book: Use case diagrams, Sequence diagrams, Class diagrams, and Statechart diagrams. For each of these notations, we describe its basic semantics and provide examples. We revisit these notations in more detail in later chapters as we describe the processes that use them. Specialized notations that are used in only one chapter are introduced later (e.g., Pert and Gantt charts in Chapter 4, *Project Management* and Component and Deployment diagrams in Chapter 8, *System Design*).

## 2.1. Introduction

UML (Unified Modeling Language, [10]) is a notation that resulted from the unification of OMT (Object Modeling Technique, [Rumbaugh et al., 1991]), OOSE (Object-Oriented Software Engineering, [Jacobson et al., 1992]), and Booch [Booch, 1994]. UML has also be influenced by other object-oriented notations, such as those introduced by Shlaer/Mellor [Mellor & Shlaer, 1998], Coad/Yourdon [Coad et al. 1995], Wirfs-Brock [Wirfs-Brock et al. 1990], and Martin/Odell [Martin & Odell, 1992]. UML has been designed for a broad range of applications and software processes. Hence, it provides constructs for many different kinds of systems and activities (e.g., real time systems, distributed systems, requirements analysis, system design, deployment).

System development focuses on three different models of the system:

1. The functional model, represented in UML with use case diagrams, describes the functionality of the system from the user's point of view.
2. The object model, represented in UML with class diagrams, describes the structure of a system in terms of objects, attributes, associations, and operations.
3. The dynamic model, represented in UML with sequence diagrams and statechart diagrams, describes the internal behavior of the system. Sequence diagrams describe behavior as a sequence of messages exchanged among a *set of objects*, whereas statechart diagrams describe behavior in terms of states of an *individual object* and the possible transitions between states.

In this chapter, we describe UML diagrams for representing these models (use case diagrams, sequence diagrams, class diagrams, and statechart diagrams). Introducing you to these notations represents an interesting challenge: on the one hand, understanding the purpose of a notation requires you to be familiar with the processes that uses it, on the other hand, you need to understand the notation before we can introduce you to these processes. To address this issue, we introduce UML iteratively. In the next section, we first provide you with an overview of the four basic UML notations. In Section 2.3, we introduce you to the fundamental ideas of modeling. In Section 2.4, we revisit the four basic UML notations in light of modeling concepts. In subsequent chapters, we examine these notations in more detail when we introduce the processes that use them. First, let us have a brief look at these four UML notations.

## 2.2. An overview of UML

### 2.2.1. Use case diagrams

Use cases are used during requirements elicitation and requirements analysis to define 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). For example, Figure 1 depicts a use case diagram for 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 may change the battery of the watch (with the `ChangeBattery` use case). We describe use case diagrams in more detail in Section 2.4.1.

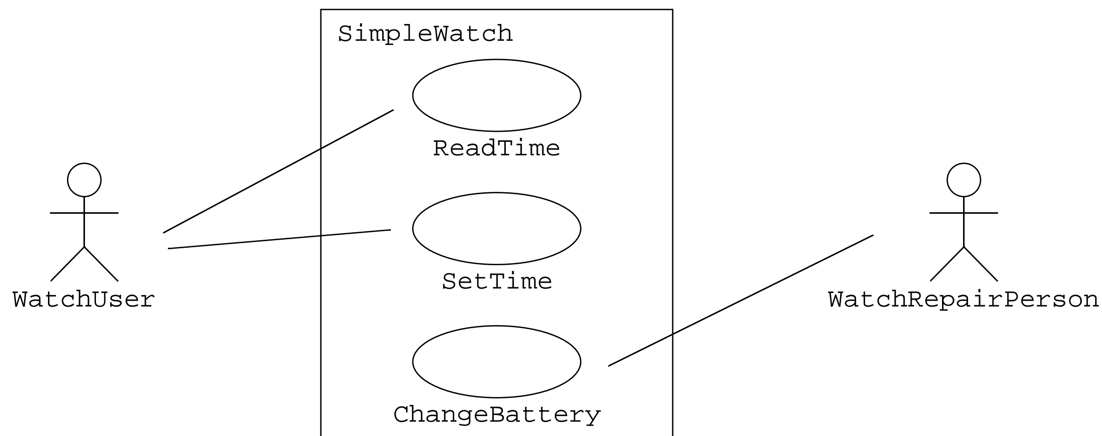


**FIGURE 1.** A UML use case diagram describing the functionality of a simple watch. The `WatchUser` actor may either consult the time on her watch (with the `ReadTime` use case) or set the time (with the `SetTime` use case). However, only the `WatchRepairPerson` actor may change the battery of the watch (with the `ChangeBattery` use case).

### 2.2.2. Sequence diagrams

Once use cases have been defined, sequence diagrams are used to formalize the behavior of the system. Sequence diagrams are used to identify objects which participate in the use case and the services they provide. We call these objects ***participating objects***, to indicate that they are the set of objects which participate in a specific use case. A sequence diagram

represents the interactions that take place among these objects. For example, Figure 2 is 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 are 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 ahead. The SetTime use case terminates when the WatchUser presses both buttons simultaneously. We describe sequence diagrams in more detail in Section 2.4.2

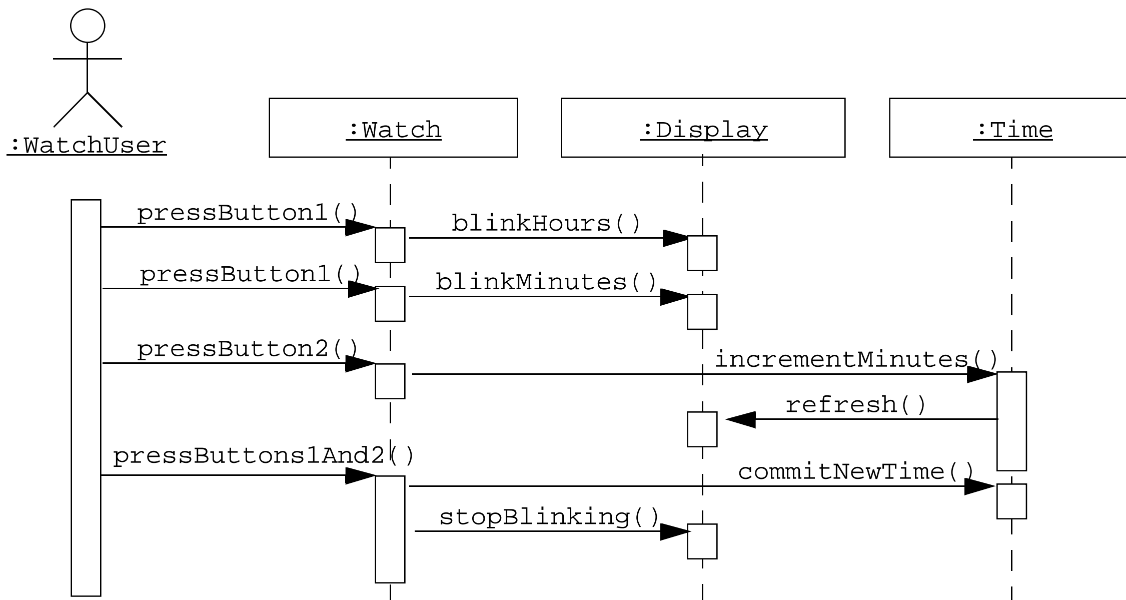


**FIGURE 2.** A UML sequence diagram for the SetTime use case of the SimpleWatch system. The left most column represents the WatchUser actor who initiates the use case. Labeled arrows are stimuli that an actor or an object sends to other objects. In this case, the WatchUser presses button 1 twice and button 2 once to set her watch a minute ahead. The SetTime use case terminates when the WatchUser presses both buttons simultaneously.

### 2.2.3. Class diagrams

Once participating objects have been identified, we use class diagrams to describe the structure of the system. Classes are abstractions that specify the common structure and behavior of a set of objects. Objects are entities that are created, modified, and destroyed during the execution of the system. Objects have state which includes the values of its attributes and its relationships with other objects. Class diagrams describe the objects,

classes, attributes, and relationships found in the system. For example, Figure 3 is a class diagram describing the elements of all the watches of the SimpleWatch class. These watch objects all have a relationship to an object of the PushButton class, an object of the LCDDisplay class, an object of the Time class, and an object of the Battery class. The numbers on the relationships (called multiplicity) denote the number of relationships each SimpleWatch object with an object of a given class. For example, a SimpleWatch has exactly two PushButtons, one LCDDisplay, two Batteries and one Time. Similarly, all PushButton, LCDDisplay, Time, and Battery objects are associated to exactly one SimpleWatch object. We describe class diagrams in more detail in Section 2.4.3.

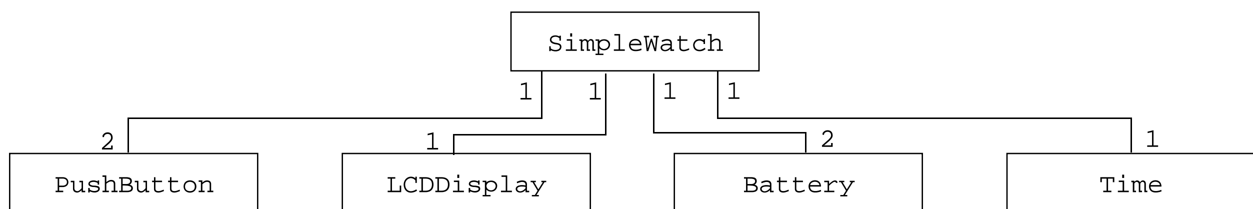


**FIGURE 3.** A UML class diagram describing the elements of a simple watch.

### 2.2.4. Statechart diagrams

Statechart diagrams describe the behavior of an individual object as a number of states and transitions. 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 4 is a statechart diagram for the SetTime use case. Note that this diagram represents slightly different information than the sequence diagram of Figure 2. The sequence diagram focuses on the messages exchanged between objects as a result of external events. The statechart diagram focuses on the transitions between states as a result of external events. Statecharts can also be used to describe activities of the development process. Statechart diagrams are described in more detail in Section 2.4.4.

This concludes our first walkthrough of the four basic notations of UML. Now we go into more detail: Section 2.3 introduces you to basic modeling concepts, including the definition of systems, models, types, and instances, abstraction, and falsification. Sections 2.4.1 to 2.4.4 describe in detail Use case diagrams, Sequence diagrams, Class diagrams, and Statechart diagrams. We quote the definitions of important terms from the UML Glossary [10] and we illustrate their use with a simple example. Section 2.4.5 describes miscellaneous constructs, such as packages and notes, that are used in all diagrams. We use these four notations throughout the book to describe software systems, work products, processes, and

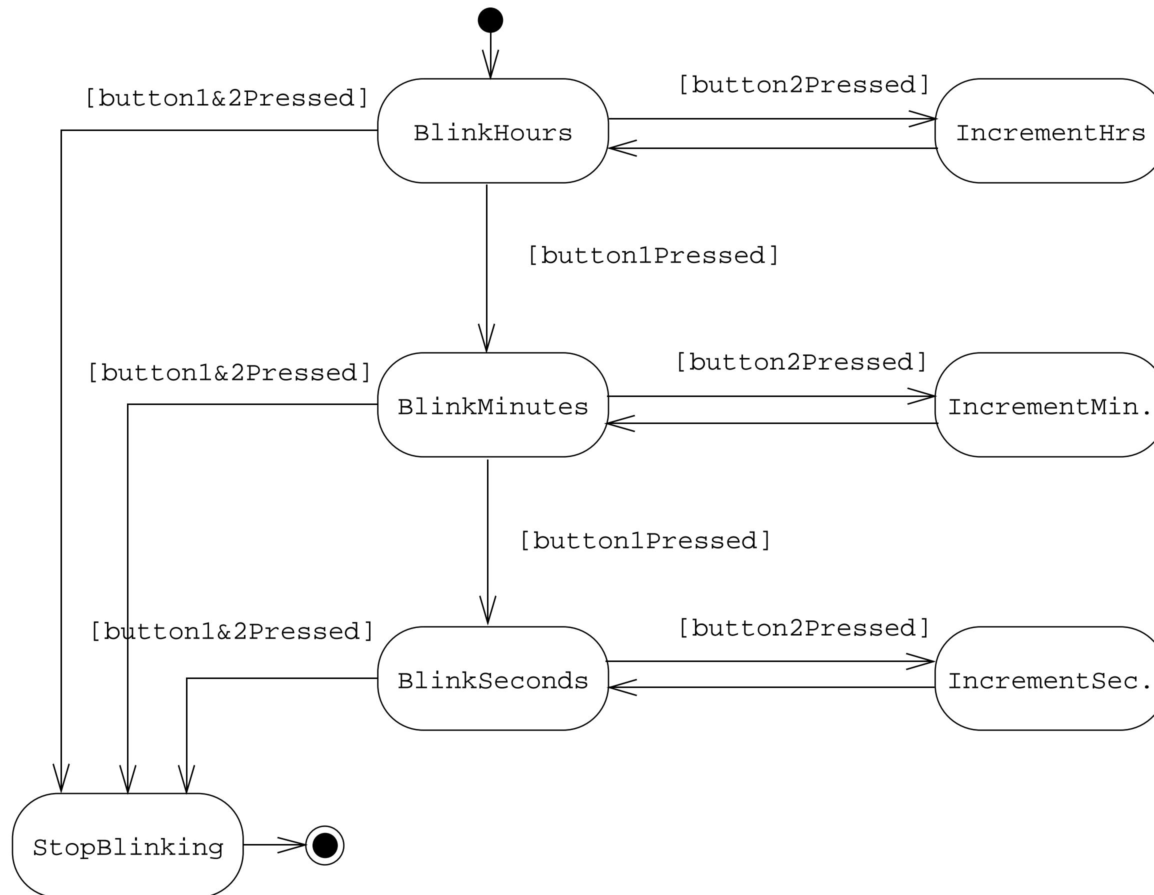


**FIGURE 4.** UML Statechart diagram for `SetTime` use case of the 2Bwatch.

organizations. By the consistent use of a small set of notations, we hope to provide you with better operational knowledge of UML. In order to avoid overloading you with too many notations, we introduce specialized notations (e.g., UML Component diagrams in Chapter 8, *System Design*) in later chapters as you need them.

## 2.3. Modeling

In this section we describe the basic concepts of modeling. We first define systems and distinguish models from the systems they represent. We then define the terms *concept* and *phenomena*. We explain their relationship to programming languages where they are called types and classes, instances and objects, respectively. Finally, we briefly 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 that is designed to accomplish a specific purpose. 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, hands, is designed to measure time. A payroll system, composed of a mainframe computer, printers, disks, software, and payroll staff, is designed to issue salary checks for employees of a company. Parts of a system can be in turn be considered as simpler systems called ***subsystems***. For example, 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 it can be fully comprehended 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 mean 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 building 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 represent accurately 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 general rule of thumb is that each entity should contain at most $7 \pm 2$ parts.

Unfortunately, even a model may become so complex that it contains many more parts than $7 \pm 2$, that is, that it is not easily understandable. As with systems, we apply the same divide-and-conquer approach to deal with the complexity of models. A ***view*** focuses on parts of a

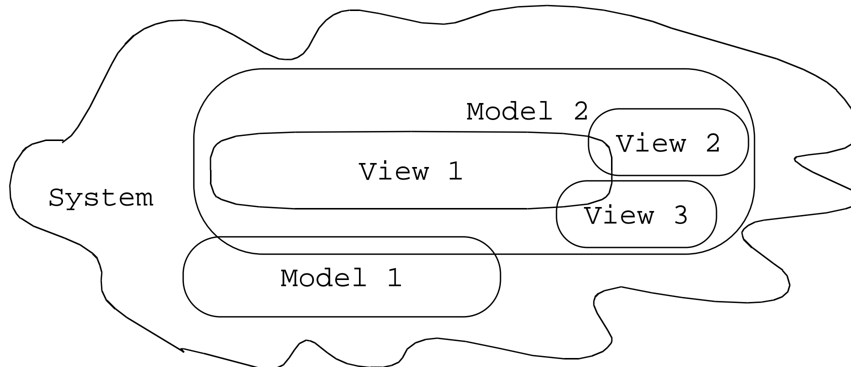


**FIGURE 5.** A model is an abstraction describing a subset of a system. A view depicts selected aspects of a model. Views and models of a single system may overlap each other.

model to make it understandable (see Figure 5). For example, all the blueprints necessary to construct an airplane constitute a model. Excerpts necessary to explain the functioning of the fuel system constitute a view (i.e., 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. 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 (see Figure 6 and Figure 7).

In software development, there are also many other notations for modeling systems. UML describes a system in terms of classes, states, interactions, and activities. The Z notation [Spivey, 1989] describes a system using set theory. Data flow diagrams [De Marco, 1978] depict how data is retrieved, processed, and stored. Each notation is tailored for a different problem.

In the next sections, we focus in more detail on the process of modeling. We examine the definitions of *concept* and *phenomenon* introduced by Aristotle and their relationship to the programming concepts of *type*, *variable*, *class*, and *object*.

### 2.3.2. Concepts and phenomena

A ***phenomenon*** is an object of the world of a domain as you perceive them. The following are phenomena:

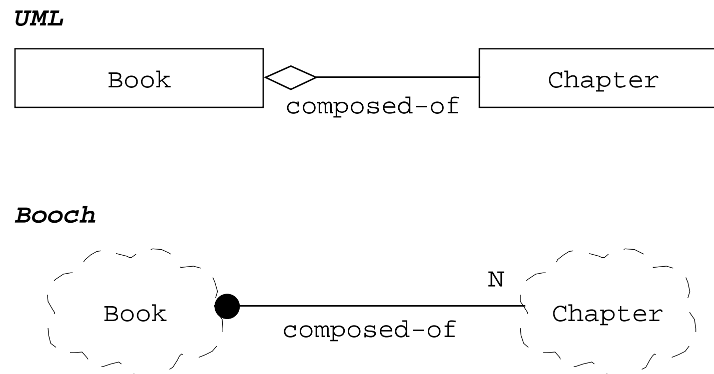


**FIGURE 6.** Example of describing the same model shown with two different notations. The model shows two classes, `Book` and `Chapter`, with the relationship, `Book is composed of Chapters`. In the UML notation, classes are represented by rectangles and aggregation associations by a line terminated with a diamond. In the Booch notation, classes are represented by clouds while aggregation associations are represented with a line terminated with a solid circle. The `N` above the aggregation association in the Booch diagram denotes that a `Book` can be composed of multiple `Chapters`.

**FIGURE 7.** Example of describing the same model shown with two different notations. This diagram represents the information of Figure 5: a `System` can be represented by many different `Models` that can be depicted by many different `Views`.

- the book you are holding
- the current savings interest rate is 3%
- my black watch

A ***concept*** is an abstraction describing a set of phenomena. The following are concepts:

- textbooks on software engineering
- savings interest rates
- black watches

A concept describes the properties of phenomena that are common. For example, the concept *black watches* is only concerned with the color of watches, not their origin or their quality. A concept is defined as a three tuple: its ***name*** (to distinguish it from other concepts), its ***purpose*** (the properties which determine if a phenomenon is part of the concept or not), and its ***members*** (the set of phenomena which are part of the concept).[1] Figure 8 illustrates the concept of clock. *Clock* is the name of the concept. *Devices that measures time* is the purpose of a clock. *My wrist watch* and *the wall clock about my desk* are members of this concept. Another example of concept is that of a club: A club has a name (e.g., "Valley Fisherman's club"), attributes that members must satisfy to be part of the club (e.g., "fisherman that live in Valley"), and actual members (e.g., "John Smith", "James Doe").

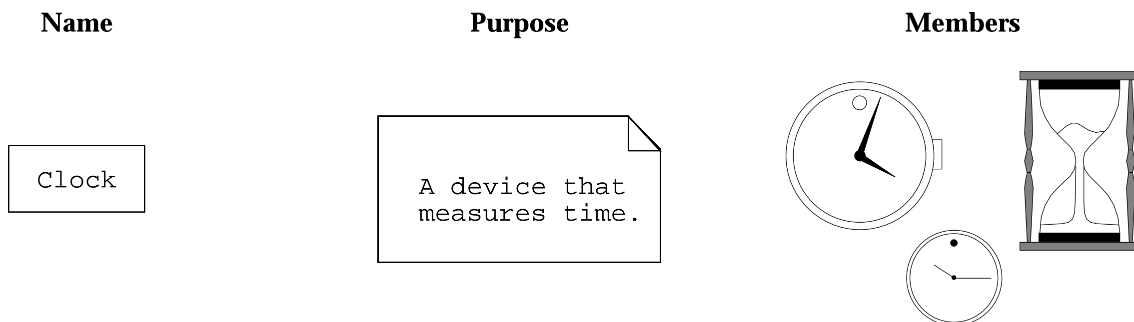


**FIGURE 8.** The three components of the `Clock` concept: name, purpose, and members

***Abstraction*** is the classification of phenomena into concepts. ***Modeling*** is the development of abstractions that can be used to answer specific questions about a set of phenomena. An abstraction is simpler to manipulate and examine than its corresponding set of phenomena because it contains less information: irrelevant details are abstracted away. In chemistry, the table of elements summarize the different types of atoms based on their atomic weight and number of electron pairs. Details such as the availability of each substance, its participation in different molecule are not represented. In biology, species are classified into family trees based on significant features (e.g., warm blooded, presence of vertebrae). A tree of species ignores issues related to behavior or habitat. In astronomy, stars are classified into different types based on their spectrum and dissipated energy. In this classification, the location of the stars, their detailed composition and dimensions are ignored.

In engineering, the model may exist prior to the phenomenon it models: a UML model might describe a system that has not been implemented yet. In sciences, the model may state

1. The three components of a concept are also sometimes referred as the name, the intension, and the extension.

the existence of systems and lead to the experiments that show their existence: the theory behind the top quark was developed before accelerator experiments in CERN were designed and executed, demonstrating the existence of the top quark.

In summary, modeling is the activity software engineers perform when they are designing a system. The purpose of modeling is to construct an abstraction of the system which ignores certain details. Software engineers abstract concepts from the application domain (i.e., the environment in which the system is operating) and from the solution domain (i.e., the technologies to build system). The resulting model is simpler than the environment or the system and thus is easier to manipulate. During the development of the model or its validation, software engineers need to communicate about the system with other engineers, clients, or users. They can represent the model in their imagination, on a napkin, in a CASE tool, or using different notations. In doing so, they construct views of the model for supporting their specific communication need.

### 2.3.3. 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, distinguishing it from other data types, it has a purpose, i.e., the structure and the operations valid on all members of the data type, and it has members, i.e., the members of the data type. Data types are used in typed languages to ensure that only valid operations are applied to specific data members.

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

An ***abstract data type*** is a special data type whose structure is hidden from the rest of the system. This allows the developer to revise the structure and the implementation of the abstract data type without impacting the rest of the system.

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***.

1. We refer to an operation b y its name followed by its arguments in parenthesis. If the arguments are not specified we just suffix the name of the operation by a pair of empty parenthesis.

An ***instance*** is any member of a specific data type. For example, `1291` is an instance of the type `int`. `3.14` is an instance of the type `float`. An instance is a member of a data type, and thus can be manipulated with the operations defined by the data type.

The relationship between data type and instance is similar to the relationship between concept and phenomenon: a data type is an abstraction that describes a set of instances which share common characteristics. For example, the operation for renaming an instance of `Person` need only be defined once in the `Person` data type but will be applicable to all possible instances of `Person`.

### 2.3.4. Classes, abstract classes, and objects

A ***class*** is an abstraction in the context of object-oriented programming languages. As in the case of an abstract data type, a class encapsulates both structure and behavior. Unlike abstract data types, classes can be defined in terms of other classes using inheritance. For example the class `CalculatorWatch` can be defined by refining the class `Watch` (see Figure 9). This type of relationship between a base class and a refined class is called generalization. The base class (e.g., `Watch`) is called the ***superclass***, the refined class is called the ***subclass*** (e.g., `CalculatorWatch`). In a generalization relationship, the subclass refines the superclass by defining new attributes and operations. In Figure 9, `CalculatorWatch` defines functionality for performing simple arithmetic that regular `Watches` do not have.

When a generalization serves only the purpose of modeling shared attributes and operations, that is, if the generalization is never instantiated, it is called an ***abstract*** class. Abstract classes often represent generalized concepts in the application domain. Whenever we classify phenomena into concepts, they often create generalizations to manage the complexity of the classification. For example, in chemistry, `Benzene` can be considered a class of molecules that belongs to the abstract class `Organic compound`. Note that Organic compound is a generalization and does not correspond to any one molecule, i.e., it does not have any direct instances. In modeling software systems, abstract classes sometimes do not correspond to an application domain concept, but rather, are introduced to reduce complexity in the model or to promote reuse.

A class defines the ***operations*** which 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, the operation `SetDate(d)`, setting the current date of a `Watch`, is also applicable for `CalculatorWatches`. The operation `EnterCalcMode()`, however, defined in the `CalculatorWatch` class is not applicable in the `Watch` class.

A class defines the ***attributes*** which all its instances contain. An attribute is a named slot in the instance where a value is stored. Attributes have a unique name within the class and a

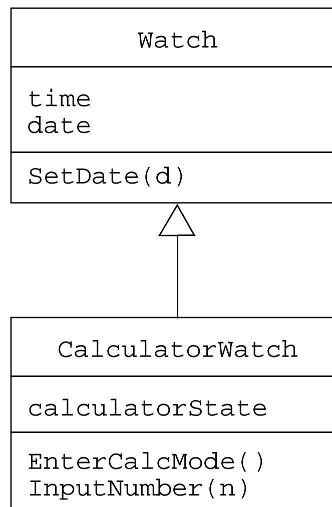


**FIGURE 9.** UML class diagram depicting two classes: `Watch` and `CalculatorWatch`. `CalculatorWatch` is a refinement of `Watch`, providing calculator functionality normally not found in normal watches. In UML an inheritance relationship is displayed by a line terminated with a triangle. The triangle points to the superclass while the other end is attached to the subclass.

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 belong 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 types.[1] In Figure 11, `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`. Unlike abstract data types, 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.

1. Underlined strings are also used for representing URLs (Uniform Resource Locators). Usually, the context in which the underlined string appears can be used to resolve this ambiguity.

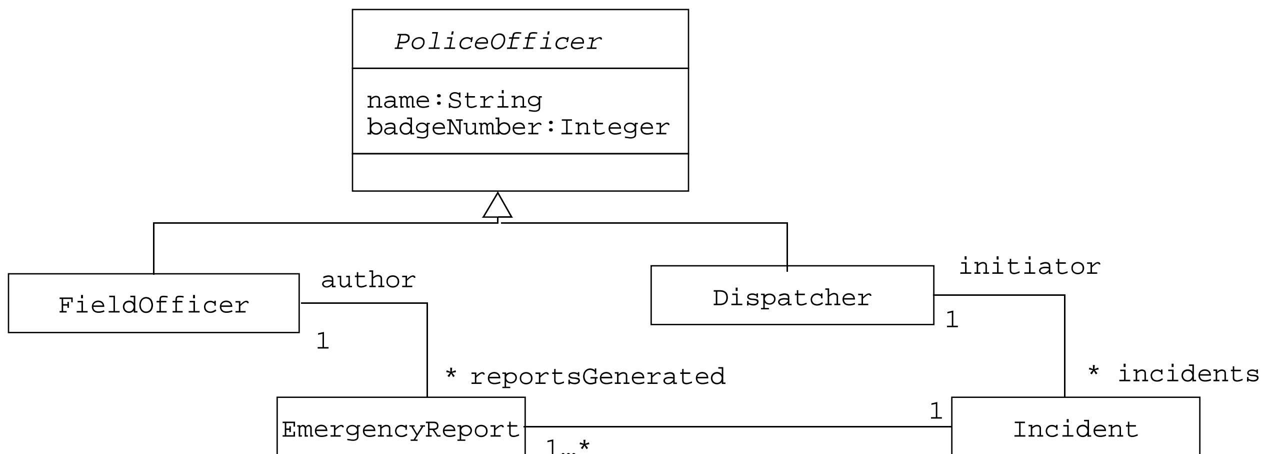


**FIGURE 10.** An example of a generalization (UML class diagram). `PoliceOfficer` is an abstract class which defines the common attributes and operations of the `FieldOfficer` and `Dispatcher` classes.

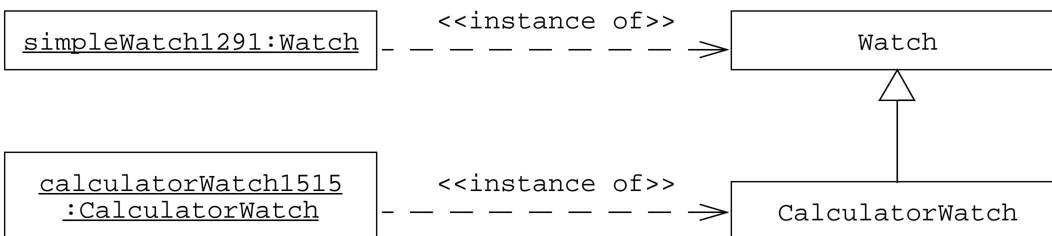


**FIGURE 11.** UML class diagram depicting instances of two classes. `simpleWatch1291` is an instance of `Watch`. `calculatorWatch1515` is an instance of `CalculatorWatch`. Although the operations of `Watch` are also applicable to `calculatorWatch1515`, the latter is not an instance of the former.

### 2.3.5. Event classes, events and messages

***Event classes*** are abstractions representing a kind of events 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 `WatchOwner` 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 operation and passes the arguments to the operation. Any results are returned to the sending object.

For example, in Figure 12, 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.

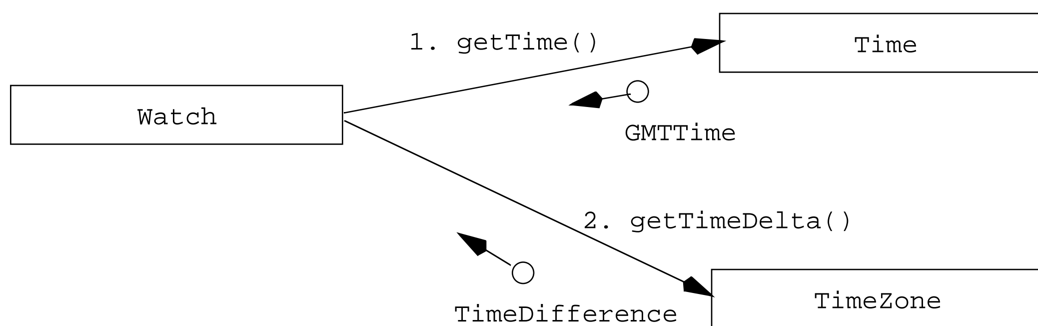


**FIGURE 12.** Examples of message sends (UML Collaboration diagram): the `Watch` object sends the `getTime()` message to the Time object to query the current Greenwich time. It then sends the `getTimeDelta()` message to the `TimeZone` objects to query the difference to add to the Greenwich time. The circles represents the results that are sent back to the message sender.

Events and messages are instances: 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. Usually, the context in which the term is used is sufficient to clarify this ambiguity.

### 2.3.6. Object-oriented modeling

The ***problem domain*** is the set of all environments in which the system presents a solution. 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 problem domain for a system to accomplish its intended task effectively. Note that the problem domain changes over time, as work processes and people change. The problem domain is also called the application domain.[1]

The ***solution domain*** is the space of all possible systems. The solution domain is much richer and more volatile than the problem domain. This can be due to emerging technologies (also called technology enablers), to changes as the implementation technology matures, or better understanding of implementation technology by the developers when they build the system. Modeling the solution domain represents the system and object design tasks of the development process. Note that the deployment of the system can change the problem domain. The solution domain is also called the implementation domain.

***Object-oriented analysis*** is concerned with modeling the problem domain with objects. ***Object-oriented design*** is concerned with modeling the solution domain. Both domains are modeled using the same representations (i.e., classes and objects). Furthermore, objects in the problem domain are included in the solution domain, that is, there are objects in the system which model the phenomena that the system manipulates. For example, an air traffic control system has a `TrafficController` class to represent individual users, their preferences and log information. The system also has a `Aircraft` class to represent information associated with the tracked aircraft. Traffic controller and aircraft are problem domain concepts which are encoded into the system (see Figure 13).

Modeling the problem domain and the solution domain with a single notation has pros and cons. On the one hand, it can be powerful: solution domain classes which represent application concepts can be traced right back to the problem domain. Moreover, these classes can be encapsulated into subsystems independent of other implementation concepts (e.g., user interface and database technology) and be made 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 system domain is bound to be simpler and biased towards the solution. To address this issue, we will use a single notation and, in cases of ambiguity, we will distinguish between the two domain. In most cases, you should assume we are referring to the model (e.g., “an `Aircraft` is composed of `Manifest` and a `FlightPlan`” is a statement about the model).

### 2.3.7. 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, that the model does not correspond to the reality it is supposed to represent.

1. The problem domain is sometimes further divided into a user domain and a client domain. The client domain includes the issues relevant to the client, e.g., 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, e.g., functionality, ease of learning and of use.

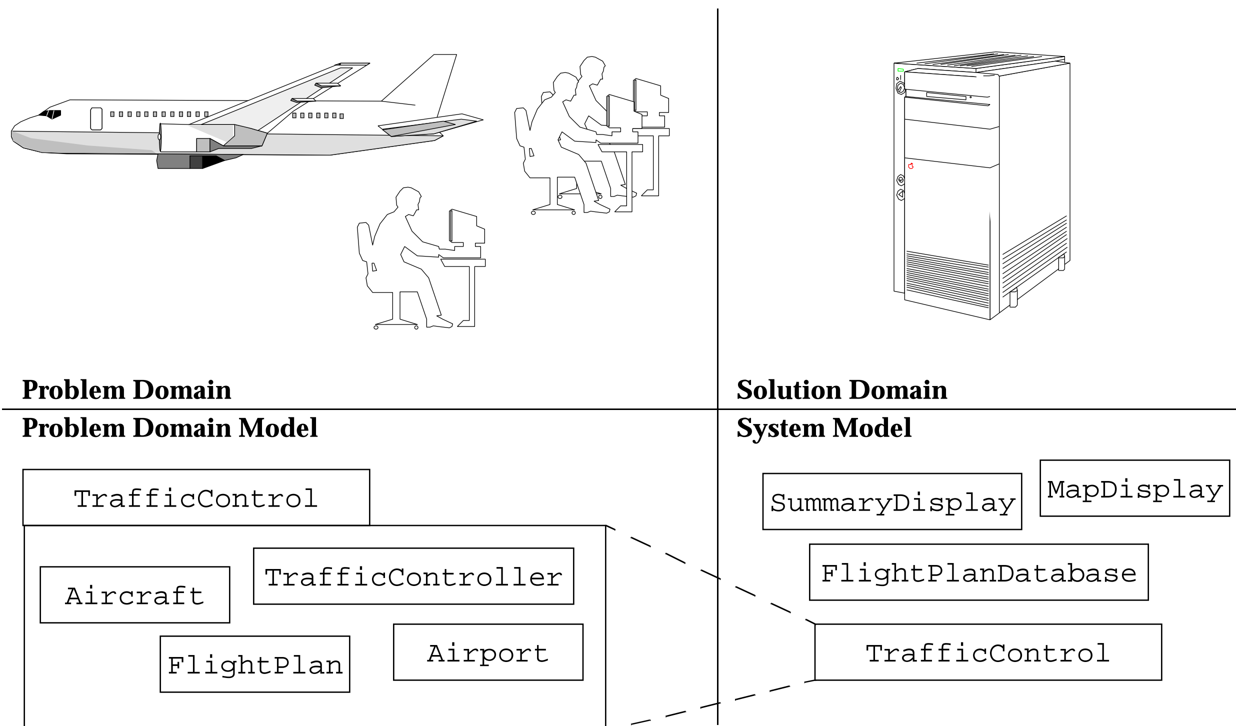


**FIGURE 13.** The problem domain model represents entities of the environment which are relevant to an air traffic control system (e.g., aircraft, traffic controllers). The system model represents entities that are part of the system (e.g., map display, flight database). In object-oriented analysis and design, the problem domain model is also part of the system model. An example in this figure is the `TrafficControl` package that appears in both models. (For more details see Chapter 7, *Requirements Analysis*).

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, but which are rejected once a counter example is found. Ptolomeus's earth-centric model of the universe was (eventually) falsified in favor of the Copernican solar-centric model once data from Gallileo was accepted. The Copernican solar-centric model was then falsified once other galaxies were discovered and the concept of galaxy had to be added to the model.

We can apply falsification to software system development as well. For example, a technique for developing a system is ***prototyping***: developers construct a prototype which only simulates the user interface of a system. The prototype is then presented to potential

users for evaluation, i.e., falsification, and modified subsequently. In the first iterations of this process, developers are likely to throw away the initial prototype due to feedback from the users. In other terms, users falsify the initial prototype, a model of the future system, because it does not represent accurately relevant details.

Note that it is only possible to demonstrate that a model is incorrect. Although 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 need of the client. However, at any time, we have to be prepared that the system can be falsified, due to changing requirements or changes in the environment.

## 2.4. A deeper view into UML

We now describe in more detail the four main UML diagrams we use in this book. Use case diagrams (Section 2.4.1) represent the system from a user's point of view. They define the boundaries of the system. Sequence diagrams (Section 2.4.2) 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. Class diagrams (Section 2.4.3) are used to represent the structure of a system in terms of objects, their attributes and relationships. Statechart diagrams (Section 2.4.4) are used to represent the behavior of non-trivial objects.

### 2.4.1. Use case diagrams

#### *Use cases and actors*

***Actors*** are external entities that interact with the system. Examples of external entities are a role a user plays (e.g., a system administrator, a bank customer, a bank teller) or another system (e.g., a legacy system, a central database, a fabrication line). Actors have a unique name and a textual description.

***Use cases*** describe the behavior of the system, as seen from an actor's point of view. Behavior described by the use case model is also called ***external behavior***. A use case describes a function provided by the system as a set of events that yield a visible result for

the actors. Actors initiate and communicate with use cases. Use cases can communicate with actors and with other use cases.

**UML definitions related to use case diagrams:**

- ***Actor*** - A coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates.
- ***Use case*** - The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system.
- ***Scenario*** - A specific sequence of actions that illustrates behaviors. A scenario may be used to illustrate an interaction.

For example, in an accident management system [FRIEND, 1994], a field officer (e.g., a police officer or a fireman) might have access to a wireless computer which enables her to interact with a dispatcher. The dispatcher in turn can visualize the current status of all its resources (e.g., police cars or trucks) on a computer screen and dispatch a resource by issuing commands from her workstation. In this example, the field officer and the dispatcher are actors.

Figure 14 depicts the actor `FieldOfficer` who invokes the use case `ReportEmergency` to notify the actor `Dispatcher` of a new emergency. As a response, the actor `Dispatcher` in turn invokes the `OpenIncident` use case to create an incident report and initiate the incident handling. Preliminary information from the `FieldOfficer` is entered in the incident database and additional units are dispatched by the `Dispatcher` actor to the scene with the `AllocateResources` use case.

To describe a use case, we use a template composed of six fields (see Figure 15):

- ***Name:*** The name of the use case is unique across the system such that developers (and project participants) can unambiguously refer to it.
- ***Participating actors*** are actors initiating the use case or receiving information from it.
- ***Entry conditions*** describe the conditions under which the use case begins.
- ***Flow of events*** describe the sequence of actions of the use case, which may be numbered for reference. The common case and the exceptional cases are described separately in different use cases for clarity.
- ***Exit conditions*** describe the conditions under which the use case ends.
- ***Special requirements*** are requirements that are not related to the functionality of the system (i.e., what the system does). These may be constraints on the performance of the system, its implementation, the hardware platforms it runs on, etc. Special requirements are described in more detail in Chapter 6, *Requirements Elicitation*

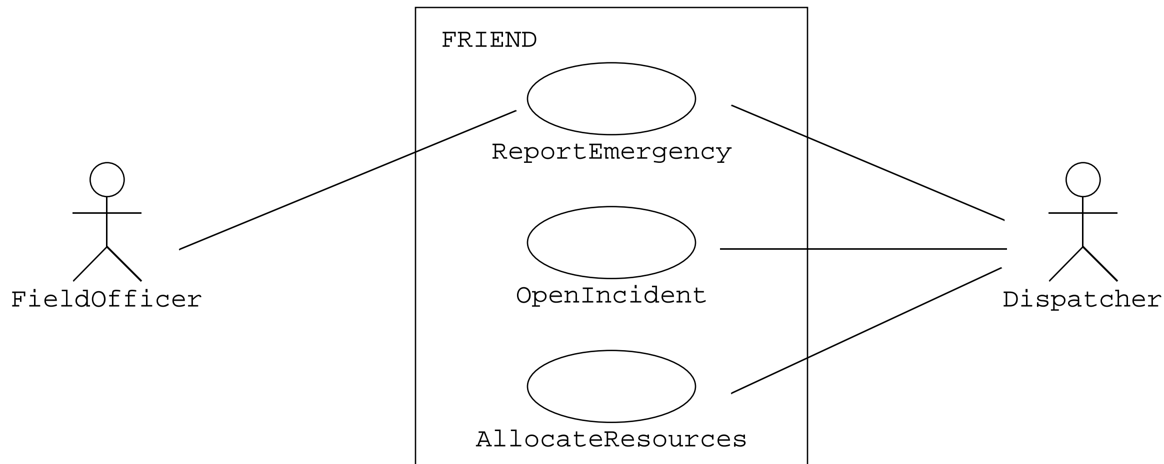


**FIGURE 14.** An example of a use case model: incident initiation in an accident management system. Associations between actors and use cases represent information flows. In UML these associations are bidirectional: they can represent the actor initiating a use case (e.g., `FieldOfficer` initiates `ReportEmergency`) or a use case providing information to an actor (e.g., `ReportEmergency` notifies `Dispatcher`)

| | |
|---|---|
| *Use case name* | `ReportEmergency` |
| *Participating actor* | invoked by `FieldOfficer`<br>communicates with `Dispatcher` |
| *Entry condition* | 1. The `FieldOfficer` activates the “Report Emergency” function of her terminal. FRIEND responds by presenting a form to the officer. |
| *Flow of events* | 2. The `FieldOfficer` fills 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, at which point, the `Dispatcher` is notified.<br>3. 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 emergency report. |

**FIGURE 15.** An example of a use case: the `ReportEmergency` use case

| | |
|---|---|
| *Exit condition* | 4. The FieldOfficer receives the acknowledgment and the selected response. |
| *Special 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 15.** An example of a use case: the ReportEmergency use case

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 a extensive knowledge of software engineering notations. The use of natural language also enables persons from other disciplines to understand the requirements of the system. Developers formalize the use case model into object models once the requirements become stable.

### *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 specific set of actions. Scenarios are used as examples for illustrating typical 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. Finally, the ***flow of events*** of a scenario describes step by step the sequence of events. 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 flow of events, such conditions are unnecessary.

Use case diagrams can include three types of relationships: communication, use, and extension. We describe these relationships in more detail next.

### *Communication relationships*

Actors and use cases are said to ***communicate*** when information is exchanged between them. Communication relationships are depicted by a solid path between the actor and use case symbol. In Figure 14, the actors FieldOfficer and Dispatcher communicate with the

| | |
|---|---|
| *Scenario name* | warehouseOnFire |
| *Participating actor instances* | bob, alice: FieldOfficer<br>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.<br>2. Alice enters the address of the building, a brief description of its location (i.e., north west corner), and an emergency level. In addition to a fire unit, he requests several paramedic units on the scene given that area appear to be relatively busy. He confirms his input and waits for an acknowledgment.<br>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 creates allocates a fire unit and two paramedic units to the Incident site and sends their estimated arrival time (ETA) to Alice.<br>4. Alice receives the acknowledgment and the ETA. |

**FIGURE 16.** warehouseOnFire scenario for the ReportEmergency use case.

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.

### *Uses relationships*

When describing a complex system, its use case model can become quite complex and contain redundancy. We can 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 Help. This can be modeled by a use case HelpDispatcher that is used by the use cases OpenIncident and AllocateResources (and any other use cases accessible by the Dispatcher). The resulting model only describes the HelpDispatcher functionality once, thus reducing complexity. Two use cases are related by a uses relationship if one of them includes the second one in its flow of events. In UML, ***uses*** relationships are depicted by a hollow arrow originating from the use case doing the use to the use case being used (see Figure 17). Uses relationships are labeled with the string <<uses>>.

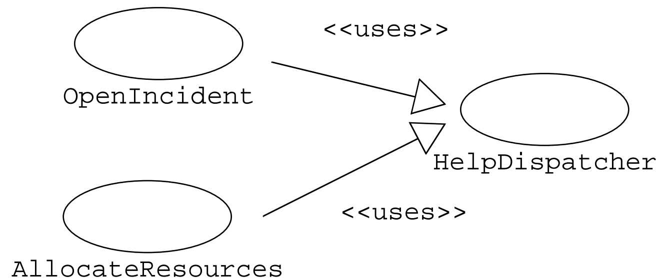


**FIGURE 17.** An example of a `uses` relationship (UML use case diagram).

### *Extends relationships*

***Extends*** relationships are an alternate means to reduce complexity in the use case model. A use case can extend another use cases by adding events. An extends relationship indicates that an instance of an extended use case may include (under certain conditions) the behavior specified by the extending use case. Typical applications of extends relationship include the specification of exceptional behavior. For example (see Figure 18), assume that the connection between the `Dispatcher` and the `FieldOfficer` can be lost at any time. This can happen 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. The use case `ConnectionDown` extends the use cases `OpenIncident` and `AllocateResources`. Separating exceptional behavior from common behavior enables us to write shorter and more focused use cases.

The difference between the uses and extends relationships is the location of the dependency. Assume that we add several new use cases for the actor `Dispatcher`. If we modeled the `HelpDispatcher` function with uses relationships, every new use case will need to use the `HelpDispatcher` use case. If we used extends relationships instead, the `HelpDispatcher` 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 extends relationships. Use cases that describe behavior commonly shared by a fixed set of use cases are modeled with uses relationships.

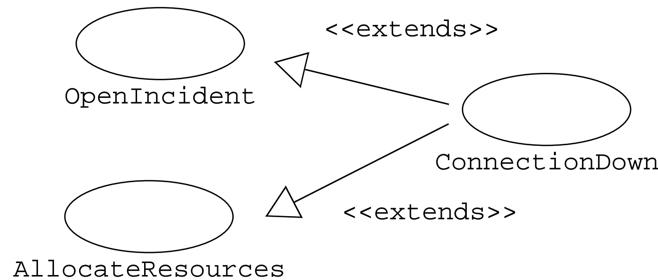


**FIGURE 18.** An example of an `<<extends>>` relationship (UML use case diagram).

***Applying use case diagrams***

Use case models define the boundaries of the system. They are developed during requirements engineering, often with the client and the users. During the next phase of the project, called requirements analysis, they are refined and corrected, as they are reviewed by a broader audience that includes developers and validated against real situations. During requirements analysis, sequence diagrams are derived from use cases. This allows the behavior of the system to be described in more detail and to identify participating objects. The next section describes sequence diagrams.

### 2.4.2. Sequence diagrams

***Sequence diagrams*** describe any pattern of communications 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 an operation 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 operation in the receiving object.

**UML definitions related to sequence diagrams:**

- ***Sequence diagram*** - A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. … A sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario). …
- ***Event*** - The specification of a significant occurrence that has a location in time and space. …
- ***Message*** - A specification of a communication between objects that conveys information with the expectation that activity will ensue. The receipt of a message is normally considered an instance of an event.
- ***Argument*** - A specific value corresponding to a parameter.

For example, let us consider the case of a digital watch with two buttons (hereafter `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, the minutes, or the seconds, day, month, year). Initially, when the `2BWatchOwner` enters the set time mode, the hours are blinking. If the actor presses the first button, the next number will blink (e.g, if the hours are blinking and the actor presses the first button, the hours will stop blinking and the minutes will start blinking. If the actor presses the second button, the blinking number will be 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 will be set to 0 if the actor presses the second button). The actor exits the set time mode by pressing both buttons simultaneously. Figure 19 depicts a sequence diagram for the case of an actor setting his `2Bwatch` one minute ahead.

Each column represents an object that is participating in the interaction. The vertical axis represents time (from top to bottom). Messages are shown by arrows. Labels on arrows represent message names and may contain arguments. Activations (i.e., executing methods) are depicted by vertical rectangles. Actors are shown as the left most column.

Sequence diagrams can be used to describe use cases (i.e., all possible interactions) and scenarios (i.e., one possible interaction, as in Figure 19). Usually, sequence diagrams are drawn for a prototypical case to discover new objects, operations, or attributes. When describing all possible interactions, sequence diagrams also provide notations for conditionals and iterators. A condition on a message is denoted by an expression in brackets before the message name (see `op1` and `op2` in Figure 20). If the expression is true, the

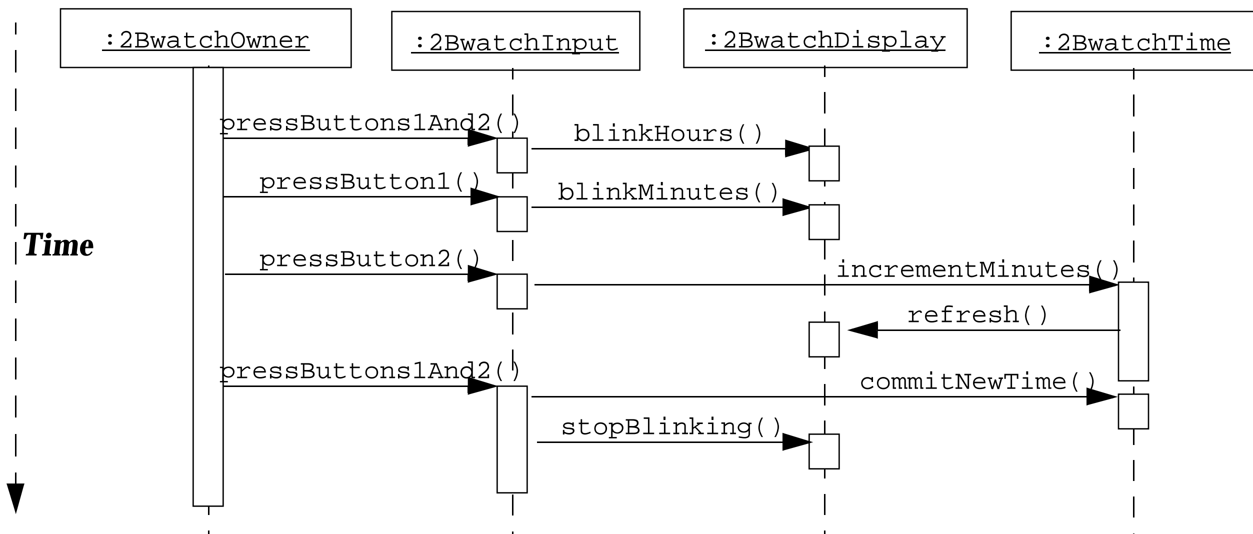


**FIGURE 19.** Example of a sequence diagram: setting the time on `2Bwatch`.

message is sent. Repetitive invocation of a message is denoted by a * before the message name (see `op3` in Figure 20)

### *Applying sequence diagrams*

Sequence diagrams describe interactions among several objects. Typically, we use a sequence diagram to describe the event flow of a use case, identify the objects which participate in the use case, and assign pieces of the use case behavior to them in the form of services. This process often 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. Once we have found a sufficient number of objects, we use class diagrams to describe their relationships and their attributes.

## 2.4.3. 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 structure and behavior of a set of objects. ***Objects*** are

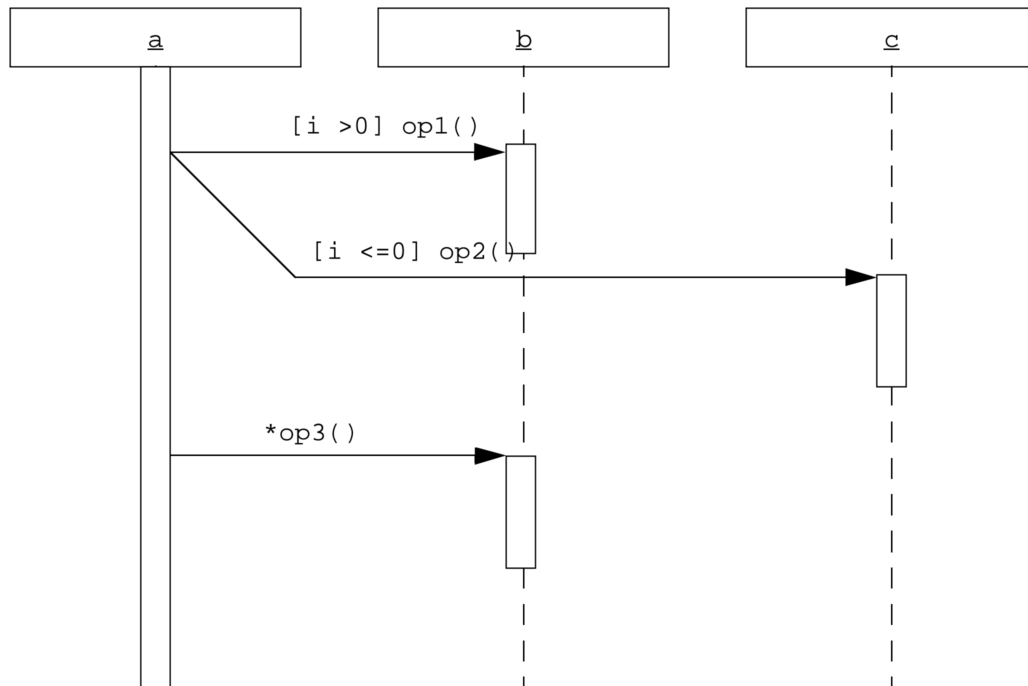


**FIGURE 20.** Examples of conditions and iterators in sequence diagrams.

entities that encapsulate state and behavior. Each object has an identity: it can be referred individually and is distinguishable from other objects.

**UML definitions related to class diagrams:**

- ***Class*** - A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment.
- ***Object*** - An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations and methods. An object is an instance of a class.
- ***Attribute*** - A named slot in a classifer that describes a range of values that instances of the classifer may hold.
- ***Relationship*** - A semantic connection among model elements. Examples of relationships include associations and generalizations.
- ***Association*** - The semantic relationship between two or more classes that involves connections among their instances.

- ***Link*** - A semantic connection among a tuple of objects. An instance of an association.
- ***Operation*** - A service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible.
- ***Method*** - The implementation of an operation. It specifies the algorithm or procedure that effects the results of an operation.

In UML, classes and objects are depicted by rectangles. The name of the class or object is displayed at the top of the rectangle. Object names are underlined to indicate that they are instances. By convention, class names start with an upper case letter. Objects in object diagrams may be given names (followed by their class) for ease of reference. In that case, their name starts with a lower case letter. In the accident management example (see Figure 21 and Figure 22), Bob and Alice are field officers and they are represented in the system as `FieldOfficer` objects called `bob` and `alice`. `FieldOfficer` is the class describing all field officer objects whereas Bob and Alice are represented by two individual `FieldOfficer` objects.

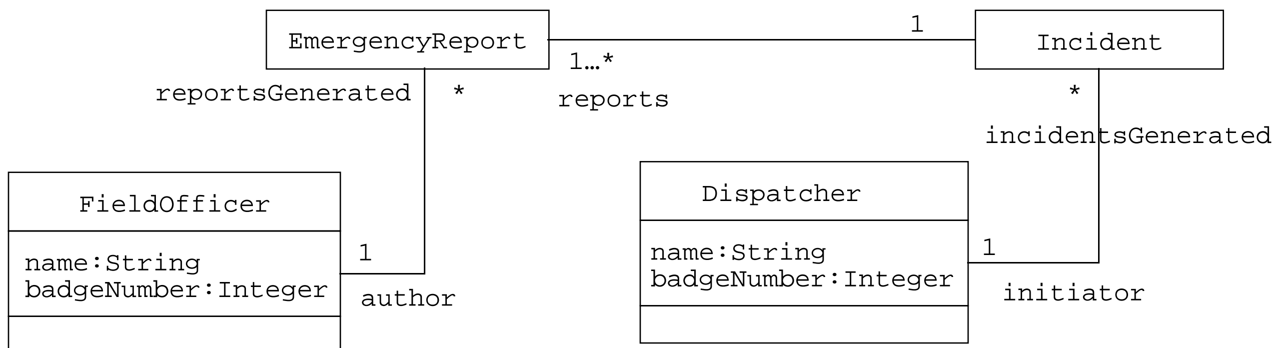


**FIGURE 21.** An example of a UML class diagram: objects participating in the `ReportEmergency` use case.

In Figure 21 the `FieldOfficer` class has two attributes: `name` and a `badgeNumber`. This indicates that all `FieldOfficer` objects have these two attributes. In Figure 22, the `bob` and `alice` objects have specific values for these attributes: “Bob. D.” and “Alice W.”, respectively. In Figure 21, the `FieldOfficer` `name` attribute is of type `String`, which indicates that only instances of `String` can be assigned to the `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 well into 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

***Associations*** are relationships between classes and represent groups of links. A ***link*** represents a connection between two objects.Links are instances of associations. Each `FieldOfficer` object also has a list of `EmergencyReports` that have been written by the `FieldOfficer`. In Figure 21, the line between the `FieldOfficer` class and the `EmergencyReport` class is an association. In Figure 22, the line between the `alice` object and the `report_1291` object is a link. This link represents state that is kept in the system to denote that `alice` generated `report_1291`.

### Roles

The end of an association can be labeled by a string called ***role.*** In Figure 21, the roles of the association between the `EmergencyReport` and `FieldOfficer` classes are `author` and `reportsGenerated`. Labeling the end of associations with roles allows us to distinguish multiple associations originating from a class. Moreover, roles clarify the purpose of the association.

### Multiplicity

The end of an association can be labeled by a set of integers indicating how many links can legitimately originate from an instance of the class connected to the association end. 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 start. The “many” multiplicity is a short hand standing for `0…n`. This means that any given `FieldOfficer` may be the author of zero or more `EmergencyReports`.

### Qualified associations

***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

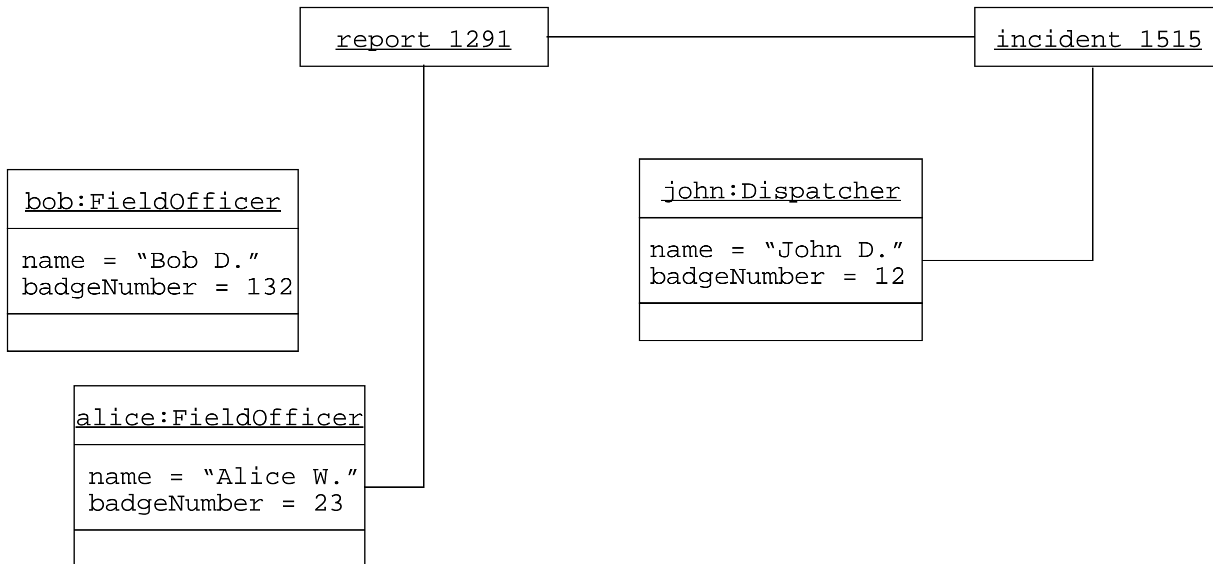


**FIGURE 22.** An example of a UML object diagram: objects participating in the `warehouseOnFire` scenario.

qualification (see top of Figure 23), 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 23). The relationship between `Directory` and `File` is called a ***qualified association***.

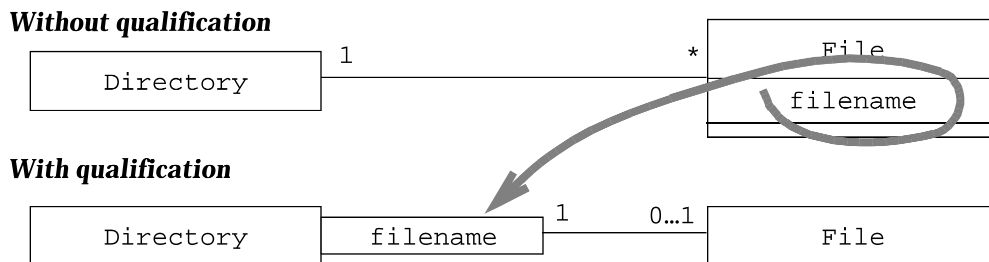


**FIGURE 23.** Example of how a qualified association reduces multiplicity (UML class diagram). Adding a qualifier clarifies the class diagram and increase the conveyed information. In this case, the model including the qualification denotes that the name of a file is unique within a directory.

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 and check if a qualifier can be added. Often, these associations can be qualified with an attribute of the target class, (e.g., the `filename` attribute in Figure 23).

### *Association class*

Similar to classes, associations can have attributes and operations attached to them. Such an association is called an ***association class*** and is depicted by a class symbol, containing the attributes and operations, connected to the association symbol with a dashed line. For example, in Figure 24, the allocation of `FieldOfficers` to an `Incident` is modeled as an association with attributes `role` and `notificationTime`.

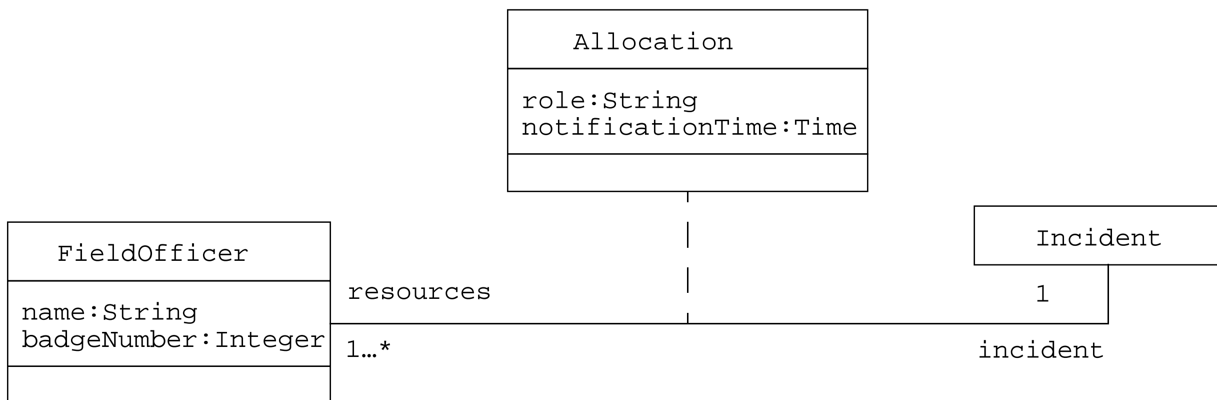


**FIGURE 24.** An example of an association class (UML class diagram).

Any association class can be transformed into a class and simple associations as shown in Figure 25. Although both representations are similar, the association class representation is clearer: an association cannot exist without the classes it links, similarly the `Allocation` object cannot exist without a `FieldOfficer` and an `Incident` object. Although Figure 25 carries the same information, this diagram requires careful examination of the multiplicity of several roles. Such modeling trade-offs will be examined in more detail in Chapter 7, *Requirements Analysis*.

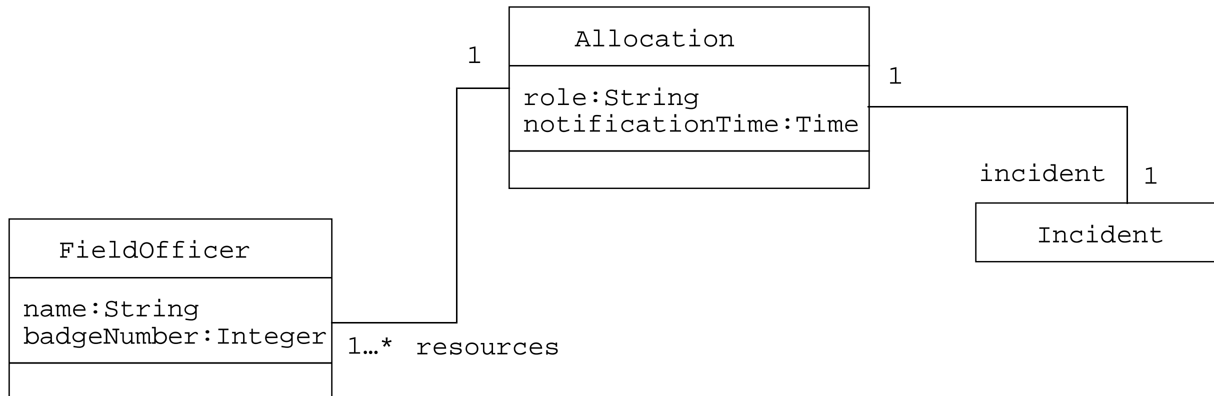


**FIGURE 25.** Alternative model for `Allocation` (UML class diagram).

### *Aggregation*

Associations can be used to represent a wide range of connections among a set of objects. In practice, a special case of association occurs frequently: composition. Composition is a hierarchical relationship. For example, a `State` is composed of `Counties` which in turn are composed of `Townships`. A `PoliceStation` is composed of `PoliceOfficers`. Another example is a `Directory` that contains a number of `Files`. Such relationships could be modeled using a one to many association. Instead, UML provides the concept of an ***aggregation*** to denote composition. An aggregation is denoted by a simple line with a diamond at the container end of the association (see Figure 26). Although one to many associations and aggregations can be used alternatively, aggregations are preferable given that they emphasize the hierarchical aspects of the model: The `PoliceOfficers` are part of the `PoliceStation`.

### *Generalization*

***Generalization*** is the relationship between a general class and one or more specialized classes. Generalization 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` while `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 27).

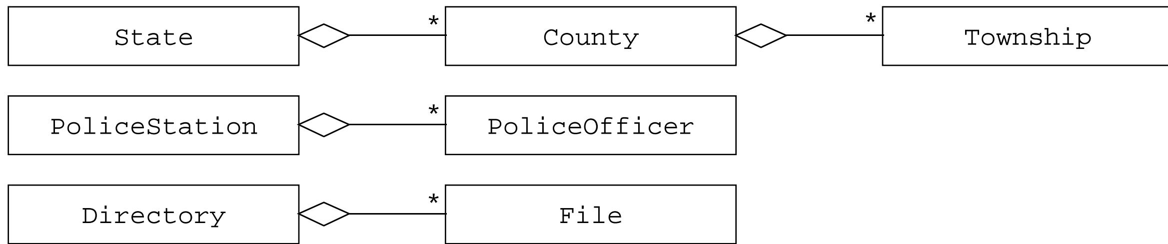


**FIGURE 26.** Examples of aggregations (UML class diagram). A `State` contains many `Counties` which in turn contains many `Townships`. A `PoliceStation` has many `PoliceOfficers`. A file system `Directory` contains many `Files`.

PoliceOfficer, the generalization, is called a ***superclass.*** `FieldOfficer` and `Dispatcher`, the specializations, are called the ***subclasses***. The subclasses ***inherit*** the attributes and operations of their subclass. Abstract classes (defined in Section 2.3.4) are distinguished from concrete classes by *italicizing* their name. In Figure 27, *`PoliceOfficer`* is such an abstract class. Abstract classes are used in object-oriented modeling to classify related concepts, and thus, to reduce the overall complexity of the model.

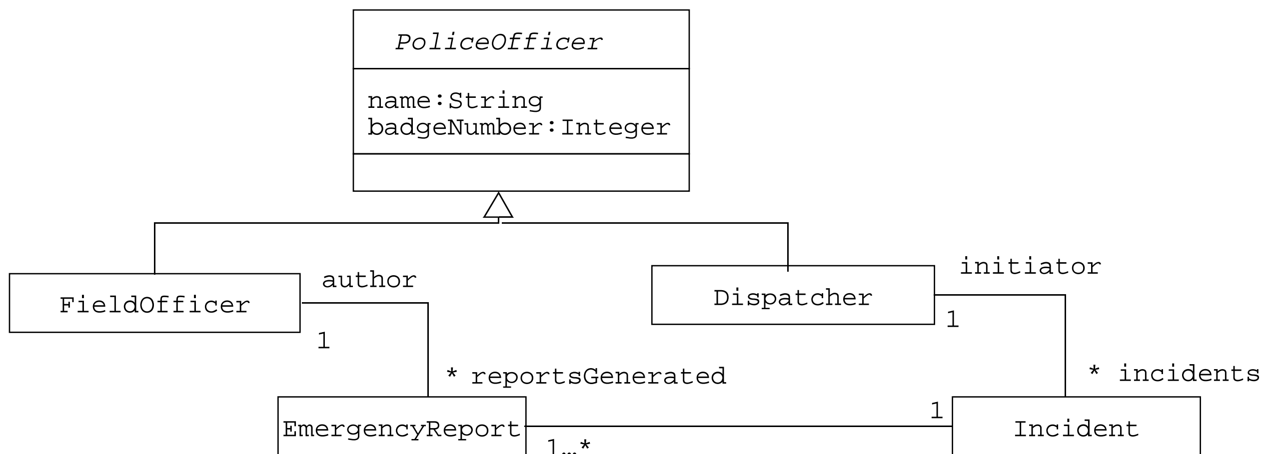


**FIGURE 27.** An example of a generalization (UML class diagram). `PoliceOfficer` is an abstract class which defines the common attributes and operations of the `FieldOfficer` and `Dispatcher` classes.

Object behavior is specified by ***operations***. A set of operations represents a ***service*** offered by a particular class. 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 operations of a class is the list of public services that the class offers. The methods of its class are the implementations of these operations.

The distinction between operations and methods allows for a cleaner separation between the mechanism to request a service and the location it is provided. For example, the class `Incident` in Figure 28 defines an `assignResource()` operation 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 the Incident class is responsible for closing the incident. This includes going over all the resource which have been assigned to the incident over time and collecting their reports.

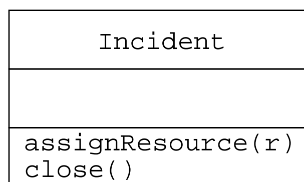


**FIGURE 28.** Examples of operations provided by the `Incident` class.

### *Applying class diagrams*

Class diagrams are used for describing the structure of a system. During requirement analysis, software engineers build class diagrams to formalize application domain knowledge. Classes represent participating objects found in use cases and sequence diagrams and describe their attributes, and operations. The purpose of requirement analysis models is to describe the scope of the system and discover its boundaries. For example, using the class diagram pictured in Figure 21, an analyst could examine the multiplicity of the association between `FieldOfficer` and `EmergencyReport` (i.e., one `FieldOfficer` can write zero or more `EmergencyReports`, each `EmergencyReport` is written by exactly one `FieldOfficer`) and ask the user whether this is correct. Can there be more than one author for 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 requirements analysis models is described in Chapter 7, *Requirements Analysis*.

Requirement analysis models do not focus on implementation. Concepts such as interface details, network communication, and database storage are not shown. 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 8, *System Design*.

### 2.4.4. Statechart diagrams

A ***UML statechart*** is a notation provided by UML to describe the sequence of states an object goes through in response to external events. Statecharts are extensions of the traditional finite state machines model. On the one hand, statecharts provide notations for nesting states and state machines (i.e., a state can be described by a state machine). On the other hand, statecharts provide notations for binding transitions with message sends and conditions on objects. UML statecharts were inspired by Harel's statecharts [Harel, 1987]. A UML statechart is equivalent to a traditional Mealy or Moore state machine.

A ***state*** is a condition that an object satisfies. A state can be thought as an aggregation of attribute values that has some significance. For example, the `Incident` class in FRIEND can have four states: active, inactive, closed, and archived (see Figure 29). An active `Incident` denotes a situation which requires a response (e.g., an ongoing fire, a traffic accident). An inactive `Incident` denotes a situation that was handled but for which reports need to be written (e.g., the fire has been put off but damage estimates have not yet been performed). A closed `Incident` denotes a situation which has been handled and documented. An archived `Incident` is a closed `Incident` whose documentation has been moved to off-site storage.

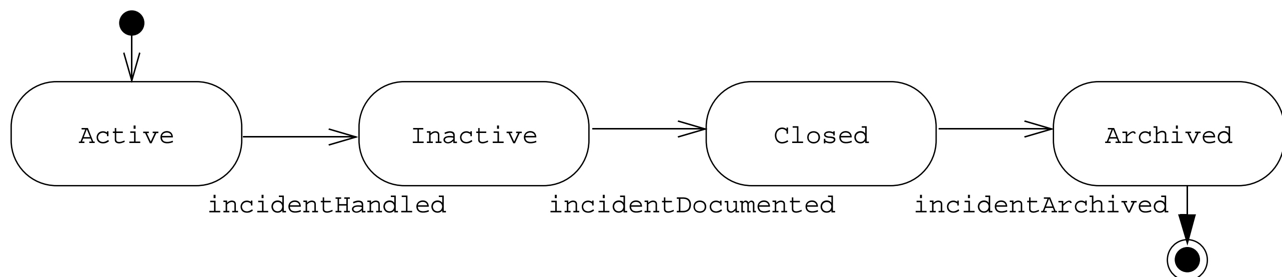


**FIGURE 29.** UML statechart diagram for the `Incident` class

A ***transition*** models changes of state triggered by events, conditions, or time. For example in Figure 29, there are three transitions: the transition 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 represented by a rounded rectangle. A transition is represented by arrows relating two states. States are labeled with their name and can be expanded. A small solid black circle indicates the initial state. A circle surrounding a small solid black circle indicates a final state.

**UML definitions related to statechart diagrams:**

- ***Statechart***- a diagram that shows a state machine.
- ***State machine*** - A behavior that specifies the sequences of states that an object goes through during its life in response to events, together with its responses and actions.
- ***State*** - A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- ***Transition*** - A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state the transition is said to fire.
- ***Action*** - The specification of an executable statement that forms an abstraction of a computational procedure. An action results in a change in the state of the model, and is realized by sending a message to an object or modifying a value of an attribute.
- ***Action state*** - A state that represents the execution of an atomic action, typically an operation.
- ***Activity diagram*** - A special case of a state diagram in which all, or most of the states, are action states and in which all, or most of the transitions are triggered by completion of actions in the source states.

Figure 30 displays another example, a statechart for the 2Bwatch (for which we constructed a sequence diagram in Figure 19). 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. When 2Bwatch is first powered, it is in the `SetTime` state. This is indicated by the small solid black circle which represents 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., `pressButtonsLAndR`) or by the passage of time (e.g., `after 2 min`.) Actions can be associated with a transition (e.g., beep when the transition between `SetTime` and `MeasureTime` is fired on the `pressButtonsLAndR` event)

The statechart diagram in Figure 30 does not represent the details of measuring or setting the time. These details have been abstracted away from the statechart diagram and can be modeled separately using either internal transitions or a nested statechart. Internal transitions (Figure 31) are transitions that remain within a single state. They can also have

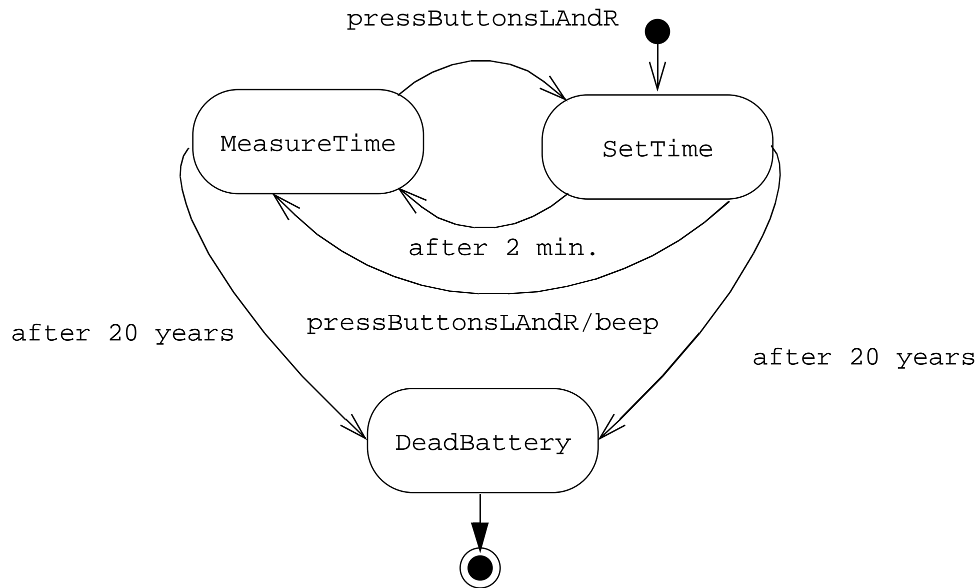


**FIGURE 30.** Statechart diagram for `2Bwatch` set time function.

actions associated with them. Entry and exit are displayed as an internal transition given that their actions do not depend on the originating and destination states.

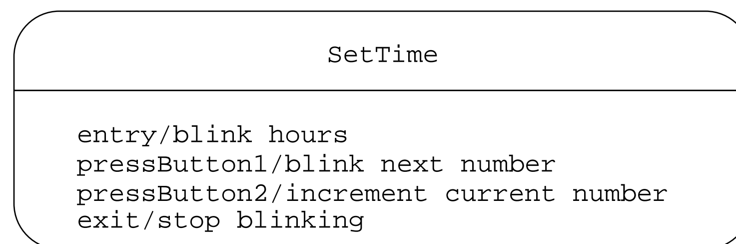


**FIGURE 31.** Internal transitions associated with the `SetTime` state.

Nested statecharts (Figure 32) reduce complexity. They can be used instead of internal transitions. In Figure 32, the current number is modeled as nested state, while actions corresponding to modifying the current number are still modeled using internal transitions. Note that each state could be modeled as a nested statechart (e.g., the `BlinkHour` statechart

would have twenty four sub states which correspond to the hours in the day, transitions between these states would correspond to pressing the second button).

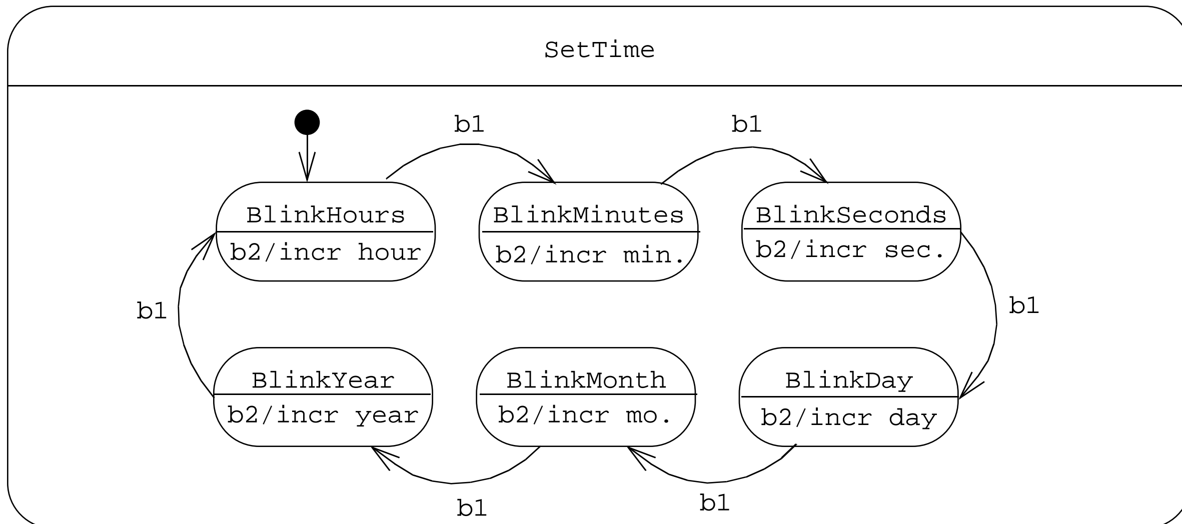


**FIGURE 32.** Refined statechart associated with the `SetTime` state.

### *Applying statechart diagrams*

Statechart diagrams are used to represent non trivial behavior of a subsystem or an object. Unlike sequence diagrams, they make explicit which attribute or set of attributes have an impact on the behavior of the object. Statecharts are used to identify object attributes and to refine the behavior description of an object, while sequence diagrams are used to identify participating objects and the services they provide. Statechart diagrams can also be used during system and detail design to describe system domain objects with interesting behavior. We will describe the use of statechart diagrams in more detail in Chapter 7, *Requirements Analysis* and Chapter 8, *System Design*.

### *Activity diagrams*

The outgoing transitions are triggered by the completion of an action associated with the state. This is called an ***action state***. By convention, the name of a state denotes a condition whereas the name of an action state denotes an action. ***Activity diagrams*** are state diagrams whose states are actions states. Figure 33 is an activity diagram corresponding to the state diagram in Figure 29. An alternate (and equivalent) view of activity diagrams is to interpret

action states as activities and transitions as a control flow, that is, the arrows are interpreted as sequential constraints between activities.

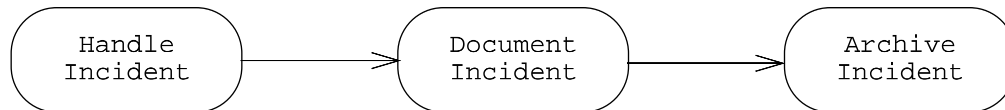


**FIGURE 33.** UML activity diagram for `Incident`. During the action state `Handle Incident` the `Dispatcher` receives reports and allocates resources. Once the `Incident` is closed, the `Incident` moves to the `Document Incident` action state in which all participating `FieldOfficers` and `Dispatchers` document the `Incident`. Finally, the `Archive Incident` action state represents the archival of the `Incident` related information onto slower access medium. Note that this activity diagram is more compact than its equivalent statechart (Figure 29).

***Decisions*** are branches in the control flow. They denote alternative transitions based on a condition in the system or the object. Decisions are depicted by a diamond with one or more incoming arrows and two or more outgoing arrows. The outgoing arrows are labeled with the conditions that select a branch in the flow of control. All outgoing transitions from a decision represent all possible outcomes. In Figure 34, 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 `ChiefOfPolice` 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.

***Complex transitions*** are transitions with multiple source states or multiple target states. Complex transitions denote the synchronization of multiple activities (in the case of multiple sources) or the splitting of the flow of control into multiple threads (in the case of multiple targets). They are used to model concurrency in parallel systems.

For example, in Figure 35, the action states `AllocateResources`, `CoordinateResources`, 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.

Actions may be grouped into ***swimlanes*** 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 36, the `Dispatcher` swimlane groups all the

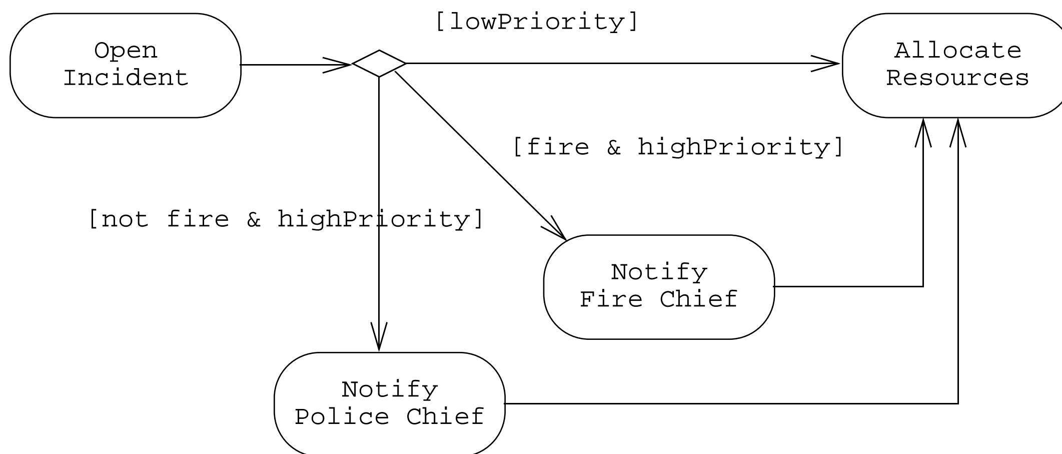


**FIGURE 34.** Example of decision in the `OpenIncident` process. If the `Incident` is a fire and is of high priority, the `Dispatcher` notifies the `Fire Chief`. If it is an `Incident` of high priority that is not a fire, the `Police Chief` is notified instead. In all cases, the `Dispatcher` allocates resources to deal with the `Incident`.

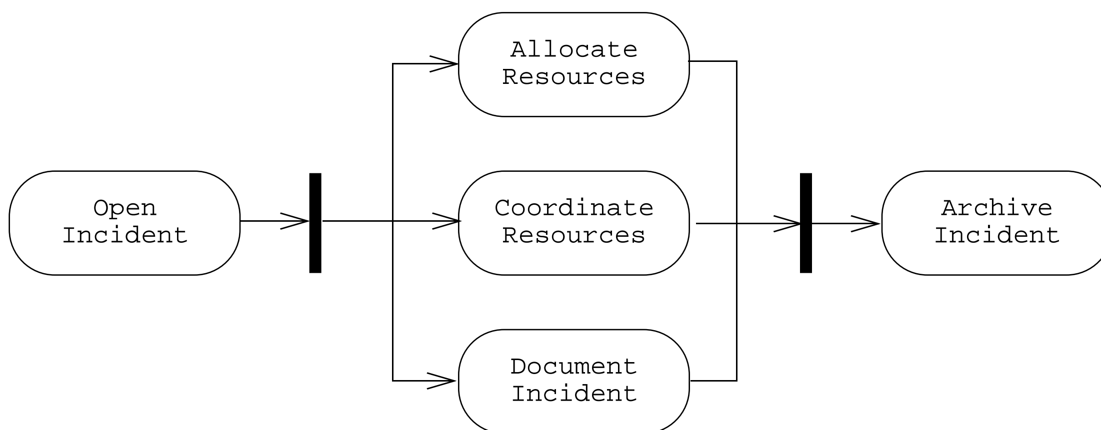


**FIGURE 35.** Example of complex transitions.

actions that are performed by the `Dispatcher` object. The `FieldOfficer` swimlane denotes that the `FieldOfficer` object is responsible for the `DocumentIncident` action.

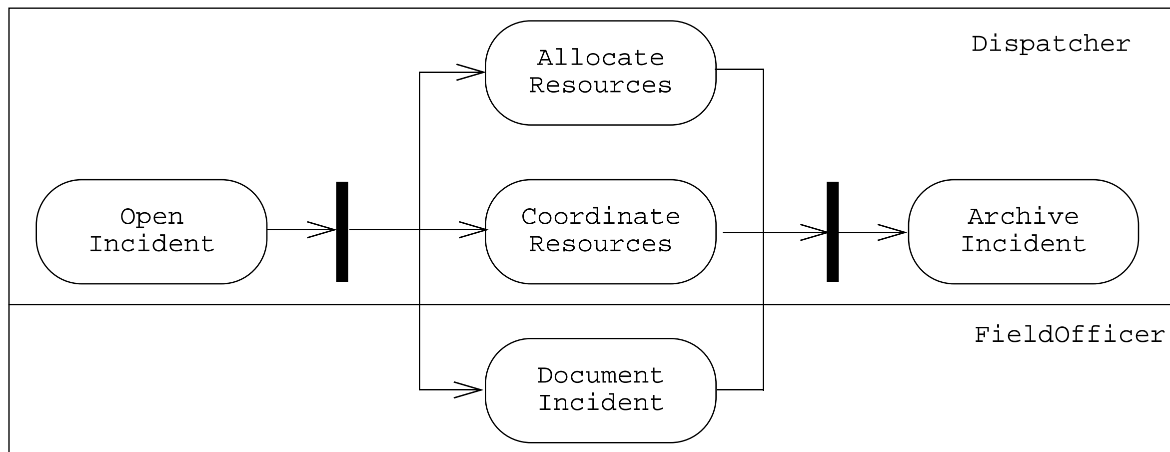


**FIGURE 36.** Example of swimlanes.

### *Applying activity diagrams*

Activity diagrams provide a task centric view of the behavior of an object. They can be used, for example, for describing 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 for describing the processes and activities during software development, in Chapter 3, *Software Life Cycle* and Chapter 4, *Project Management*.

## 2.4.5. Diagram organization

Models of complex systems quickly become complex as developers refine them. Most developers can manage complexity up to 7 ± 2 items. 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, classes, or activities. They are used to reduce the complexity of large models by defining scopes of understanding. For example, Figure 37 displays use cases of the FRIEND emergency response 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.

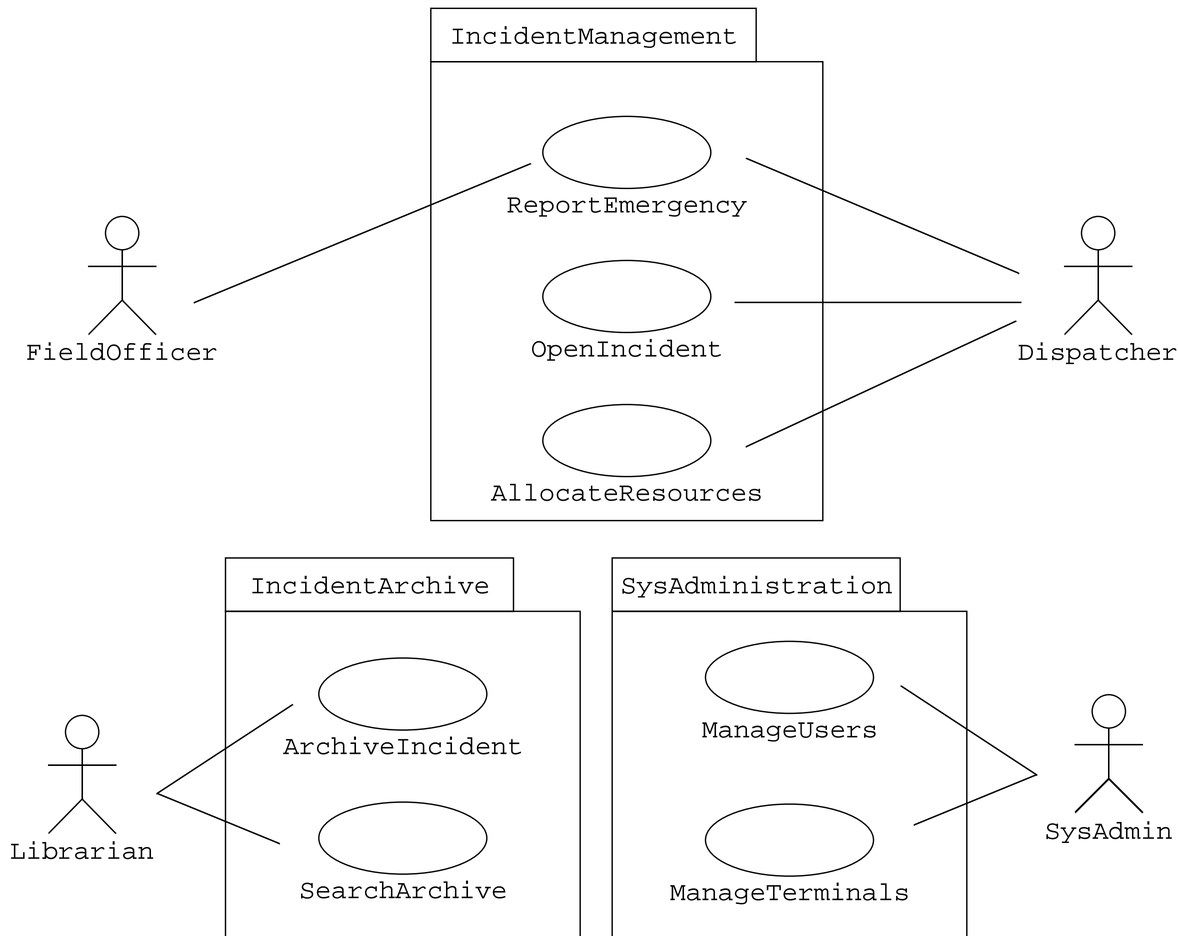


**FIGURE 37.** Example of packages: use cases of FRIEND organized by actors (UML use case diagram).

Figure 37 and Figure 38 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 while `Dispatcher` and `Incident` are part of the `DispatcherStation`. Figure 37 displays the packages with the model elements they contain while Figure 38 displays the same information without the contents of each package. Figure 38 is a higher level picture of the

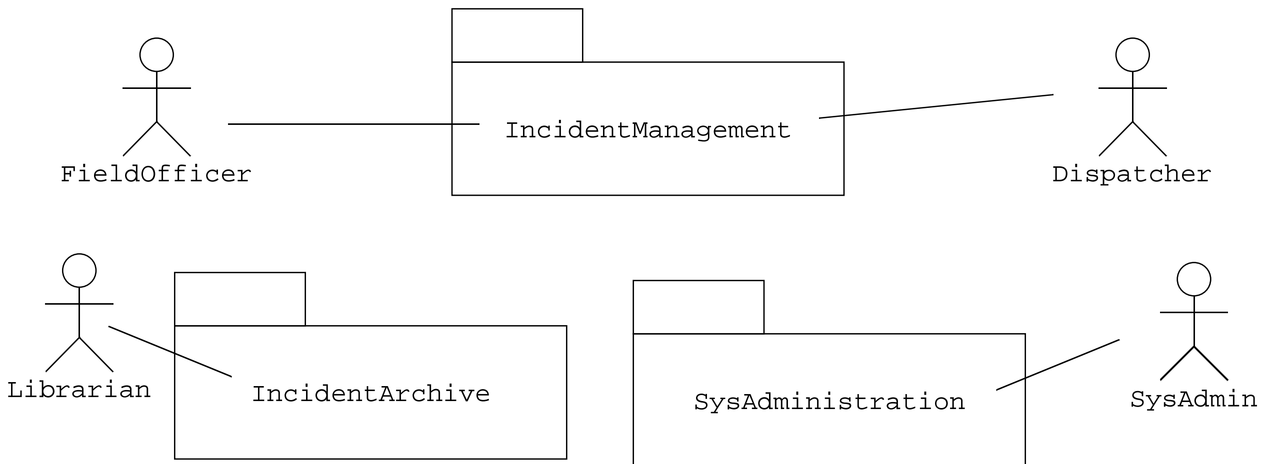


**FIGURE 38.** Example of packages: this figure displays the same packages as Figure 37 except that the details of each packages are suppressed (UML use case diagram).

system and would be used for discussing system level issues while Figure 37 is a more detailed view that can be used to discuss the content of specific packages.

**UML definitions related to diagram organization:**

- ***Packages*** - A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages. A system may be thought of as a single high-level package, with everything else in the system contained in it.
- ***Notes*** - A comment attached to an element or a collection of elements. A note has no semantics.

Packages are used to tame complexity 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 while the other packages are said to refer to the modeling element. Note, that packages are organizing constructs, not objects. They have no behavior associated with them 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

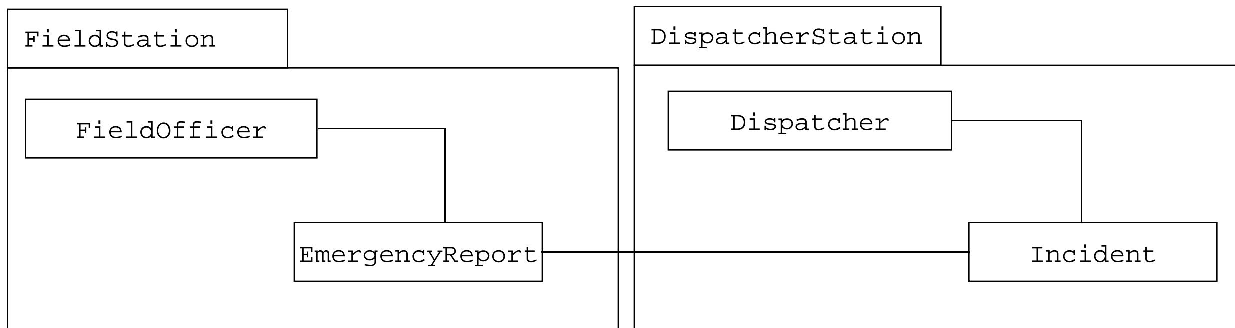


**FIGURE 39.** Example of packages. The `FieldOfficer` and `EmergencyReport` classes are located in the `FieldStation` package while the `Dispatcher` and `Incident` classes are located on the `DispatcherStation` package. A given class can appear in more than one package, however, it is owned by exactly one package.

constraints that cannot be expressed in UML otherwise. Figure 40 provides an example of note.

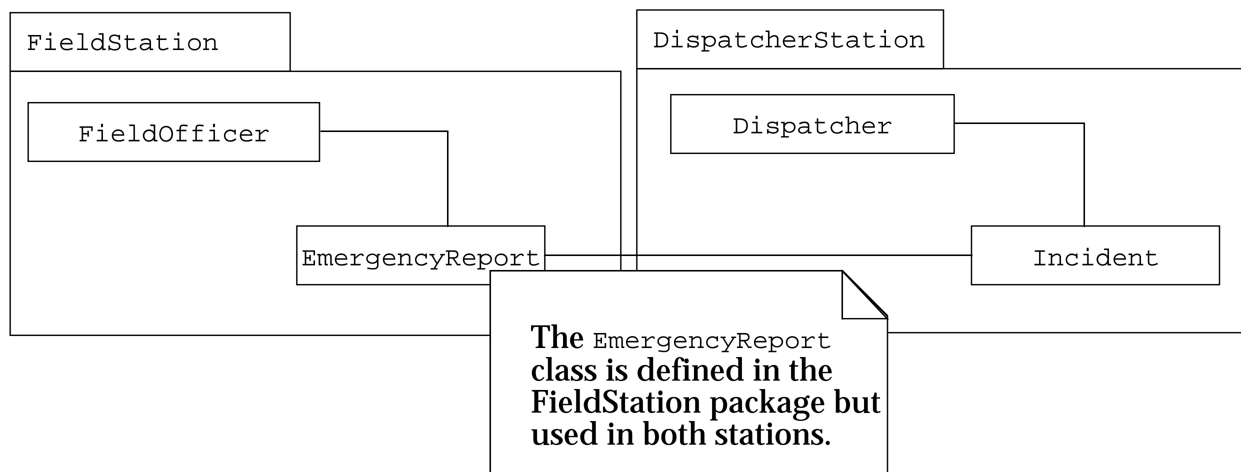


**FIGURE 40.** An example of note. Notes can be attached to a diagram or a specific element in a diagram.

## 2.5. Exercises

1. Draw an object diagram representing this book. All objects and links in this diagram should be instances of the classes which are in the previous diagram. Include attribute values only for the title of each chapter.
2. Draw a class diagram representing the chapter structure of books. Consider both aggregation relationships (a book contains chapters) and associations (chapters cross reference other chapters). Chapters also have attributes denoting the number of pages, figures, its title, abstract, and contents. Include multiplicity on all relationships.
3. Assume the authors of this book and you, the reader, are actors. Identify and describe at least three use cases that involve one or more of these actors.
4. Draw a sequence diagram for one of the use cases you described in the previous exercise. This sequence diagram should have one column for each actor and for each object (of exercise 2) which are involved in the use case.

## 2.6. References

[Booch, 1994] G. Booch, *Object-Oriented Analysis and Design with Applications,* Second Edition, Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[Coad et al. 1995] P. Coad, D. North, M. Mayfield, *Object models: strategies, patterns, & applications.* Prentice-Hall, Englewood Cliffs, 1995.

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

[FRIEND, 1994] FRIEND Project Documentation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 1992-95.

[Harel, 1987] D. Harel, ˇStatecharts: A Visual Formalism for Complex Systems,ˇ *Science of Computer Programming*, pp. 231-274, 1987.

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

[Liskov & Guttag, 1986] B. Liskov & J. Guttag, *Abstraction and Specification in Program Development,* MIT Press, McGraw-Hill, New York, 1986.

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

[Mellor & Shlaer, 1998] S. Mellor and S. Shlaer, Recursive Design Approach, Yourdon Press, Prentice Hall, 1998.

[Popper, 1992] K. Popper. *Objective Knowledge: an Evolutionary Approach.* Clarendon Press,

Oxford, 1992.

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

[Spivey, 1989] J. M. Spivey, *The Z Notation, A Reference Manual.* Prentice Hall International (UK) Ltd., Hemel Hempstead, Hertfordshire, U.K. 1989.

[10] UML1.1 Specification, http://www.rational.com/uml, Rational Software Corporation, Santa Clara, CA. 1997.

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