UML Tutorial: Sequence Diagrams.

Robert C. Martin

Engineering Notebook Column

April, 98

In my last column, I described UML Collaboration diagrams. Collaboration diagrams allow the designer to specify the sequence of messages sent between objects in a collaboration. The style of the diagram emphasizes the relationships between the objects as opposed to the sequence of the messages.

In this column we will be discussing UML Sequence diagrams. Sequence diagrams contain the same information as Collaboration diagrams, but emphasize the sequence of the messages instead of the relationships between the objects.

The Cellular Phone Revisited

Here again is the final collaboration diagram from last column's Cellular Phone example. (See Figure 1.)

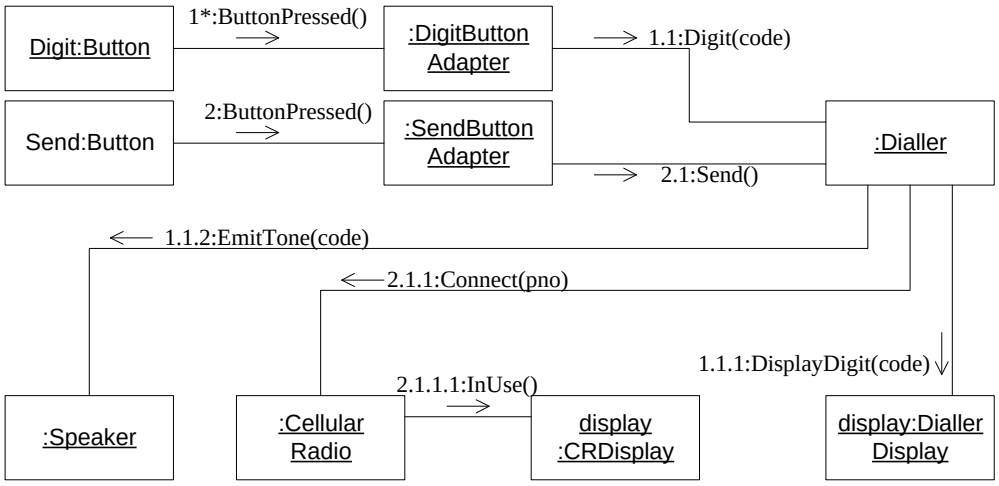

Figure 1: Collaboration diagram of Cellular Phone.

The Sequence diagram that corresponds to this model is shown in Figure 2. It is pretty easy to see what the diagrams in Figure 2 are telling us, especially when we compare them to Figure 1. Let's walk through the features.

First of all, there are two sequence diagrams present. The first one captures the course of events that takes place when a digit button is pressed. The second captures what happens when the user pushes the 'send' button in order to make a call. At the top of each diagram we see the rectangles that represent objects. As in collaboration diagrams, the names of the objects are underlined to distinguish them from classes. Also the object name is separated from the class name by a colon. Some objects, like the Dialer, have no particular object name, so the colon precedes the class name without an object name in front of it.

Descending from each object is a dashed line known as the "lifeline". These lines define the time axis of the diagram. By convention, time proceeds in a downward direction[1]. The lifelines depict how long the

[1] UML also allows the diagram to be tipped on its side so that time proceeds to the right. However this is a very uncommon variation.

objects that they are connected to are in existence. In Figure 2, we cannot see the full extent of the lifelines; they extend from the very top of the diagram to the very bottom. This implies that the objects portrayed in this diagram are in existence before the start of time in the diagram and remain in existence beyond the end of the diagram.

The arrows between the lifelines represent messages being sent between the objects. Sequence numbers, as shown in the previous column, are permitted but are not necessary. The white narrow rectangles that the arrows terminate on are called activations. They show the duration of the execution of a method in response to a message. The methods implicitly return to their caller at the end of the activation.[2]

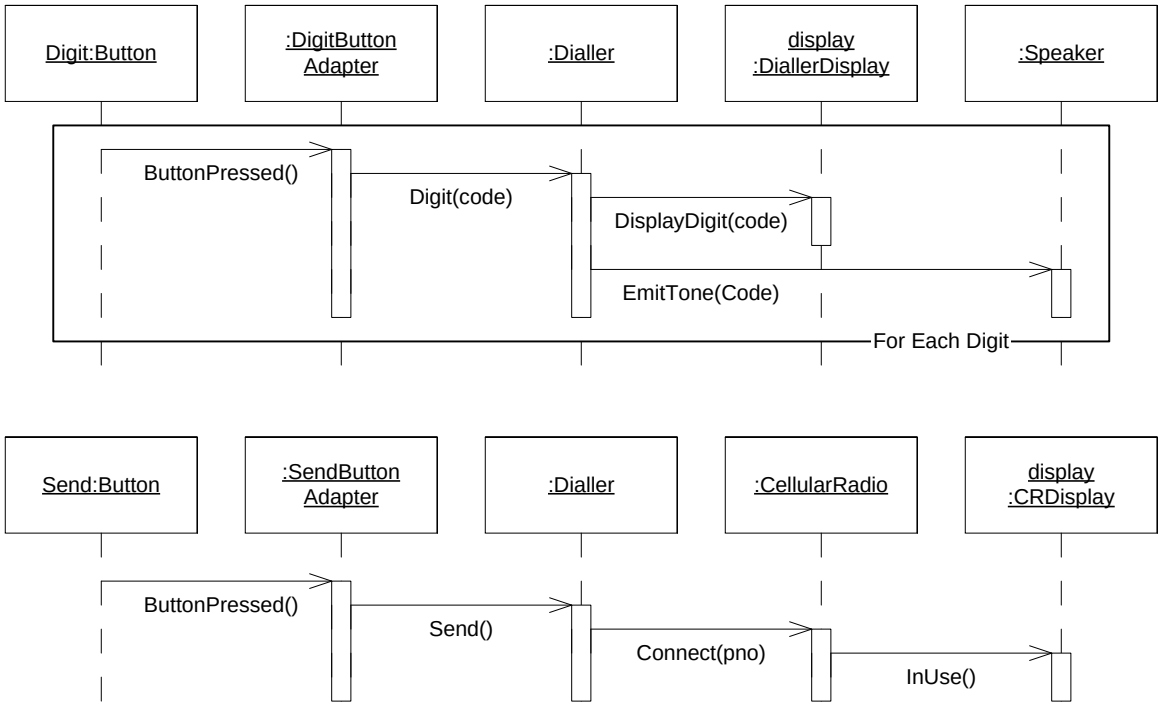

Figure 2: Sequence Diagram of Cellular Phone.

The large rectangle that encloses the group of messages in the first sequence diagram defines an iteration. The looping condition for that iteration is shown at the bottom of the rectangle.

Take a little time to inspect the two figures. You should be able to prove to yourself that they represent precisely the same information. Yet the two forms are radically different. The Sequence diagrams take up a bit more space, but are much easier to follow algorithmically. The Collaboration diagram shows the whole collaboration in one dense diagram, but obscures the algorithm to some extent. Which of the two you use depends upon the information you are trying to emphasize. Sometimes you really want to show the cohesiveness of a collaboration; other times you want to show the flow of the algorithm.

Creation and Deletion of Objects

Figure 3 shows how we depict the creation and deletion of objects on a sequence diagram. Here we see the CellularRadio object creating a Connection object in response to a Connect message. Creation is denoted

[2] Such returns can be shown by an unlabeled arrow that extends from the bottom of the activation back to the lifeline of the calling object. In the case of asynchronous messages (described later), the end of an activation does not imply a return.

by a message arrow that terminates on the object rectangle. Deletion is likewise denoted by a message arrow that terminates on an X at the end of the object's lifeline.

This notation is very intuitive. The lifetime of the Connection object is plain for all to see. Such things are much harder to depict clearly in a Collaboration diagram.

Asynchronous Messages and Concurrency

Notice that some of the arrowheads on Figure 3 are incomplete. These "half-arrowheads" denote asynchronous messages. An asynchronous message is a message that returns immediately after spawning a new thread in the receiving object[3]. The Connect message, for instance, returns immediately to the CellularRadio object. Yet you can see by the activation rectangle on the Connection lifeline that the Connect method continues to execute. The Connect method is executing in a separate thread. This demonstrates the power that sequence diagrams have for showing concurrent multi-threaded interactions. Depicting this kind of information on a collaboration diagram is clumsy at best.

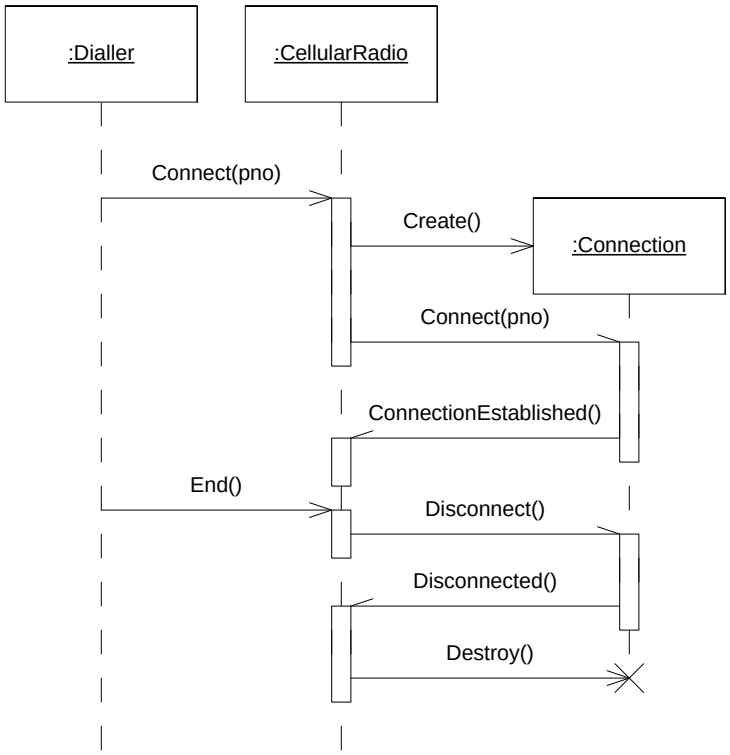

Figure 3: Connecting and Disconnecting

Race Conditions

Wherever concurrency is present, race conditions are possible. Race conditions occur when a single thread or object receives messages from two competing sources. If not handled properly, the participating objects can get quite confused. Consider Figure 4.

---

[3] It is not actually necessary that a new thread be spawned, all that is required is that the receiving object execute the method in a separate thread. That thread could be in existence prior to the sending of the asynchronous message, and just waiting for something to do.

Here again we see two sequence diagrams. In these diagrams the activation rectangles have been omitted for clarity[4]. The top diagram shows the normal course of events when the cellular phone receives a call. The CellularRadio object detects the incoming call and sounds the ringer. It also tells the Dialler object that a call is coming in. This puts the Dialler in a special state. When the Send button is pushed while the Dialler is in this special state, the Dialler sends the Answer message to the CellularRadio; thus connecting the call.

Why the special state? Look back at Figure 1. You can see that the Dialler can receive the Send message after the user has entered a phone number. Thus there are two different circumstances under which the Dialler will receive the Send message. One when calling, and the other when answering. When calling, the Dialler sends the Connect(pno) message to the CellularRadio. When answering the Dialler sends the Answer message to the CellularRadio.

The second diagram shows the race condition. In this case the user is actually in the process of dialing a phone number. A call comes in to the CellularRadio object just as the user pushes the Send button to dial the outgoing call. The IncommingCall message and the Connect(pno) message cross paths. The crossing of the messages shows the race.

When a message is angled downwards as these two are, it shows that time can pass between the sending of the message and its reception. In systems that pass messages over networks it is pretty clear that messages spend time in transmission, or waiting in a queue before being received. In multithreaded applications, messages between objects are often converted into message objects that wait on queues. Thus, significant time can pass between the sending (enqueueing) of those messages and their eventual reception and execution.

Clearly, if the CellularRadio object has not been written to expect this turn of events, it is going to get confused. It is possible that the engineers who designed the state machine of the CellularRadio class might not have anticipated that a Connect(pno) message would be received after it had just sent an IncommingCall message.

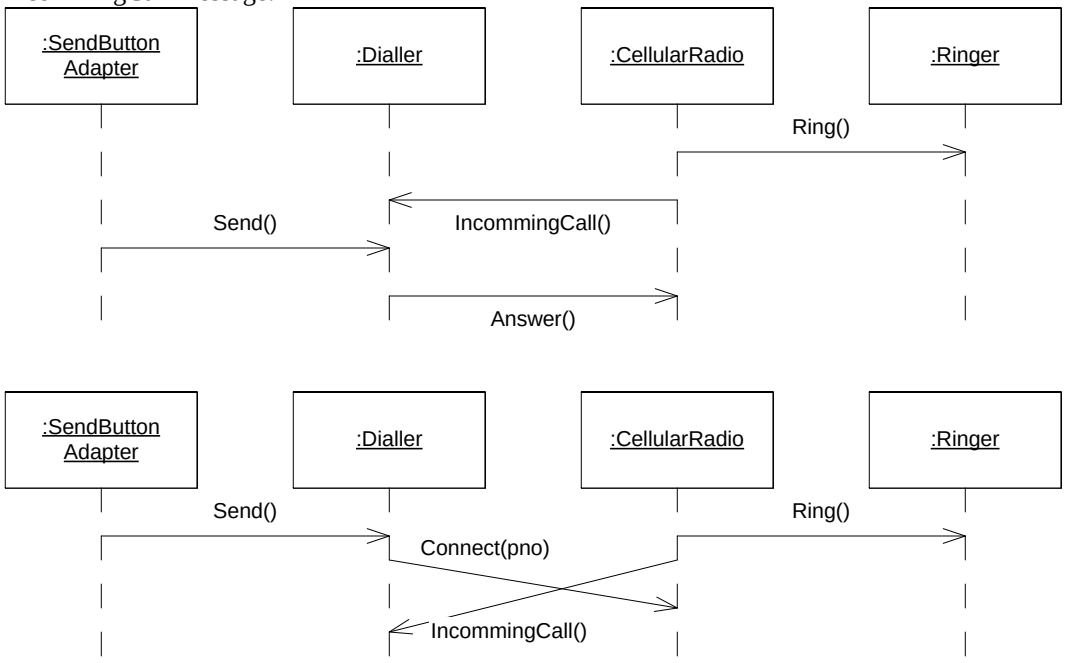

Figure 4: Race condition between dialing and answering.

[4] UML allows such omissions at the designer's discretion.

Race conditions like this are a significant source of errors in concurrent systems. Debugging them can be remarkably difficult since they are critically dependent upon the timing of several events. Such problems lead to the "intermittent unexplainable unrepeatable crash" that is the bane of all real-time software engineers. Sequence charts with angled arrows [5]are one of the best tools we have for identifying where race conditions can occur.

Conclusion

It is clear that sequence charts have a number of very powerful advantages. They clearly depict the sequence of events, show when objects are created and destroyed, are excellent at depicting concurrent operations, and are invaluable for hunting down race conditions. However, with all their advantages, they are not perfect tools. They take up a lot of space, and do not present the interrelationships between the collaborating objects very well. With all the power that sequence diagrams have to offer, I still find the density and elegance of collaboration diagrams more pleasing. Therefore, as a matter of personal taste, I use sequence diagrams only when their particular powers are required.

In subsequent columns we will be investigating, among other things, the UML notation for finite state machines, Activity diagrams, Domain models, Use Cases, Packages, and Components.

[5] Such charts are often referred to as "Message Sequence Charts"