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 Mellor and Shlaer [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).

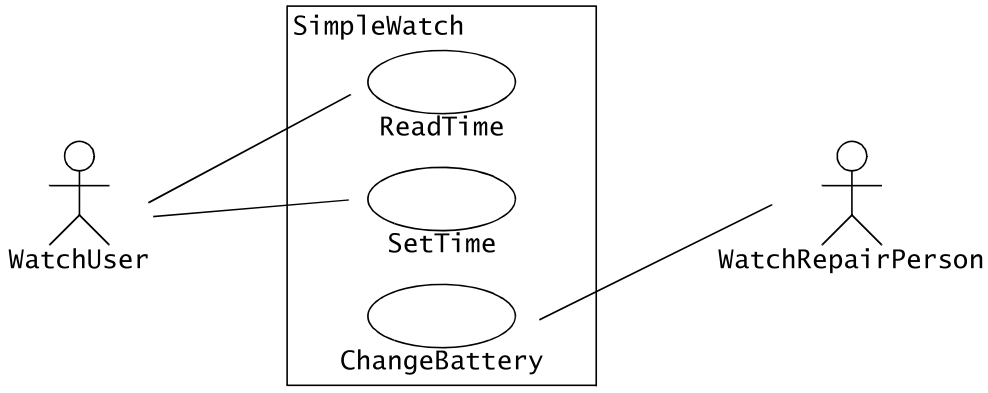

Figure 2-1 A UML use case diagram describing the functionality of a simple watch. The `WatchUser` actor may either consult the time on her watch (with the `ReadTime` use case) or set the time (with the `SetTime` use case). However, only the `WatchRepairPerson` actor can change the battery of the watch (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.

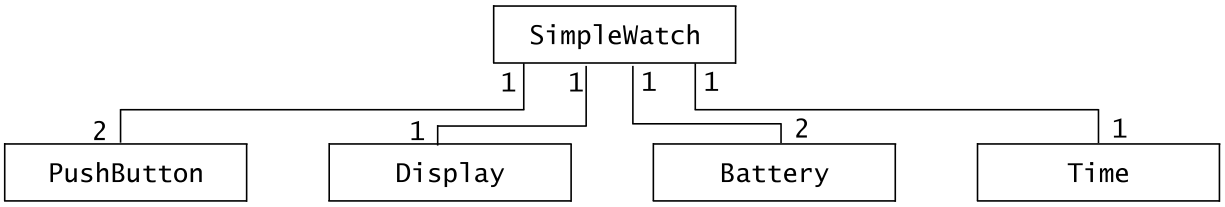

Figure 2-2 A UML class diagram describing the elements of a simple watch.

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

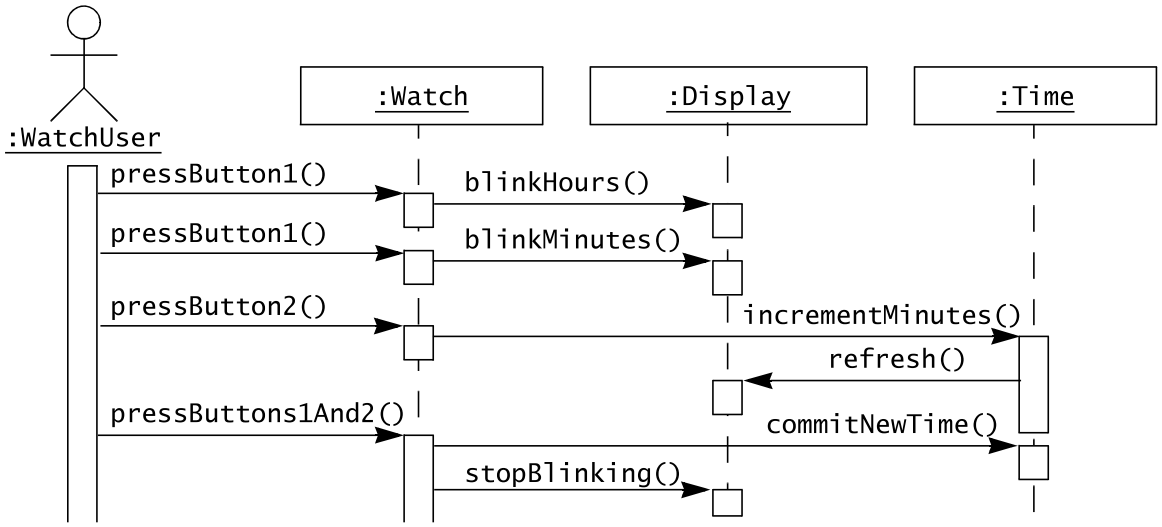

Figure 2-3 A UML sequence diagram for the `Watch`. The left-most column represents the timeline of the `WatchUser` actor who initiates the use case. The other columns represent the timeline of the objects that participate in this use case. Object names are underlined to denote that they are instances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

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`. A small black circle initiates that `BlinkHours` is the initial state. A circle surrounding a small black circle indicates that `StopBlinking` is a final state. 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

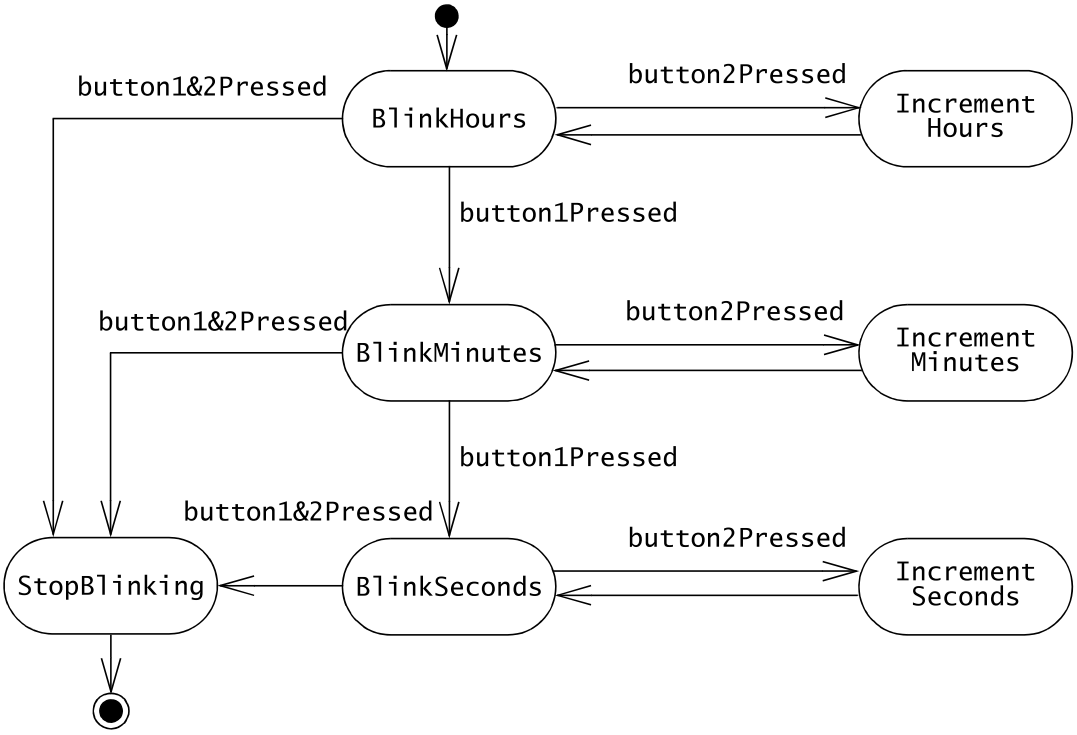

Figure 2-4 A UML state machine diagram for `SetTime` use case of the `Watch`.

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

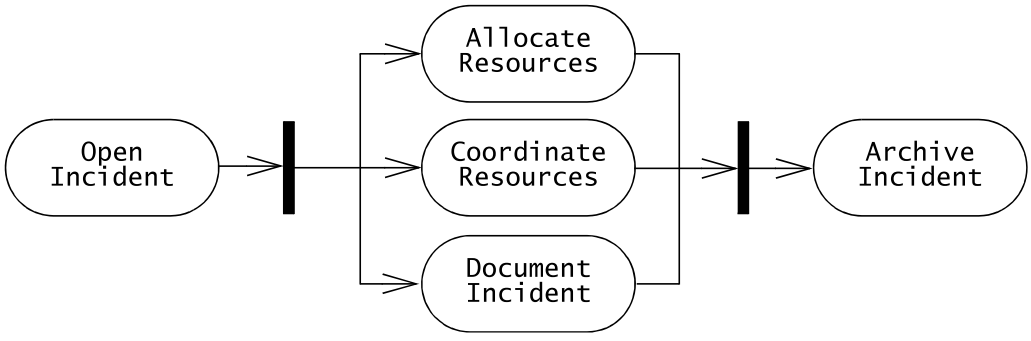

Figure 2-5 An example of a UML activity diagram. Activity diagrams represent behavior in terms of activities and their precedence constraints. The completion of an activity triggers an outgoing transition, which in turn may initiate another activity.

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**, **model**, and **view**, 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,