8

Advanced Interaction Modeling

The interaction model has several advanced features that can be helpful. You can skip this chapter on a first reading of the book.

8.1 Use Case Relationships

Independent use cases suffice for simple applications. However, it can be helpful to structure use cases for large applications. Complex use cases can be built from smaller pieces with the *include*, *extend,* and *generalization* relationships.

8.1.1 Include Relationship

The ***include*** relationship incorporates one use case within the behavior sequence of another use case. An included use case is like a subroutine—it represents behavior that would otherwise have to be described repeatedly. Often the fragment is a meaningful unit of behavior for the actors, although this is not required. The included use case may or may not be usable on its own.

The UML notation for an include relationship is a dashed arrow from the source (including) use case to the target (included) use case. The keyword «*include*» annotates the arrow. Figure 8.1 shows an example from an online stock brokerage system. Part of establishing a secure session is validating the user password. In addition, the stock brokerage system validates the password for each stock trade. Use cases *secure session* and *make trade* both include use case *validate password.*

A use case can also be inserted within a textual description with the notation *include use-case-name*. An included use case is inserted at a specific location within the behavior sequence of the larger use case, just as a subroutine is called from a specific location within another subroutine.

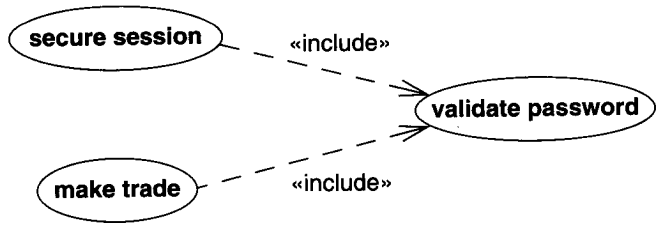

Figure 8.1 Use case inclusion. The *include* relationship lets a base use case incorporate behavior from another use case.

You should not use include relationships to structure fine details of behavior. The purpose of use case modeling is to identify the functionality of the system and the general flow of control among actors and the system. Factoring a use case into pieces is appropriate when the pieces represent significant behavior units.

8.1.2 Extend Relationship

The ***extend*** relationship adds incremental behavior to a use case. It is like an include relationship looked at from the opposite direction, in which the extension adds itself to the base, rather than the base explicitly incorporating the extension. It represents the frequent situation in which some initial capability is defined, and later features are added modularly. The include and extend relationships both add behavior to a base use case.

For example, a stock brokerage system might have the base use case *trade stocks*, which permits a customer to purchase stocks for cash on hand in the account. The extension use case *margin trading* would add the ability to make a loan to purchase stocks when the account does not contain enough cash. It is still possible to buy stocks for cash, but if there is insufficient cash, then the system offers to proceed with the transaction after verifying that the customer is willing to make a margin purchase. The additional behavior is inserted at the point where the purchase cost is checked against the account balance.

Figure 8.2 shows the base use case *trade stocks* for a stock brokerage system. The UML notation for an extend relationship is a dashed arrow from the extension use case to the base use case. The keyword «*extend*» annotates the arrow. The base use case permits simple purchases and sales of a stock at the market price. The brokerage system adds three capabilities: buying a stock on margin, selling a stock short, and placing a limit on the transaction price. The use case *trade options* also has an extension for placing a limit on the transaction price.

The extend relationship connects an extension use case to a base use case. The extension use case often is a fragment—that is, it cannot appear alone as a behavior sequence. The base use case, however, must be a valid use case in the absence of any extensions. The extend relationship can specify an insert location within the behavior sequence of the base use case; the location can be a single step in the base sequence or a range of steps. The behavior sequence of the extension use case occurs at the given point in the sequence. In most cases, an extend relationship has a condition attached. The extension behavior occurs only if the condition is true when control reaches the insert location.

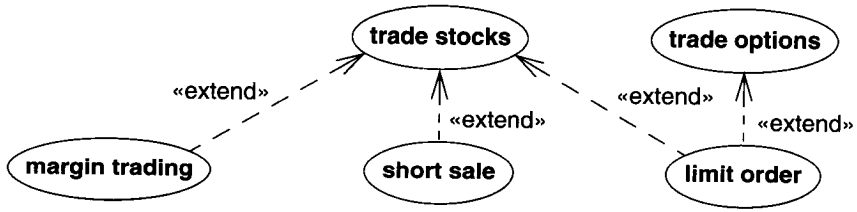

Figure 8.2 Use case extension. The *extend* relationship is like an *include* relationship looked at from the opposite direction. The extension adds itself to the base.

8.1.3 Generalization

Generalization can show specific variations on a general use case, analogous to generalization among classes. A parent use case represents a general behavior sequence. Child use cases specialize the parent by inserting additional steps or by refining steps. The UML indicates generalization by an arrow with its tail on the child use case and a triangular arrowhead on the parent use case, the same notation that is used for classes.

For example, an online stock brokerage system (Figure 8.3) might specialize the general use case *make trade* into the child use cases *trade bonds, trade stocks,* and *trade options*. The parent use case contains steps that are performed for any kind of trade, such as entering the trading password. Each child use case contains the additional steps particular to a specific kind of trade, such as entering the expiration date of an option.

Figure 8.3 Use case generalization. A parent use case has common behavior and child use cases add variations, analogous to generalization among classes.

A parent use case may be abstract or concrete—an abstract use case cannot be used directly. As with the class model, we recommend that you consider only abstract parents and forego concrete ones. Then a model is more symmetric and a parent use case is not cluttered with the handling of special cases. Use cases also exhibit polymorphism—a child use case can freely substitute for a parent use case, for example, as an inclusion in another use case. In all these respects, generalization is the same for use cases and for classes.

In one respect, use case generalization is more complicated than class generalization. A subclass adds attributes to the parent class, but their order is unimportant. A child use case adds behavior steps, but they must appear in the proper locations within the behavior se-

quence of the parent. This is similar to overriding a method that is inherited by a subclass, in which new statements may be inserted at various locations in the parent's method. The simplest approach is to simply list the entire behavior sequence of the child use case, including the steps inherited from the parent. A more general approach is to assign symbolic locations within the parent's sequence and to indicate where additions and replacements go. In general, a child may revise behavior subsequences at several different points in the parent's sequence.

With classes there can be multiple inheritance, but we do not allow such complexity with use cases. In practice, the include and extend relationships obviate the need for multiple inheritance with use cases.

8.1.4 Combinations of Use Case Relationships

A single diagram may combine several kinds of use case relationships. Figure 8.4 shows a use case diagram from a stock brokerage system. The *secure session* use case includes the behavior of the *validate password*, *make trade*, and *manage account* use cases. *Make trade* is an abstract parent with the children—*trade bonds*, *trade stocks*, and *trade options*. Use case *make trade* also includes the behavior of *validate password*. The brokerage system validates the password once per session and additionally for every trade.

The use case *margin trading* extends both *trade bonds* and *trade stocks*—a customer may purchase stocks and bonds on margin, but not options. Use case *limit order* extends abstract use case *make trade*—limit orders apply to trading bonds, stocks, and options. We assume that a *short sale* is only permitted for stocks and not for bonds or options.

Note that the *Customer* actor connects only to the *secure session* use case. The brokerage system invokes all the other use cases indirectly by inclusion, specialization, or extension. The *Securities exchange* actor connects to the *make trade* use case. This actor does not initiate a use case but it is invoked during execution.

8.1.5 Guidelines for Use Case Relationships

Don't carry use case relationships to extremes and lapse into programming. Use cases are intended to clarify requirements. There can be many ways to implement requirements and you should not commit to an approach before you fully understand a problem. Here are some additional guidelines.

- **Use case generalization.** If a use case comes in several variations, model the common behavior with an abstract use case and then specialize each of the variations. Do not use generalization simply to share a behavior fragment; use the include relationship for that purpose.
- **Use case inclusion.** If a use case includes a well-defined behavior fragment that is likely to be useful in other situations, define a use case for the behavior fragment and include it in the original use case. In most cases, you should think of the included use case as a meaningful activity but not as an end in itself. For example, validating a password is meaningful to users but has a purpose only within a broader context.

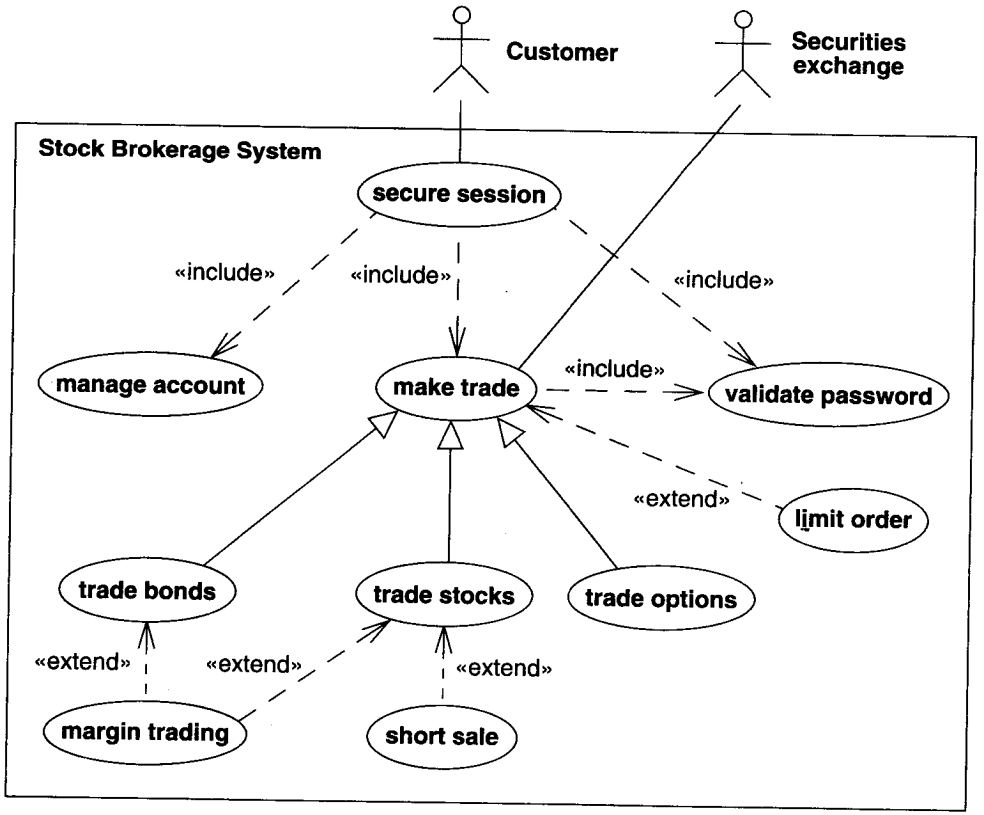

Figure 8.4 Use case relationships. A single use case diagram may combine several kinds of relationships.

- **Use case extension.** If you can define a meaningful use case with optional features, then model the base behavior as a use case and add features with the extend relationship. This permits the system to be tested and debugged without the extensions, which can be added later. Use the extend relationship when a system might be deployed in different configurations, some with the additional features and some without them.
- **Include relationship vs. extend relationship.** The include relationship and the extend relationship can both factor behavior into smaller pieces. The include relationship, however, implies that the included behavior is a necessary part of a configured system (even if the behavior is not executed every time), whereas the extend relationship implies that a system without the added behavior would be meaningful (even if there is no intention to configure it that way).

8.2 Procedural Sequence Models

In Chapter 7, we saw sequence diagrams containing independent objects, all of which are active concurrently. An object remains active after sending a message and can respond to other messages without waiting for a response. This is appropriate for high-level models. However, most implementations are procedural and limit the number of objects that can execute at a time. The UML has elaborations for sequence diagrams to show procedure calls.

8.2.1 Sequence Diagrams with Passive Objects

With procedural code all objects are not constantly active. Most objects are passive and do not have their own threads of control. A passive object is not activated until it has been called. Once the execution of an operation completes and control returns to the caller, the passive object becomes inactive.

Figure 8.5 computes the commission for a stock brokerage transaction. The transaction object receives a request to compute its commission. It obtains the customer's service level from the customer table, then asks the rate table to compute the commission based on the service level, after which it returns the commission value to the caller.

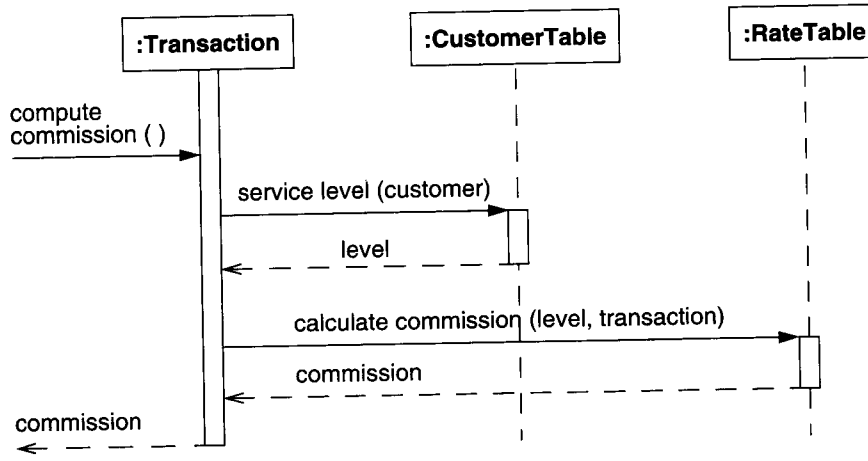

Figure 8.5 Sequence diagram with passive objects. Sequence diagrams can show the implementation of operations.

The UML shows the period of time for an object's execution as a thin rectangle. This is called the ***activation*** or ***focus of control***. An activation shows the time period during which a call of a method is being processed, including the time when the called method has invoked another operation. The period of time when an object exists but is not active is shown as a dashed line. The entire period during which the object exists is called the ***lifeline***, as it shows the lifetime of the object.

8.2.2 Sequence Diagrams with Transient Objects

Figure 8.6 shows further notation. *ObjectA* is an active object that initiates an operation. Because it is active, its activation rectangle spans the entire time shown in the diagram. *ObjectB* is a passive object that exists during the entire time shown in the diagram, but it is not active for the whole time. The UML shows its existence by the dashed line (the lifeline) that covers the entire time period. *ObjectB's* lifeline broadens into an activation rectangle when it is processing a call. During part of the time, it performs a recursive operation, as shown by the doubled activation rectangle between the call by *objectC* on *operationE* and the return of the result value. *ObjectC* is created and destroyed during the time shown on the diagram, so its lifeline does not span the whole diagram.

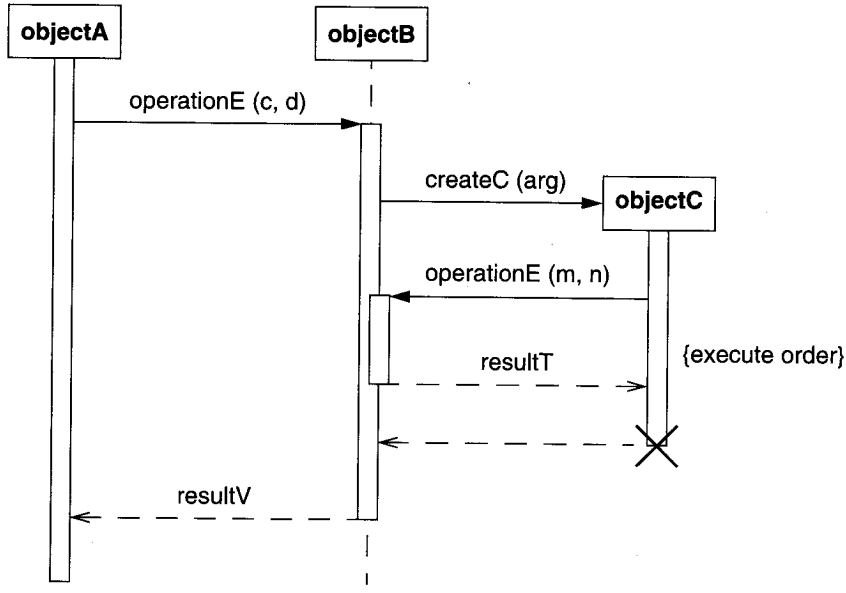

Figure 8.6 Sequence diagram with a transient object. Many applications have a mix of active and passive objects. They create and destroy objects.

The notation for a call is an arrow from the calling activation to the activation created by the call. The tail of the arrow is somewhere along the rectangle of the calling activation. The arrowhead aligns with the top of the rectangle of the newly created activation, because the call creates the activation. The filled arrowhead indicates a call (as opposed to the stick arrowhead for an asynchronous signal in Chapter 7).

The UML shows a return by a dashed arrow from the bottom of the called activation to the calling activation. Not all return arrows have result values—for example, the return from *objectC* to *objectB*. An activation, therefore, has a call arrow coming into its top and a return arrow leaving its bottom. In between, it may have arrows to and from subordinate activations

that it calls. You can suppress return arrows, because their location is implicit at the bottom of the activation, but for clarity it is better to show them.

If an object does not exist at the beginning of a sequence diagram, then it must be created during the sequence diagram. The UML shows creation by placing the object symbol at the head of the arrow for the call that creates the object. For example, the *createC* call creates *objectC*. The new object may or may not retain control after it is created. In the example, *objectC* does retain control, as shown by the activation rectangle that begins immediately below the object rectangle.

Similarly, a large 'X' marks the end of the life of an object that is destroyed during the sequence diagram. The 'X' is placed at the head of the call arrow that destroys the object. If the object destroys itself and returns control to another object, the 'X' is placed at the tail of the return arrow. In the example, *objectC* destroys itself and returns control to *objectB*. The lifeline of the object does not extend before its creation or after its destruction.

The UML shows a call to a second method on the same object (including a recursive call to the same method) with an arrow from the activation rectangle to the top of an additional rectangle superimposed on the first. For example, the second call to *operationE* on *objectB* is a recursive call nested within the first call to *operationE*. The second rectangle is shifted horizontally slightly so that both rectangles can be seen. The number of superimposed rectangles shows the number of activations of the same object.

You can also show conditionals on a sequence diagram, but this is more complex than we wish to include in this book. For further information, see [Rumbaugh-05].

8.2.3 Guidelines for Procedural Sequence Models

There are additional guidelines that apply to procedural sequence models beyond those mentioned in Chapter 7.

- **Active vs. passive objects**. Differentiate between active and passive objects. Most objects are passive and lack their own thread of control. By definition, active objects are always activated and have their own focus of control.
- **Advanced features**. Advanced features can show the implementation of sequence diagrams. Be selective in using these advanced features. Only show implementation details for difficult or especially important sequence diagrams.

8.3 Special Constructs for Activity Models

Activity diagrams have additional notation that is useful for large and complex applications.

8.3.1 Sending and Receiving Signals

Consider a workstation that is turned on. It goes through a boot sequence and then requests that the user log in. After entry of a name and password, the workstation queries the network to validate the user. Upon validation, the workstation then finishes its startup process. Figure 8.7 shows the corresponding activity diagram.

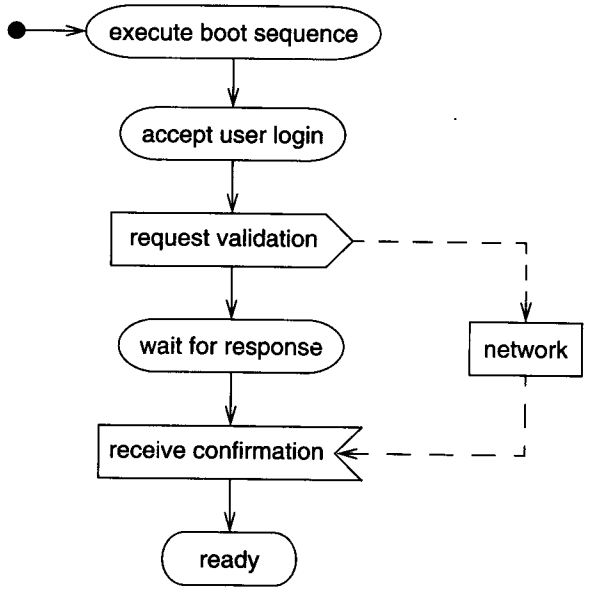

Figure 8.7 Activity diagram with signals. Activity diagrams can show fine control via sending and receiving events.

The UML shows the sending of a signal as a convex pentagon. When the preceding activity completes, the signal is sent, then the next activity is started. The UML shows the receiving of a signal as a concave pentagon. When the preceding activity completes, the receipt construct waits until the signal is received, then the next activity starts.

8.3.2 Swimlanes

In a business model, it is often useful to know which human organization is responsible for an activity. Sales, finance, marketing, and purchasing are examples of organizations. When the design of the system is complete, the activity will be assigned to a person, but at a high level it is sufficient to partition the activities among organizations.

You can show such a partitioning with an activity diagram by dividing it into columns and lines. Each column is called a ***swimlane*** by analogy to a swimming pool. Placing an activity within a particular swimlane indicates that it is performed by a person or persons within the organization. Lines across swimlane boundaries indicate interactions among different organizations, which must usually be treated with more care than interactions within an organization. The horizontal arrangement of swimlanes has no inherent meaning, although there may be situations in which the order has meaning.

Figure 8.8 shows a simple example for servicing an airplane. The flight attendants must clean the trash, the ground crew must add fuel, and catering must load food and drink before a plane is serviced and ready for its next flight.

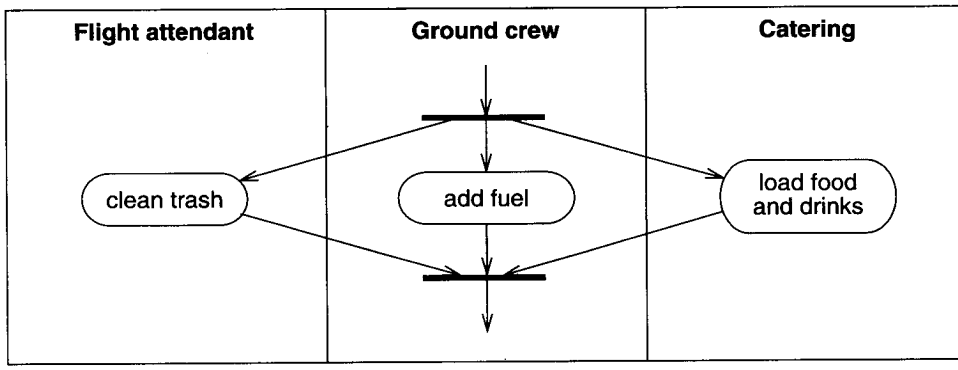

Figure 8.8 Activity diagram with swimlanes. Swimlanes can show organizational responsibility for activities.

8.3.3 *Object Flows*

Sometimes it is helpful to see the relationships between an operation and the objects that are its argument values or results. An activity diagram can show objects that are inputs to or outputs from the activities. An input or output arrow implies a control flow, therefore it is unnecessary to draw a control flow arrow where there is an object flow.

Frequently the same object goes through several states during the execution of an activity diagram. The same object may be an input to or an output from several activities, but on closer examination an activity usually produces or uses an object in a particular state. The UML shows an object value in a particular state by placing the state name in square brackets following the object name. If the objects have state names, the activity diagram shows both the flow of control and the progression of an object from state to state as activities act on it. In Figure 8.9 an airplane goes through several states as it leaves the gate, flies, and then lands again.

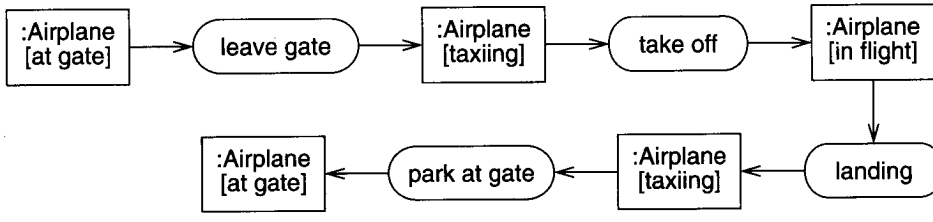

Figure 8.9 Activity diagram with object flows. An activity diagram can show the objects that are inputs or outputs of activities.

An activity diagram showing object flows among different object states has most of the advantages of a data flow diagram without most of their disadvantages. In particular, it unifies data flow and control flow, whereas data flow diagrams often separate them.

8.4 Chapter Summary

Independent use cases suffice for simple applications. However, it can be helpful to structure use cases for large applications using the include, extend, and generalization relationships. The include relationship incorporates one use case within the behavior sequence of another use case, like a subroutine call. The extend relationship adds incremental behavior to a base use case. Generalization can show specific variations on a general use case, analogous to generalization among classes. Don't use these relationships to excess. Remember that use cases are intended to be informal—use case relationships should only be used to structure major behavior units.

Sequence models are not only useful for fleshing out the interactions behind use cases, but they are also helpful for showing details of implementation. Not all objects in a sequence model need be active and exist for the entire computation. Some objects are passive and lack their own flow of control. Other objects are transient and may exist for only part of the duration of an operation.

Activity models also have additional notation that is helpful for large and complex applications. You can show fine controls via the sending and receiving of events that may interact with other objects that are not the focus of an activity diagram. You can augment activity diagrams with swimlanes to show the organizations that are responsible for different activities. And you can show the evolution of states of an object and how the states interleave with the flow of activities.

activation	passive object	use case extension
activity diagram	sequence diagram	use case generalization
focus of control	swimlane	use case inclusion
interaction model	transient object	
lifeline	use case	

Figure 8.10 Key concepts for Chapter 8

References

[Rumbaugh-05] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition.* Boston: Addison-Wesley, 2005.