Object-Oriented Software Engineering

Practical Software Development using UML and Java

Second edition

Timothy C. Lethbridge
Robert Laganière

The McGraw·Hill Companies

London • Burr Ridge, IL • New York • St. Louis • San Francisco •Auckland
Bogotá • Caracas • Lisbon • Madrid • Mexico • Milan • Montreal • New Delhi
Panama • Paris• San Juan • São Paulo • Singapore •Tokyo • Toronto

8 Modeling interactions and behavior

In Chapters 5 and 6 we showed you how to use class diagrams to build a static model of objects in a software system.

In this chapter, we look at how to model system dynamics, focusing on two aspects: interactions and behavior. An interaction model shows a set of actors and objects interacting by exchanging messages. A behavior model shows how an object or system changes state in reaction to a series of events.

In this chapter you will learn about the following

- Two types of UML interaction diagram used to model detailed scenarios of system execution: sequence diagrams and communication diagrams.
- State and activity diagrams, two other UML diagram types that are used to model the possible behavior of a system.

8.1 Interaction diagrams

Interaction diagrams are used to model the dynamic aspects of a software system – they help to visualize how the system runs. They show how a set of actors and objects communicate with each other to perform the steps of a use case, or of some other piece of functionality. The set of steps, taken together, is called an *interaction*.

Interaction diagrams can show several different types of communication. These include messages exchanged over a network, simple procedure calls, and commands issued by an actor through the user interface. Collectively, these are referred to as *messages*.

The following elements can be found in an interaction diagram:

- **Instances of classes or actors**. As discussed in Chapter 5, instances of classes (i.e. objects) are shown as boxes with the class and object identifier underlined. Actors are shown using the same stick-person symbol as in use case diagrams, introduced in Chapter 4.
- **Messages**. These are shown as arrows from actor to object, or from object to object. One of the main objectives of drawing interaction diagrams is to better understand the sequence of messages.

Since you need to know the actors and objects involved in an interaction, you should normally develop a class diagram and a use case model before starting to create an interaction diagram.

Two kinds of diagrams are used to show interactions: *sequence diagrams* and *communication diagrams*. Both contain similar information about an interaction, although sequence diagrams have notations that make them somewhat more powerful. Sequence diagrams explicitly show the sequence of events on a time line, whereas communication diagrams are more compact.

Sequence diagrams

A sequence diagram shows the sequence of messages exchanged by the set of objects (and optionally an actor) performing a certain task. Figure 8.1 gives an example.

The objects are arranged from left to right across the diagram – an actor that initiates the interaction is often shown on the left. The vertical dimension represents time. The top of the diagram is the starting point, and time progresses downwards towards the bottom of the diagram. A vertical dashed line, called a *lifeline*, is attached to each object or actor. The lifeline becomes a box, called an *activation box*, during the period of time that the object is performing computations. The object is said to have *live activation* during these times.

A message is represented as an arrow between activation boxes of the sender and receiver. You give each message a label; it can optionally have an argument list and a response. The complete syntax is as follows:

```
response:=message(arg,...)
```

Figure 8.1 shows how the classes work together to allow a student actor to register in a course; the corresponding class diagram is shown in Figure 8.2.

There are three objects and one actor involved in this interaction. A `Student` object and a `CourseSection` object exist initially; a `Registration` object is created as the interaction proceeds. A creation message is shown using a dashed line with the label `create`. Note the different types of arrowheads used by the create message and the others; it is important to use the correct arrowheads to conform to UML 2.0 syntax.

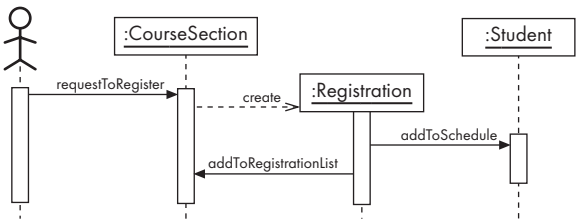

Figure 8.1 **A simple sequence diagram showing the student registration process**

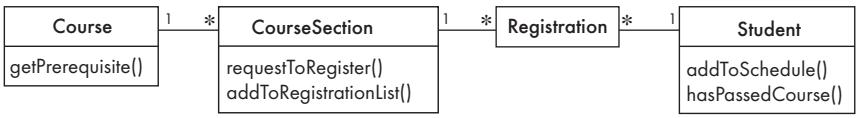

Figure 8.2 **Class diagram for the sequence diagrams shown in Figures 8.1, 8.3 and 8.4**

The objects that exist initially should be lined up along the top of the diagram. Since the `Registration` is created later, its box appears further down, at the time when it is created. Unfortunately, many tools can only draw diagrams in which all the objects appear at the top, as shown in Figure 8.3. However, the create message still makes it clear when the object is created.

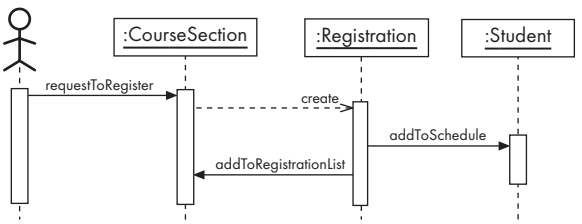

Figure 8.3 **Sequence diagram similar to Figure 8.1 as it would have to be drawn by most tools, which require all objects to be listed along the top of the diagram**

The actor initiates the interaction via the user interface; the user interface sends a `requestToRegister` message to the `CourseSection`, which in turn creates a `Registration`. The `Registration` object then asks the `Student` to add it to the list of courses the student is taking, and also asks the `CourseSection` to add it to the list of registered students.

The labels on the messages in Figure 8.1 correspond to operations in Figure 8.2. The reception of a message by an object causes one of its methods to be run. Sequence diagrams are therefore useful for identifying the operations that have to be included in each class.

Often, when an actor interacts with a system, a corresponding object will exist that contains information about that actor. In Figure 8.1, the actor shown on the

far left is most likely a student, whose corresponding object is shown on the far right. Both are abstractions of the same reality, but should not be confused.

As with class diagrams, interaction diagrams can be drawn at various levels of detail. The level of detail you choose depends on what you wish to communicate. For example, Figure 8.4 provides more detail than is presented in Figure 8.1.

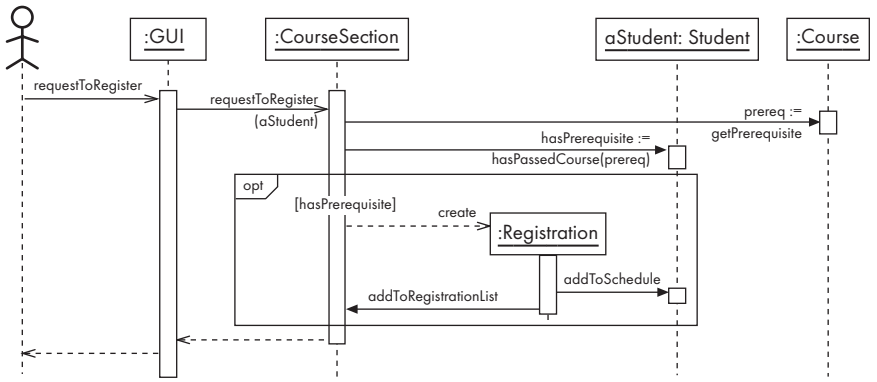

Figure 8.4 **A sequence diagram showing more detail about the student registration process including an optional combined fragment**

- Figure 8.1 showed the user directly interacting with a `CourseSection` object. In reality, the user interacts with the user interface, which in turn interacts with the rest of the data in the system. Figure 8.4 shows this more clearly.
- Figure 8.4 gives the arguments and return values of certain messages. For example, the `requestToRegister` message has `aStudent` as an argument. This same object is also the destination of two messages, therefore the second-to-right object has been labeled `aStudent:Student` to make this clear.
- Figure 8.4 makes use of a *combined fragment* marked 'opt'. A combined fragment is a subsequence of an interaction that is special in some way, and is shown within a box. The 'opt' label means that it may or may not occur. A Boolean *condition*, written within square brackets, describes the circumstances when it will occur. In this case, the condition is written over the `:CourseSection` lifeline, and indicates that the subsequence in the combined fragment will only occur if the `hasPrerequisite` variable (the return value of the previous message) is `true`.
- Sometimes a message is sent, but the reply to that message is sent back after considerable delay. A dashed line from the `CourseSection` to the GUI in Figure 8.4 indicates when the reply to the original `requestToRegister` message is sent.

In some cases, a sequence of messages must be repeated – in other words *iteration* must occur. You show iteration using a combined fragment marked 'loop', as is illustrated in Figure 8.5. The number of times to loop is specified

using the syntax `min..max`. In Figure 8.5 the `getSubtotal` message will be sent to `numPurchase` different `Purchase` objects. The class diagram for this example is given in Figure 8.6.

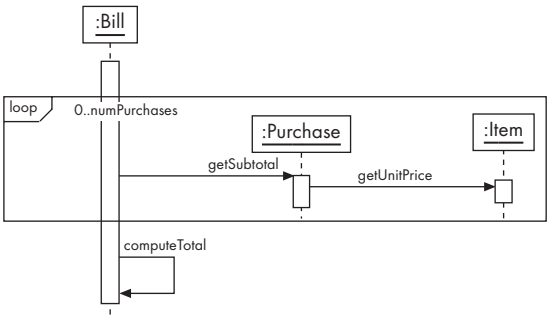

Figure 8.5 A sequence diagram showing a loop fragment

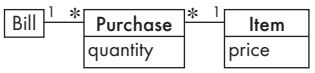

Figure 8.6 Class diagram for Figure 8.5

Figure 8.5 also shows the case of a message being sent from an object to itself.

A sequence diagram can show the destruction of an object using a big ✕ symbol on a lifeline. For example, Figure 8.7 shows what might happen when a booking in the airline system is canceled. The corresponding class diagram is Figure 5.32.

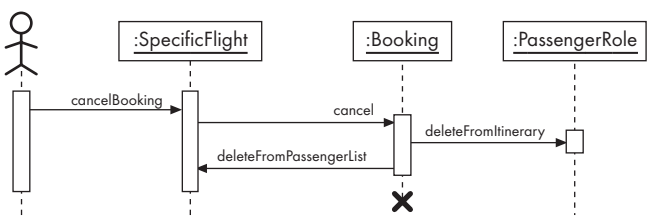

Figure 8.7 Illustration of deleting an object in a sequence diagram

Exercises

E152 Draw sequence diagrams representing the following interactions:

(a) A client searches for a book in a library. He or she then asks to borrow the book. If a copy of this book is available, a loan object is created.

(b) To obtain the cousins of an individual, you must determine his or her two parents. You then obtain the siblings of these parents. For each of these siblings, you obtain their children (refer to Figure 5.25(c)).

(c) A client wishes to open a new account at a bank branch. To do so, his instance of class `Client` must first be retrieved from the central bank server. For a new client, an instance of `Client` must be created. An instance of `BankAccount` is then created using the `Client` object. A deposit must then immediately follow, to complete the account creation process.

Communication diagrams

A communication diagram shows several objects working together. It appears as a graph with a set of objects and actors as the vertices.

A communication diagram is very much like an object diagram except that, as we will discuss below, it shows *communication* links instead of links of associations. It also has much in common with a sequence diagram, except that lifelines, activation boxes and combined fragments are absent. Instead, you draw a communication link between each pair of objects involved in the sending of a message; the messages themselves are attached to this link.

You represent a message using an arrow, labeled with the message name and optional arguments. You specify the order in which messages are sent by prefixing each message using some numbering scheme.

Figures 8.8 and 8.9 show examples of communication diagrams describing the same interactions as the sequence diagrams of Figures 8.1 and 8.4.

Figure 8.8 **A communication diagram representing the same interaction as the sequence diagram of Figure 8.1**

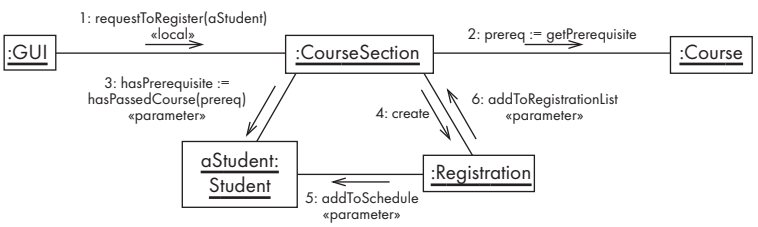

Figure 8.9 **A communication diagram corresponding to the sequence diagram of Figure 8.4**

Communication links can exist between two objects whenever it is possible for one object to send a message to the other one. Several situations can make this possible:

- The classes of the two objects are joined by an association. This is the most common case. In Figure 8.9, message 2 is sent over such a link. If all messages are sent in the same direction, then probably the association can be made unidirectional.
- The receiving object is stored in a *local* variable of the sending method (but the objects are not yet joined by an association). This can happen when the receiving object is created in the sending method, such as in message 4 of Figure 8.9, or when some computation returns an object that is only kept in a local variable. In message 1 of Figure 8.9 we tag such a message with the stereotype «local».
- A reference to the receiving object has been received as a parameter of an earlier message to the sender. In Figure 8.9, we tag such messages with the stereotype «parameter». For example, in message 3 the `Student` was previously passed as a parameter to the `CourseSection`.
- The receiving object is *global*. This is the case when a reference to an object can be obtained using a public static method (e.g. using the Singleton pattern). The stereotype «global» could be used in this case. Note that the use of global data should be minimized, as we will discuss in Chapter 9.
- The objects communicate over a network. The stereotype «network» could be used to show this.

Exercises

E153 Draw a communication diagram corresponding to Figure 8.5.

E154 Draw communication diagrams for the interactions described in Exercise E152.

How to choose between using a sequence or a communication diagram

Since sequence and communication diagrams contain much the same information, you have to decide which of the two you should draw. Sequence diagrams are often the better choice in the following four situations:

- You want the reader to be able to easily see the order in which messages occur.
- You want to build an interaction model from a use case. Use cases already have a sequence of steps; sequence diagrams expand on these to show which objects are involved.
- You need to show details of messages, such as parameters and return values. Doing so on communication diagrams can result in clutter.
- You need to show loops, optional sequences and other things that can only be properly expressed using combined fragments.

On the other hand, you may prefer a communication diagram when you are deriving an interaction diagram from a class diagram. This is because communication diagrams are effectively object diagrams with communication links instead of association links. Communication diagrams can in fact be used to help validate class diagrams – a communication diagram might suggest, for example, that you should add a new association in order to make the interaction possible.

Communication diagrams, patterns and collaborations

A communication diagram can be used to represent aspects of a design pattern – such as those discussed in Chapter 6. Figure 8.10(a) shows the two steps involved in the main interaction of the Proxy pattern. First, a client object makes a request to a «Proxy» object. Then, if the «HeavyWeight» is needed and is not already loaded, the «Proxy» causes it to be loaded, before returning the result to the client. Compare this to Figure 6.13.

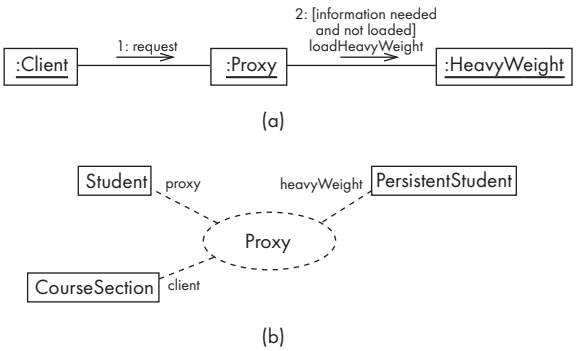

Figure 8.10 **Using a communication diagram and a collaboration symbol to represent the Proxy design pattern**

The *collaboration* between the classes involved in an interaction can be represented in a class diagram using a dashed ellipse, as shown in Figure 8.10(b). Dashed lines link the ellipse to classes that fulfill the various roles of the collaboration. Here, for example, `PersistentStudent` has the role of «HeavyWeight». This notation is particularly convenient for showing the classes fulfilling the roles in a design pattern.

8.2 State diagrams

A *state diagram*, also known as a *state machine diagram*, is another way of expressing dynamic information about a system. It is used to describe the externally visible behavior of a system or of an individual object.

At any given point in time, the system or object is said to be in a certain *state*. It remains in this state until an *event* occurs that causes it to change state. Being

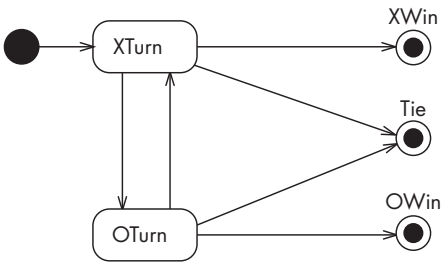

Figure 8.11 State diagram of a tic-tac-toe (noughts and crosses) game

in a state means that it behaves in a specific way in response to any events that occur. You represent a state using a rounded rectangle that contains the name of the state. Figure 8.11, for example, contains two such states.

Several different types of event can cause the system to change from one state to another. In each state, the system behaves in a different way. A *transition* represents a change of state in response to an event, and is considered to occur instantaneously – that is, to take no time. You draw a transition using an arrow connecting two states. You can also show a label on a transition; this represents the event that causes the change of state.

There are two other special symbols that can appear on a state diagram:

- A black circle represents the *start state*. When the system or object starts running, it immediately takes a transition from the start state to a regular state. There should be only one start state in each top-level state diagram, and there should be only one unlabelled transition pointing out of the start state.
- A black circle with a ring around it represents an *end state*. The system or object finishes its work when such a state is reached. There can be more than one end state in a state diagram. The symbol is supposed to resemble a target.

In addition, there are several other pieces of notation that can be placed inside states and on transitions to describe their behavior more precisely. We will discuss these in the context of examples below.

Figure 8.11 represents the game of tic-tac-toe, also known as noughts and crosses. Since player X always goes first, the initial transition from the start state points to the 'X Turn' state. From then on, the game alternates between 'X Turn' and 'O Turn' states, until the game ends. There can be three possible outcomes of the game, represented by the three end states: X can win, O can win or there can be a tie. The 'Tie' end state can be reached from both 'X Turn' state and 'O Turn' state.

Notice that there is more than one transition leading out of both 'X turn' and 'O turn' states. In this situation, the system will take the transition that corresponds to the *first* occurring event: a move resulting in a win, a tie or continued play. In the remainder of this chapter we will use event labels of various kinds to make clear what causes each transition.

Elapsed-time transitions

The event that triggers a transition can be a certain amount of elapsed time. Figure 8.12(a) illustrates the use of such *elapsed-time* transitions to model a simple traffic signal.

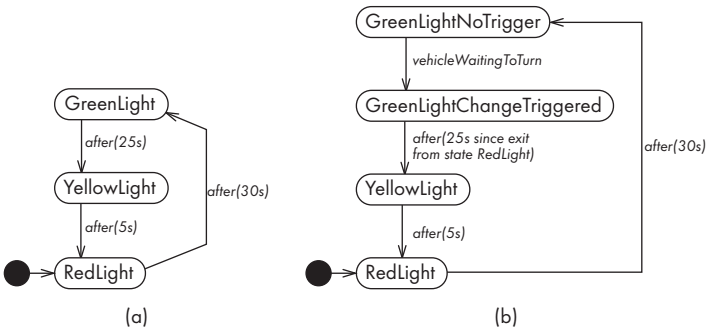

Figure 8.12 **State diagrams of a simple traffic light, illustrating elapsed-time transitions**

There are three main states in Figure 8.12(a), corresponding to the three colors of the traffic signal at point 1 of Figure 8.13. The initial state has a transition to 'RedLight' state to indicate what happens when the system starts up. After startup, the system indefinitely rotates among green, yellow and red, therefore there is no end state.

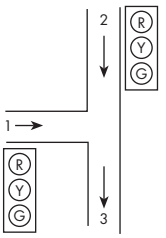

Figure 8.13 **Street layout for the traffic signals of Figure 8.12**

The red light only stays on for 30 seconds, at which time the green light comes on. The green light stays on for 25 seconds, at which time the system moves to the 'YellowLight' state. After five more seconds the yellow light gives way to the red light again. Note that the traffic signal at point 2 of Figure 8.13 would have the same state diagram, except that when it goes green, the signal at point 1 would go red, and vice versa.

Figure 8.12(b) extends the above scenario by showing a slightly different pattern for the signal at point 2. In this case, traffic moving from point 2 to 3 always has a green light unless a vehicle arrives at point 1 and triggers a sensor. When the sensor at point 1 is triggered, the system moves to 'YellowLight' state, but only after the traffic coming from point 2 has had at least 25 seconds of green light. An extra state 'GreenLightChangeTriggered' is used to model this latter situation. Without this state and its outgoing transition, a steady series of

Traffic lights work differently in different locales

You might notice that the traffic light state diagram of Figure 8.12 does not work in quite the same way as your local traffic lights do. For example, in many countries, just before the light goes green, the red and yellow lights come on simultaneously to let drivers know they should prepare to move. The timing of lights and the rules for the triggering of sensors can differ at each intersection. Designing a generic and internationalized traffic light system would therefore require considerable domain analysis.

vehicles arriving at point 1 would prevent the light at point 2 from staying green long enough for traffic to flow.

Transitions triggered by a condition becoming true

Figure 8.14 gives an example of a state diagram containing transitions that are made whenever certain conditions become true. A condition can be distinguished from an event name since it contains a Boolean operator. The two conditions in this figure are `classSize >= minimum`, and `classSize >= maximum`.

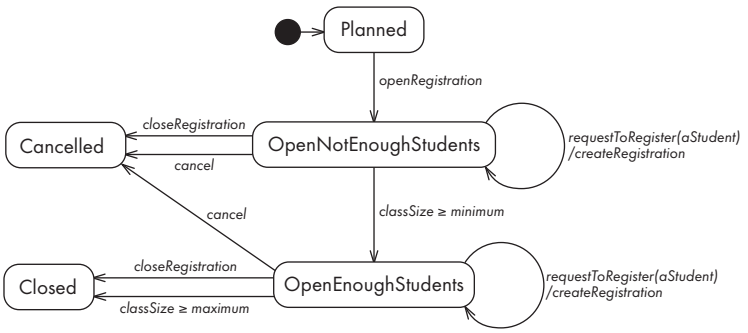

Figure 8.14 State diagram of a CourseSection class

The figure shows the behavior of instances of the `CourseSection` class shown in Figure 8.2. When it is first created, a `CourseSection` is in 'Planned' state and is not yet ready to receive students. When registration is opened, the system moves to 'OpenNotEnoughStudents' state.

In this state, the `CourseSection` can accept requests to register, but the course will not actually be taught until the class size reaches a certain minimum. The object evaluates the `classSize >= minimum` condition every time anything occurs that could make the condition `true`; as soon as it becomes `true`, a transition is taken to 'OpenEnoughStudents' state.

If requests to register continue while in 'OpenEnoughStudents' state, the object will eventually exceed a predefined maximum number of students, at which time it automatically moves to 'Closed' state. The course section can also be explicitly closed by a `closeRegistration` event, indicating that the registration deadline has passed. If there are not enough students, closing a course section has the same effect as canceling it.

There is no end state (the target symbol) because we will keep a permanent record of the course section, whatever happens.

The `requestToRegister` transitions in Figure 8.14 shows that a transition can lead from a state back to the same state. Also, the `/createRegistration` notation designates an *action*; this is discussed in the next subsection.

Exercises

E155 Enhance Figure 8.11 to show the conditions that should be present to determine when the system should transition to each of the end states ('X Win', 'O Win' and 'Tie') or to the regular states 'X Turn' and 'O Turn'.

E156 Given the state diagram in Figure 8.14, in what state would the system be in after the following sequences of events? Assume that `minimum` is set to 3 and `maximum` is set to 5.

(a) `openRegistration`, `requestToRegister` (repeated twice), `cancel`.

(b) `openRegistration`, `requestToRegister` (repeated 8 times in total), `closeRegistration`

E157 Enhance Figure 8.14 to handle the following situations:

(a) The course section can be canceled when in any state, except after it has been taught.

(b) A student can drop out at any time, except after the course section has been taught.

E158 Figure 8.12(b) shows the state diagram for a sensor-activated light at point 2 of Figure 8.13, where the light remains green until the sensor is triggered by traffic wishing to move from point 1 to point 3. Show the corresponding diagram for point 1 (where the light remains red until the sensor is triggered).

E159 Draw a state diagram for a four-way intersection (also known as a crossroads). There will be several different signals controlling the different directions of flow. You have to consider which lights are on at any given time. In the basic case, the signals facing north will look the same as the signals facing south, and the signals facing east will appear the same as those facing west. Thus the system as a whole can have the following states, where NS means north and south, and EW means east and west: 'NSRed-EWGreen', 'NSChanging-EWYellow', ` 'NSGreen-EWRed' and 'NSYellow-EWChanging'.

E160 Enhance your answer to the last exercise by considering a set of traffic signals for an intersection with turning lanes controlled by separate turning signals. You may want to study a real intersection to understand what states can occur.

E161 Model a simple vending machine that can be in four states: 'Waiting', 'Receiving Money', 'Returning Money', and 'Delivering Item'.

Activities and actions in state diagrams

You can represent two kinds of computations using state diagrams. These computations are called *activities* and *actions*.

An activity is something that occurs over a period of time and takes place while the system is in a state. The system may take a transition out of the state in response to completion of the activity. However, if some other transition is triggered first, then the system has to terminate the activity as it leaves the state.

An activity is shown textually within a state box by the word 'do' followed by a '/' symbol, and a description of what is to be done. When you have details such as actions in a state, you draw a horizontal line above them to separate them from the state name.

Figure 8.15 shows a jukebox with just two states. In 'ProposeSelection' state, the system waits for the user to press a button, selecting some music. In the 'MusicPlaying' state, the system plays the chosen music until it comes to an end. The system then takes a transition back to 'ProposeSelection' state.

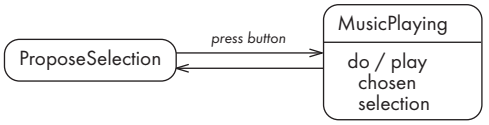

Figure 8.15 **State diagram for a jukebox, illustrating an activity in a state**

An *action* is something that takes place effectively *instantaneously* in any of the following situations:

- When the system takes a particular *transition*.
- Upon entry into a particular state, no matter which transition causes entry into that state.
- Upon exit from a particular state, no matter which transition is being taken.

An action should take place with no noticeable consumption of time; therefore it should be something simple, such as sending a message, starting a hardware device or setting a variable.

An action is always shown preceded by a slash ('/') symbol. If the action is to be performed during a transition, then the syntax is `event/action`. If the action is to be

Action semantics

UML allows you to specify actions using whatever notation you find suitable. However, the UML specification describes over 40 classes of action that you can represent in models. Some examples are actions to: send a message, change a variable, create or destroy an object, and create or destroy a link.

performed when entering or exiting a state, then it is written in the state box with the notation `enter/action` or `exit/action`.

Figure 8.16 illustrates the use of actions in the state diagram for a garage door opener. Upon entry into each state, a particular action occurs that has an effect on the garage door motor: the motor controller can be told to start the motor running forwards (opening the door), to start the motor running in reverse (closing the door), or to stop the motor. For example, whenever the garage door becomes completely open, it enters 'Open' state; the action taken when this occurs is to stop the motor.

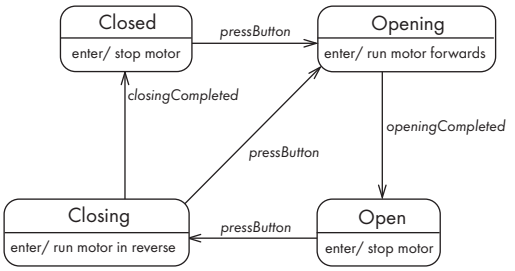

Figure 8.16 **State diagram for a garage door opener, showing actions triggered by entry into a state**

Normally the controller cycles around the four states in a clockwise manner, a button being used to initiate opening or closing. When closing the door, however, there is a safety mechanism: if the button is pressed while the door is closing, the motor is immediately thrown into the forwards direction, causing the door to start opening again.

Figure 8.17 shows another example of the use of actions. This is a partial state diagram of a tape recorder. In this case, there is an exit action that stops the recording process no matter what causes the 'Recording' state to be exited. In addition, the `startOfTape` transition has a `stop` action that occurs during the transition to the 'Wait' state.

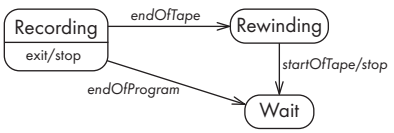

Figure 8.17 **Partial state diagram for a tape recorder showing an action on a transition and an action on exiting a state**

Figure 8.17 is clearly incomplete, since there is no transition out of 'Wait' state. An important part of validating a state diagram is to ensure that each state has at least one outgoing transition, and at least one incoming transition. You should also make sure that it is possible to get from any state to any other state, otherwise the system can reach what is called 'livelock'. We will discuss this concept in Chapter 10.

Nested substates and guard conditions

A state diagram can be nested inside a state. The states of the inner diagram are called *substates*.

Figure 8.18 shows a state diagram of an automatic transmission; at the top level this has three states: 'Neutral', 'Reverse' and a driving state, which is not explicitly named. The driving state is divided into substates corresponding to the three gears that the system automatically chooses. The advantage of the nesting is that it shows compactly that the driving substates are all very similar to each other – in particular, that they can all transition to 'Neutral' at any time, upon the user's command. The start symbol inside the driving state shows that it by default starts at the 'First' substate. However, the user can also manually select 'First' or 'Second' to force the transmission to move into, and stay in, these substates.

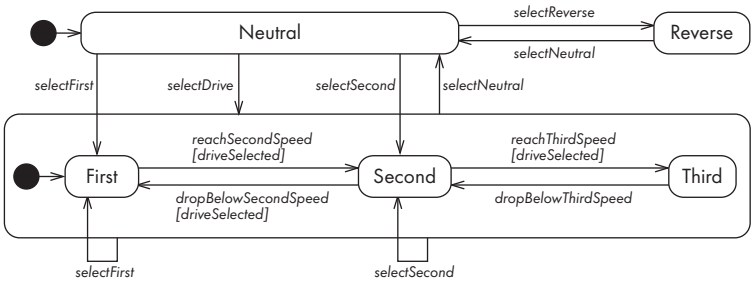

Figure 8.18 **State diagram for a car's automatic transmission showing substates**

The notation `reachSecondSpeed[driveSelected]` illustrates the use of a *guard condition*. The system will only respond to the indicated event (`reachSecondSpeed`) if the condition in square brackets is true. In Figure 8.18, this is used to prevent the transmission from changing gear if the driver had manually selected first or second gear. A guard condition differs from the type of condition we saw in Figure 8.14: a guard condition is only evaluated when its associated event occurs.

Figure 8.19 shows how we have converted Figure 8.14 to use nested substates. Now we need to show only one `cancel` transition and one `requestToRegister` transition. Note that the 'Planned' state has a transition that points directly to the 'NotEnoughStudents' substate, and both the transitions to the 'Closed' state comes directly from the inner 'EnoughStudents' state. Finally, note that we have added an activity to the 'Canceled' state that deletes all registrations.

Exercises

E162 There is a missing transition in Figure 8.18. Study the diagram, and see if you can find it (do not add any new states or event types).

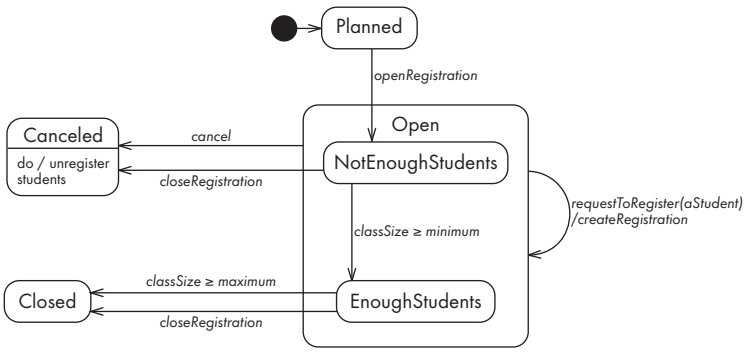

Figure 8.19 **A version of the course section example from Figure 8.14, showing the effect of nested states**

E163 Modify the diagram shown in Figure 8.18 so that it correctly incorporates the 'Park' control of an automatic transmission.

E164 There are several other subtleties of nested states that we will not discuss here. One of the important skills of a software engineer is to be able to look up information directly from documentation. Go to the OMG web site and find the official specification of the latest version of UML. Use this to learn about the 'history' state and how to use it in the context of nested state diagrams.

E165 Draw state diagrams for the following situations:

(a) Modify the jukebox diagram shown in Figure 8.15 so that it has an 'AcceptingMoney' state. The correct amount of money must be accepted prior to the user selecting the music.

(b) Create a state diagram for the microwave oven system described in Example 4.6.

(c) Expand Figure 8.17 to model the general operation of a VCR. It can be in at least the following states: 'Off', 'StandbyForAutomaticRecording', 'AutomaticRecording', 'ManualRecording', 'PlayingTape', 'Showing TV Channel', 'Rewinding', and 'FastForwarding'. Tape operations can only occur if a tape has been inserted. Automatic recording occurs when the VCR is programmed to record a program at a certain time, and for a certain period of time.

(d) Create a state diagram to model the steps involved in programming a VCR to record a program at a certain time. Study a real VCR to understand the possible states. The events are buttons pressed on the VCR's remote control or console. This diagram represents the behavior of part of the user interface of the VCR, whereas Exercise E165(c) represents the behavior of the functional layer.