Chapter 5 – Extending the Requirements Models

Table of Contents
- Chapter Overview
- Learning Objectives

Chapter Overview
This chapter has a lot of details. There are some sections that may require you to read and reread to understand them. And, you may not quite get it until you start to draw the diagrams. Hang in there and don’t give up. If you have questions please email me and be specific such as what question, page number, figure number, and attach your file as needed.

The main objective of defining requirements in system development is understanding users’ needs, how the business processes are carried out, and how the system will be used to support those business processes. Model building is one of the best ways for analysts to understand the business processes and to remember what they have learned from their fact finding activities.

This chapter explains four types of models and how they are used to capture the business rules and processes. Three of the models extend the definitions of the use cases, and the final model extends information about the classes.

Use case related models are the “fully developed use case description,” UML activity diagrams, and UML system sequence diagrams (SSD). The “fully developed use case descriptions” are used to document the context, purpose, description, conditions, and workflow of each use case. Activity diagrams are a graphical depiction of the use case workflow and are useful in illustrating the alternative paths through a business process. SSDs are used to document the inputs and outputs that are passed between the user and the system during a use case. Not only do these models document the internal steps of a use case, but the very act of developing these models force the analyst to ask detailed questions and help improve the understanding of the requirements.

The State machine diagram is a class related model. For some object classes in the domain model it is necessary to understand the life cycle of individual objects. This is especially true for business objects that can have different status conditions. A state machine diagram documents what these status conditions are and how an object changes status condition by transitioning from state to state.

Learning Objectives
After reading this chapter, you should be able to:
- Write fully developed use case descriptions
- Develop activity diagrams to model flow of activities
- Develop system sequence diagrams
- Develop state machine diagrams to model object behavior
- Explain how use case descriptions and UML diagrams work together to define functional requirements
Opening Case

Electronics Unlimited: Integrating the Supply Chain: This case is a short description of the need for businesses now to integrate their own systems, particularly the input/output interfaces with systems from other companies. To have a truly integrated supply chain, information must be passed between companies. The object-oriented models, as defined by UML, provide a strong basis to document interface requirements. Object-oriented system development provides the techniques to actually pass data between companies. (The other important technique is XML, which is not discussed in this case.)

End of Chapters Case

On the Spot Courier Services (running case): You will continue to work on this case study for your assignment. Here’s brief overview. Read it the full description in the back of the chapter. On the Spot is a small, but growing, courier service that needs to track customers, package pickups, package deliveries, and delivery routes. The description in this chapter extends the basic processes to include various types of real-time, online, and web-based capabilities. This chapter will require you to reuse the skills you developed in earlier chapters. You are then asked to create fully developed use case descriptions, activity diagrams, SSDs, and a state machine.

Use Case Descriptions

Key Terms

- use case description – a textual model that lists and describes the processing details for a use case
- scenarios or use case instances – unique sets of internal activities within use cases
- precondition – a condition that must be true before a use case begins
- postcondition – what must be true upon the successful completion of a use case

Foundation

A use case description lists and describes the processing details for a use case. Implied in all use cases is a person who uses the system. In UML, that person is called an actor, as shown on use case diagrams. Another way to think of an actor is as a role.

Internally, a use case includes a whole sequence of steps to complete a business process. Frequently, several variations of the business steps exist within a single use case. These different flows of activities are called scenarios or sometimes use case instances. Thus, a scenario is a unique set of internal activities within a use case and represents a unique path through the use case.

A use case description provides the documentation for the processing steps that are internal to a use case. There may be several descriptions, each one describing a different scenario. Depending on an analyst’s needs, use case descriptions tend to be written at two separate levels of detail: brief description and fully developed description. As shown later in the chapter, an activity diagram is also a useful technique to describe the internal processing of a use case.
**Brief Use Case Descriptions**

A brief description can be used for very simple use cases, especially when the system to be developed is a small, well-understood application. A simple use case would normally have a single scenario and very few—if any—exception conditions. Figure 5-1 illustrates several single sentence brief descriptions. A brief description does not document the internal processing since for simple use cases there are few steps.

**Fully Developed Use Case Descriptions**

The fully developed description is the most formal method for documenting a use case. The following items document each row of the example shown in Figure 5-2.

1. **Use case name**: a unique name for this use case
2. **Scenario**: the instance of the use case being documented
3. **Triggering Event**: the business event that initiates or triggers this use case
4. **Brief Description**: a one or two sentence description of the results of the use case
5. **Actors**: all those actors who use this use case
6. **Related use case**: any included use cases, or use cases that “include” this one
7. **Stakeholders**: other persons, other than the actors, who have an interest in the results or successful completion of this use case
8. **Precondition**: The state of the system before the use case begins, especially any required conditions of the database
9. **Postcondition**: the state of the system at the successful completion of the use case, especially any updates to the database
10. **Flow of activities**: a step by step sequence of the actions by the actor and the system internal to the use case
11. **Exception conditions**: any exception conditions that cause the system not to follow the expected flow of activities or that cause the use case to terminate abnormally

Usually the flow of activities is the most difficult part to develop, but also assists the analyst and the user to understand the requirements more deeply.

**Activity Diagrams for Use Cases**

**Key Terms**

Activity diagrams were introduced in Chapter 2 as a technique for documenting user workflows. See Chapter 2 for terms and complete examples. This chapter used less symbols, but then for your homework you use some symbols that were covered in Chapter 2 pages 57-60.

An activity diagram is an easily understood diagram to document the workflows of the business processes. Activity diagrams are a standard UML diagram that are also an effective technique to document the flow of activities within a use case. Activity diagrams are helpful when the flow of
activities for a use case is complex.

Remember that an activity diagram used rounded squares to identify individual activities within the workflow. It is also possible to include ovals, which represented included use cases, within a workflow. Figure 5-5 illustrates the Fill shopping cart use case, with three other use cases included as part of the total workflow.

The System Sequence Diagram – Identifying Inputs and Outputs

Key Terms

- **system sequence diagram (SSD)** – a diagram showing the sequence of messages between an external actor and the system during a use case or scenario
- **interaction diagram** – either a communication diagram or a sequence diagram that shows the interactions between objects
- **lifeline or object lifeline** – the vertical line under an object on a sequence diagram to show the passage of time for the object
- **loop frame** – notation on a sequence diagram showing repeating messages
- **true/false condition** – part of a message between objects that is evaluated prior to transmission to determine whether the message can be sent
- **opt frame** – notation on a sequence diagram showing optional messages
- **alt frame** – notation on a sequence diagram showing if-then-else logic

In the object-oriented approach, the flow of information is achieved through sending messages either to and from actors or back and forth between internal objects. A **system sequence diagram (SSD)** is used to describe this flow of information into and out of the automated system, e.g. between the actor and the system. Often a SSD is used in conjunction with an activity diagram to get a complete picture of the steps and information flows of a use case. An SSD is a type of interaction diagram, which are diagrams showing interactions between objects.

System Sequence Diagram (SSD) Notation

In a use case diagram, the actor “uses” the system, but the emphasis in an SSD is on how the actor “interacts” with the system by entering input data and receiving output data. Refer to figure 5-6 and 5-7 to note the basic notation used in an SSD. A stick figure represents an actor. The box labeled :System is an object that represents the entire automated system. (UMO object notation is similar to UML class notation, except the object name is often preceded by a colon and is underlined.)

Connected to the actor and the :System object are dashed lines, which represent **life lines** or **object lifeline**. Messages are passed between the actor and the :System and are assumed to be passed sequentially top to bottom along the life lines.

A message is documented with an arrow and a message descriptor. Since a SSD is a UML object-oriented model, the syntax for the messages is similar to interaction, e.g. programming methods, syntax that is found in various OO programming languages. Here is the notation for the message descriptor:
*true/false condition* return-value := message-name (parameter-list)

Any part of the message can be omitted. In brief, the notation components do the following:

- An asterisk (*) indicates repeating or looping of the message.
- Brackets [ ] indicate a true/false condition. This is a test for that message only. If it evaluates to true, the message is sent. If it evaluates to false, the message isn’t sent.
- Message-name is the description of the requested service. It is omitted on dashed-line return messages, which only show the return data parameters.
- Parameter-list (with parentheses on initiating messages and without parentheses on return messages) shows the data that are passed with the message.
- Return-value on the same line as the message (requires :=) is used to describe data being returned from the destination object to the source object in response to the message. A return value may also be shown as a separate return message on a dashed line.

To show a more complex flow of activities, **loop frames**, **opt frames**, and **alt frames** are used. The frame is a rectangle which encloses the messages that are part of the loop or opt or alt. The rectangle is labeled as a loop (looping set of messages), opt (optional messages), or alt (if-then-else messages).

**Developing a System Sequence Diagram (SSD)**

To develop an SSD, it is useful to have a detailed description of the use case—either in the fully developed form or as an activity diagram. One advantage of using activity diagrams is that it is easy to identify when an input or output occurs. Inputs and outputs occur whenever an arrow in an activity diagram goes from an external actor to the computer system.

Recall for a workflow diagram there are two swimlanes: the actor and the computer system. The system boundary coincides with the vertical line between the actor's swimlane and the computer system's swimlane. The development of an SSD based on an activity diagram falls into four steps:

1. Identify the input messages, which are the transition arrows that cross the system boundary.
2. Describe the message from the external actor to the system by using the message notation described earlier. The message name should describe the service requested from the system, such as *create a customer*. Include the parameters that the system will need to carry out the requested service.
3. Identify and add any special conditions such as iteration and true/false conditions.
4. Identify and add the outputs or return values from each message.

This sequence of steps two through four are done for each message identified in step one. One of the advantages of creating a SSD is that it helps the analyst understand and verify not only the processing steps of the use case, but the data values that are required.

Upon completion of this activity, the analyst will have a full description of a use case, with its workflow in an activity diagram, and good documentation of the inputs and outputs required. Model building is a powerful tool to help analysts understand user requirements. It also provides a comprehensive document that can be used in systems design and programming.
The State Machine Diagram – Identifying Object Behavior

Key Terms

- **state** – a condition during an object’s life when it satisfies some criterion, performs some action, or waits for an event
- **transition** – the movement of an object from one state to another state
- **state machine diagram** – a diagram showing the life of an object in states and transitions
- **pseudostate** – the starting point of a state machine diagram, indicated by a black dot
- **destination state** – for a particular transition, the state to which an object moves after the completion of a transition
- **origin state** – for a particular transition, the original state of an object from which the transition occurs
- **action-expression** – a description of the activities performed as part of a transition
- **guard-condition** – a true/false test to see whether a transition can fire
- **concurrency or concurrent state** – the condition of being in more than one state at a time
- **path** – a sequential set of connected states and transitions
- **composite state** – a state containing other states and transitions (that is, a path)

Foundation

Sometimes, it is important for a computer system to maintain information about the status of problem domain objects. The status condition for a real-world object is often referred to as the state of the object. A **state** of an object is a condition that occurs during its life when it satisfies some criterion, performs some action, or waits for an event. For real-world objects, the authors equate the state of an object with its status condition.

A state might have a name of a simple condition, such as On or In repair. Other states are more active, with names consisting of gerunds or verb phrases, such as *Being shipped* or *Working*. The name of a state shouldn’t be an object (or noun); it should be something that describes the object (or noun).

States are described as semi-permanent conditions because external events can interrupt a state and cause the object to go to a new state. An object remains in a state until some event causes it to move, or transition, to another state. A **transition**, then, is the movement of an object from one state to another state. Transitioning is the mechanism that causes an object to leave a state and change to a new state. States are semi-permanent because transitions interrupt them and cause them to end. Generally, transitions are short in duration—compared with states—and they can’t be interrupted.

A **state machine diagram** is composed of ovals representing the states of an object and arrows representing the transitions. After a transition begins, it runs to completion by taking the object to the new state, called the **destination state**. A transition begins with an arrow from an **origin state**—the state prior to the transition—to a destination state, and it is labeled with a string to describe the components of the transition.
The transition label consists of the following syntax with three components:

\[
\text{transition-name} \ (\text{parameters, ...}) \ [\text{guard-condition}] \ / \ \text{action-expression}
\]

any of which may be empty and which have the following meanings:

- **transition-name** – the trigger or event that causes the transition to fire
- **parameters** – any parameters that need to be passed to the object from the triggering object
- **guard-condition** – this “guards” the transition, or prohibits it from executing, even if the trigger fires, unless the guard condition is true
- **action-expression** – some actions that must be completed as part of the transition. The action expression may require the input parameters.

**Composite States and Concurrency**

In the real world, it is very common for an object to be in multiple states at the same time. The condition of being in more than one state at a time is called concurrency, or concurrent states. One way to show this is with a synchronization bar and concurrent paths, as in activity diagrams. Another way to show concurrent states is to have states nested inside other, higher-level states. These higher-level states are called composite states. A **composite state** represents a higher level of abstraction and can contain nested states and transition paths. See Figure 5-12.

The idea of composite states and concurrency can be extended one step further by allowing multiple paths within a composite state. Perhaps an object has entire sets of states and transitions—multiple paths—that are active concurrently. To document concurrent multiple paths for a single object, you can draw a composite state with the lower portion divided into multiple compartments—one for each concurrent path of behavior. See Figure 5-13.

**Rules for Developing State Machine Diagrams**

Usually, the primary challenge in building a state machine diagram is to identify the right states for the object. It is often helpful you do object think exercises and pretend to be the object itself, such as “I am an order.” “How do I come into existence?” and “What are my status conditions that need to be recorded?”

The other major area of difficulty for new analysts is to identify and handle composite states with nested threads. Usually, the primary cause of this difficulty is a lack of experience in thinking about concurrent behavior. The best solution is to remember that developing state machine diagrams is an iterative behavior—more so than developing any other type of diagram. Analysts seldom get a state machine diagram right the first time. They always draw it and then refine it again and again.

Finally, don’t forget to ask about an exception condition—especially when there are the words verify or check. Usually, there will be two transitions out of states that verify something: one for acceptance and one for rejection.

Here is a list of steps that will help in developing state machine diagrams:

1. Review the class diagram and select the classes that might require state machine diagrams.
2. For each selected class in the group, make a list of all the status conditions that can be identified
3. Begin building state machine diagram fragments by identifying the transitions that cause an object to leave the identified state

4. Sequence these state-transition combinations in the correct order

5. Review the paths and look for independent, concurrent paths

6. Look for additional transitions

7. Expand each transition with the appropriate message event, guard-condition, and action-expression

8. Review and test each state machine diagram
   ◦ Make sure states apply to that – the correct – object class
   ◦ Trace the entire life cycle of the object to make sure you have all states
   ◦ Handle all exception conditions as well as normal paths
   ◦ Double check for concurrent paths

Developing RMO State Machine Diagrams

This section contains two examples of how to develop a state machine. Each state machine diagram is for a single class. For these examples the Sale and SaleItem classes are chosen. First a state machine diagram for SaleItem is developed and then a state machine diagram for Sale is developed.

The example to create the SaleItem state machine diagram follows the eight step program as identified above. Figure 5-14 shows down through step three with the states and exit transitions identified. Figure 5-15 completes step four with a preliminary partial state machine that includes the states and transitions in the correct order. Notice that it is not complete. There are not start and end points. Additional transitions, such as for error conditions or backward transitions, have not yet been identified. Figure 5-16 is the completed state transition machine. The beginning and ending transitions have been added. Transitions to handle back ordered situations have been added, along with an action expression to make a purchase order. This state transition machine is straightforward with no need to have concurrent states or complex paths.

The example for the Sale is more complex. A Sale has multiple items and has a more elaborate life cycle. Since there are multiple SaleItems associated with a sale, its status condition needs to reflect things such as partial shipments and waiting for back orders for some items while others are being shipped. In Figure 5-17 there are 6 states with their appropriate exit transitions identified. Figure 5-18 is the initial state machine diagram which organizes those states and transitions into a diagram. However, two states appear to have some overlap (In shipping and Waiting for back orders) and in fact appear to be concurrent states. The methodology for developing state machine diagrams does not have any algorithm of how to handle these complex situations. The analyst must analyze it and determine the best solution. There are various ways that this concurrent condition can be fixed. In this example, it is decided to create a new state of Being shipped and to promote In shipping to a high-level composite state. Figure 5-19 shows the solution. The state machine diagram should then be tested by walking through each state and transition to see if it models real world behavior correctly, e.g. apply step 8 listed above. In this situation, it appears to work correctly with a Sale remaining in the In
shipping state until all of the individual SaleItems have been shipped.

The benefit of developing a state machine diagram for a problem domain object is that it helps you capture and clarify business rules. As always, the benefits of careful model building help us gain a true understanding of the system requirements.

Integrating Requirements Models

Key Terms

none

At this point, you have learned how to create several requirements models. One of the key concepts for you to understand is that all of these models link together to provide a complete picture of the user's requirements. And, it is extremely important that the models are consistent with each other. Information across all of these models must be consistent and provide a unified picture of the requirements.

Two diagrams are especially important because they provide an overview or comprehensive view of the entire system. One is the use case diagram, which in its complete form identifies all of the use cases to be implemented. The other is the class diagram or the domain model, which provides information about all of the data items required.

As shown in Figure 5-20, other diagrams feed off of these two diagrams. The Use case diagram feeds into the Use case descriptions, Activity diagrams, and SSDs, all of which document information about individual use cases. The State machine diagram and to some extent the SSD are fed by the class diagram. A state machine diagram covers only one class, where the SSD may refer to many different objects or attributes as parameters in the messages.

Troubleshooting Tips

Fully Developed Use Case Description

Sometimes you may have difficulty understanding the difference between a use case and a scenario. A use case is the generalized version, and a scenario is a particular instance of it. The best way to explain this concept is with an example. A generalized use case of Purchase merchandise is different for shopping online than it is for shopping in a store. The online version has many more computer system related steps to find merchandise, show images of the merchandise, add it to the shopping cart, and then checking out. The store version does not involve the computer system until it is time to check out. So even though the general use case is the same, the internal processing flow is very different. Hence, each version must be documented separately.

Also sometimes you may have a hard time understanding pre-conditions and post-conditions. Almost all pre-conditions and post-conditions have to do with the data in the database, its availability, and its condition. Again, the best way to teach it is through examples. In order for the Purchase merchandise use case to even begin, there must be merchandise available. Depending on the user’s definition, a customer data record may also need to be in existence. These are pre-conditions.

After the use case has finished executing, then there must also be a reduction in merchandise inventory, and there must be a purchase object in a particular status (such as ready for shipping). Depending on the requirements there must either be a payment object, or perhaps an increase in the customer account
value.
The Flow of Activities is also sometimes confusing. The idea is simply that the user does something, the system responds. It goes back and forth. For some of you, especially those that are visually oriented, it may be easier to develop an activity diagram rather than the Flow of Activities. Both essentially capture the same information.

**Activity Diagrams**

You have already used activity diagrams in documenting user workflows. So hopefully the activity diagrams will be fairly easy concepts for you to grasp. The two swimlanes, between the user and the system, keep the activity diagram simple.

Probably the major area of difficulty is how detailed to make the activity ovals. One extreme has the ovals almost at the keystroke level, while the other extreme can extend to the entire use case. First, you need to remember the “Prefect technology assumption” and try to remove those activities. Then each activity should be a logical action, such as “search for a product” or at a little lower level, “enter product information for search.” The activities need to be at a low enough level to show the interactions between the user and the system.

One potential danger to avoid is to keep the examples and problems simple. In today's Web based systems with hotlinks and hyperlinks in abundance, the users can take a multitude of alternative paths through a process. Be sure to keep your examples focused, such as only to find merchandise and buy it. Avoid all of the other things that users can do to bounce around the entire site.

**System Sequence Diagram**

Sequence diagrams are one of the more difficult UML models to learn. Sequence diagrams are a powerful design model that is taught in depth in Chapter 11. In order to facilitate learning and help you understand sequence diagrams, the authors introduce sequence diagrams with the more simplified system sequence diagram (SSD) version. However, an SSD is a powerful analytical model in its own right and provides a good insight into use case internals.

There are many parts to an SSD for you to learn. Be sure to take the time so that they understand the concept of a lifeline (simply the life of an object, which is read top to bottom), and a message (from a programming viewpoint it is the invoking of a method, or a method call, with data parameters being passed). The objective of the SSD is to document the actual input data and the output data that flows between the user actor and the system. The easiest way to identify what messages are required is to use the activity diagram of the use case. Every time an arrow crosses the center boundary line, there is an input or output message.

**State machine diagram**

This model can also be somewhat difficult for you. The important point you should grasp is that a state machine diagram describes the life of a single object in an object class. (Of course all objects in that class will have the same behavior.)

Probably the key to creating a state machine diagram is to be able to identify the correct states. One good way is for you to do the object think exercise mentioned earlier. Another way is to think about all
of the status conditions for an object, particularly those status conditions that are of interest to the outside world. That is a good place to start to identify states.

The other major difficulty is to identify concurrent states, where an object is doing two things at the same time. Modeling concurrent states can get rather complex, either as parallel paths or as composite states. You may prefer parallel paths, while other analysts will prefer composite states. Either will work. There does not seem to be an easy way for students to be able to do this successfully, other than practicing. The only way to learn and be good at creating this diagram is with practice and experience and even experienced analysts will do several iterations and have other analysts proof their drawing.

Again email if you have questions or need help.