UML Tutorial: Collaboration Diagrams

Robert C. Martin

Engineering Notebook Column

Nov/Dec, 97

In this column we will explore UML collaboration diagrams. We will investigate how they are drawn, how they are used, and how they interact with UML class diagrams.

UML 1.1

On the first of September, the three amigos (Grady Booch, Jim Rumbaugh, and Ivar Jacobson) released the UML 1.1 documents. These are the documents that have been submitted to the OMG for approval. If all goes well, the OMG will adopt UML by the end of this year.

The differences between the UML 1.0 and UML 1.1 notation are minimal. The previous article in this series, (September issue) has not been affected by the changes.

Dynamic models

There are three kinds of diagrams in UML that depict dynamic models. State diagrams describe how a system responds to events in a manner that is dependent upon its state. Systems that have a fixed number of states, and that respond to a fixed set of events are called finite state machines (FSM). UML has a rich set of notational tools for describing finite state machines. We'll be investigating them in another column.

The other two kinds of dynamic diagram fall into a category called Interaction diagrams. They both describe the flow of messages between objects. However, sequence diagrams focus on the order in which the messages are sent. They are very useful for describing the procedural flow through many objects. They are also quite useful for finding race conditions in concurrent systems. Collaboration diagram, on the other hand, focus upon the relationships between the objects. They are very useful for visualizing the way several objects collaborate to get a job done and for comparing a dynamic model with a static model

. Collaboration and sequence diagrams describe the same information, and can be transformed into one another without difficulty. The choice between the two depends upon what the designer wants to make visually apparent.

Sequence diagrams will be discussed in a future article. In this article we will be concentrating upon collaboration diagrams.

The interplay between static and dynamic models.

There is a tendency among novice OO designers to put too much emphasis upon static models. Static models depicting classes, inheritance relationships, and aggregation relationships are often the first diagrams that novice engineers think to create. Disastrously, they are sometimes the *only* diagrams that they create.

In fact, a static emphasis on object oriented design is inappropriate. Software design is about behavior; and behavior is dynamic. Object oriented design is a technique used to separate and encapsulate behaviors. Therefore an emphasis upon dynamic models is very important.

More important, however, is the interplay that exists between the static and dynamic models. A static model cannot be proven accurate without associated dynamic models. Dynamic models, on the other hand, do not adequately represent considerations of structure and dependency management. Thus, the designer must iterate between the two kinds of models, driving them to converge on an acceptable solution.

Example: A Cellular Phone.

Consider the software that controls a very simple cellular telephone. Such a phone has buttons for dialing digits, and a "send" button for initiating a call. It has "dialer" hardware and software that gathers the digits to be dialed and emits the appropriate tones. It has a cellular radio that deals with the connection to the cellular network. It has a microphone, a speaker, and a display.

From this simple spec, we might be tempted to create a static model as shown in Figure 1.

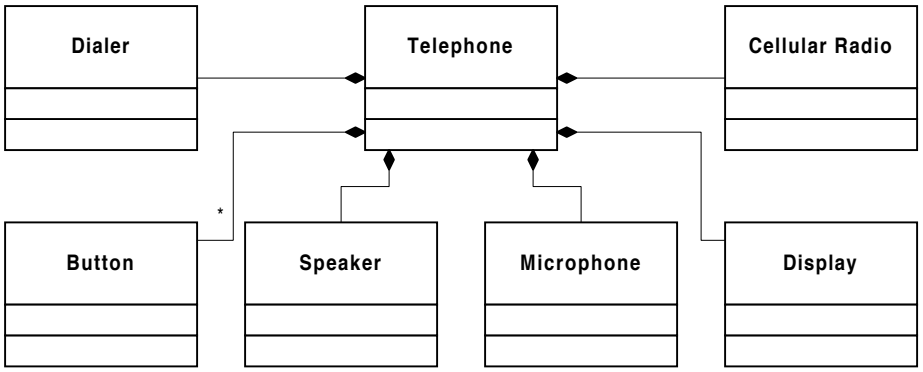

Figure 1: Static model of a Cellular Phone

It is very hard to argue with this static model. The composition relationships reflect, very clearly, the specification above. Indeed, the telephone "has" all the listed components. But is this the correct static model? How would be know?

One criterion is to compare the static model to the real world. Certainly Figure 1 passes this test. In the real world, a cellular phone "has" all the components shown above. However, experienced object oriented designers know that, while this test is essential, it is not sufficient. Figure 1 does not show the *only* static model that matches the real world of the cellular telephone. In order to choose between the many possible static models a more sensitive test is needed.

Specifying Dynamics.

How does the cellular phone work? To keep things simple, lets just look at how a customer might make a phone call. The use case for this interaction looks like this:

```
Use case: Make Phone Call
1. User presses the digit buttons to enter the phone number.
2. For each digit, the display is updated to add the digit to the
   phone number.
3. For each digit, the dialer generates the corresponding tone and
   emits it from the speaker.
4. User presses "Send"
5. The "in use" indicator is illuminated on the display
6. The cellular radio establishes a connection to the network.
7. The accumulated digits are sent to the network.
8. The connection is made to the called party.
```

This is simplistic, but adequate for our purposes. The use case makes it clear that there is a procedure involved with making a call. How do the objects in the static model collaborate to execute this procedure?

Let's trace the process one step at a time. The first thing that happens when this use case is initiated is that the user presses a digit button to begin entering the phone number. How does the software in the phone know that a button has been pushed?

There are a variety of ways that this can be accomplished; but they can all be simplified to having a `Button` object that sends a `digit` message. Which object should receive the digit message? It seems clear that it should be the `Dialer`. The `Dialer` must then tell the `Display` to show the new digit, and must tell the `Speaker` to emit the appropriate tone. The `Dialer` must also remember the digits in the list that accumulates the phone number. Each new button press follows the same procedure until the "Send" button is pressed.

When the "Send" button is pressed, the appropriate `Button` object sends the `Send` message to the `Dialer`. The `Dialer` then sends a `Connect` message to the `CellularRadio` and passes along the accumulated phone number. The `CellularRadio` then tells the `Display` to illuminate the "In Use" indicator.

This simple procedure is depicted in the collaboration diagram in Figure 2.

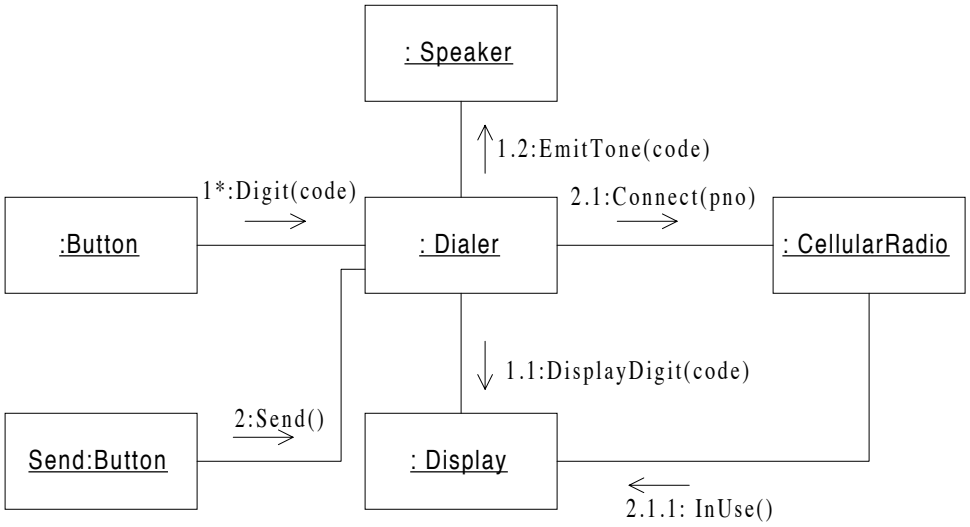

Figure 2: Collaboration Diagram of "Make Phone Call" use case.

Syntax

First, let's look at the syntax of the diagram above. The rectangles in this diagram depict objects, not classes. You can tell that they are objects because they are underlined. In UML, something that is underlined is an *instance*; whereas something that is not underlined is a template from which an instance can be created. Notice the object on the lower left entitled `Send:Button`. In UML the full name of an object is a composite that includes the name of its class. The name of the object and the name of the class are separated by a colon. Notice that all the other objects in the diagram are anonymous; they have no name, and their class is shown with a colon in front.

The lines connecting the objects are called links. Links are *instances* of associations. You are not allowed to create a link on a collaboration diagram if there is no corresponding association, (or aggregation, or composition) on a class diagram. Remember this rule, we'll come back to it later.

The arrows represent messages; and are labeled with their names, sequence numbers, and arguments. The name of a message corresponds to the name of a member function. That member function must exist in the class that the receiving object is instantiated from. The sequence numbers show the order in which the messages occur. The sequence numbers are nested so that you can tell which messages are sent from within other messages.

For example, message `2:Send()` is sent to the `Dialer`. As a result the `Dialer` begins executing a member function. Before that function returns, it sends message `2.1:Connect(pno)` to the

`CellularRadio`. The `CellularRadio` then sends message `2.1.1:InUse()` to the `Display`. Thus, the dot structure of the sequence numbers make it easy to see the procedural nesting of the messages.

Message 1*:Digit(code) has an asterisk in order to denote that it may occur many times before message 2. UML defines way to use this syntax to represent loops and conditions that are beyond the scope of this article. We'll come back to such details in a subsequent article.

Reconciling the static model with the dynamic model.

It should be clear that the structure of the objects in the dynamic model (Fig 1) does not look very much like the structure of the classes in the static model (Fig 2). Yet by the rule we talked about above, a link between objects must be represented by a relationship between the classes. Thus, we have a problem.

The problem could be that our dynamic model is incorrect. Perhaps we should force the dynamic model to look like the static model. However, consider what such a dynamic model would look like. There would be a `Telephone` object in the middle that would receive messages from the other objects. The `Telephone` object would respond to each incoming message with outgoing messages of its own.

Such a design would be highly coupled. The `Telephone` object would be the master controller. It would know about all the other objects, and all the other objects would know about it. It would contain all the intelligence in the system, and all the other objects would be correspondingly stupid. This is not desirable because such a "god" object becomes highly interconnected. When any part of it changes, other parts of it may break.

I prefer the dynamic model shown in Figure 2. The concerns are decentralized in a reasonable fashion. Each object has its own little bit of intelligence, an no particular object is in charge of everything. Changes to one part of the model do not necessarily ripple to other parts.

But this means that our static model is inappropriate. It should be changed to look like Figure 3. Notice that I have demoted all the composition relationships to associations. This is because none of the connected classes share a whole/part relationship with the others. The `Dialer` is not part of the `Button` class, The `CellularRadio` is not part of the `Dialer`, etc. Notice also that I have specified the direction of navigation. The dynamic model makes it very clear which class needs to navigate to which other classes. I have also added the member functions into the class icons; again because the dynamic model made them so apparent.

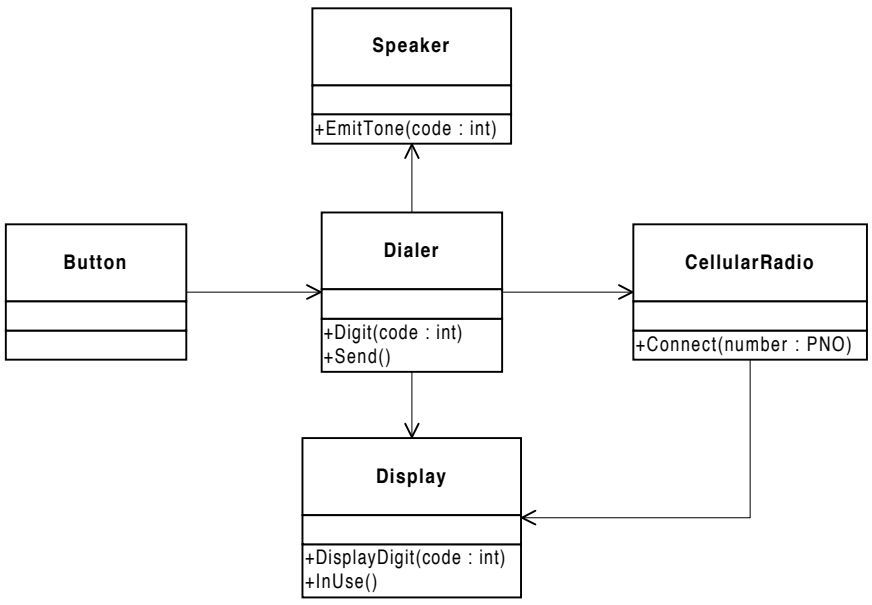

Figure 3: Reconciled static model

You might feel uncomfortable with this static model because it does not seem to reflect the real world as well as the first. After all, we have lost the notion of the telephone containing the buttons and the display, etc. But that notion was based upon the *physical* components of the telephone, not upon its behavior. Indeed the new static model is based upon the real world behavior of the telephone rather than upon its real world physical makeup.

We also lost a few classes. The `Telephone` and `Microphone` classes played no part in the dynamic model, and so have been removed. It may be that some other dynamic scenario will require them. If that happens, then we will put them back.

This points out the fact that many dynamic models usually accompany a single static model. Each dynamic model explores a different variation of a use case, scenario, or requirement. The links between the objects in those dynamic models imply a set of associations that must be present in a static model. Thus, dynamic models tend to vastly outnumber static models.

Scrutinizing the static model

Our static model has a few problems. For example, why should a class named `Button` know anything about a class named `Dialer`? Shouldn't the `Button` class be reusable in programs that don't have `Dialers`?

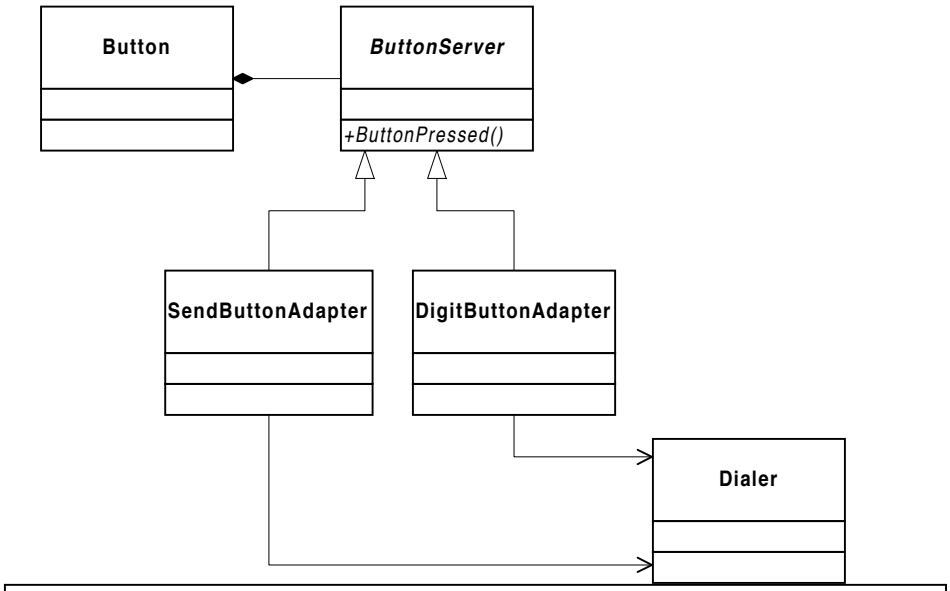

Figure 4: ADAPTED SERVER pattern for decoupling Button from Dialer

We can solve this problem by employing the ADAPTED SERVER pattern as shown in Figure 4. Now the `Button` class is completely reusable. Any other class that needs to detect a button press simply derives from `ButtonServer` and implements the pure virtual `ButtonPressed` function. If, like `Dialer`, the class must detect many different `Button` objects, ADAPTERS can be used to catch the `ButtonPressed` messages and translate them.

Another problem with the static model in Figure 3 is the high coupling of the Display class. This class will be the target of an association from many different clients. Those of you who recall my column entitled "The Interface Segregation Principle" (ISP) *[C++ Report, Aug, 1996. This document is also available in the 'publications' section of http://www.oma.com]* will understand that an unwarranted dependency exists between the `CellularRadio` class and the `Dialer` class. If one of the methods of `Display` needs to

be altered because of the needs of `Dialer`, then `CellularRadio` will be affected; at very least, by an unwarranted recompile.

To solve this problem, we can segregate the interfaces of the Display class as shown in Figure 5.

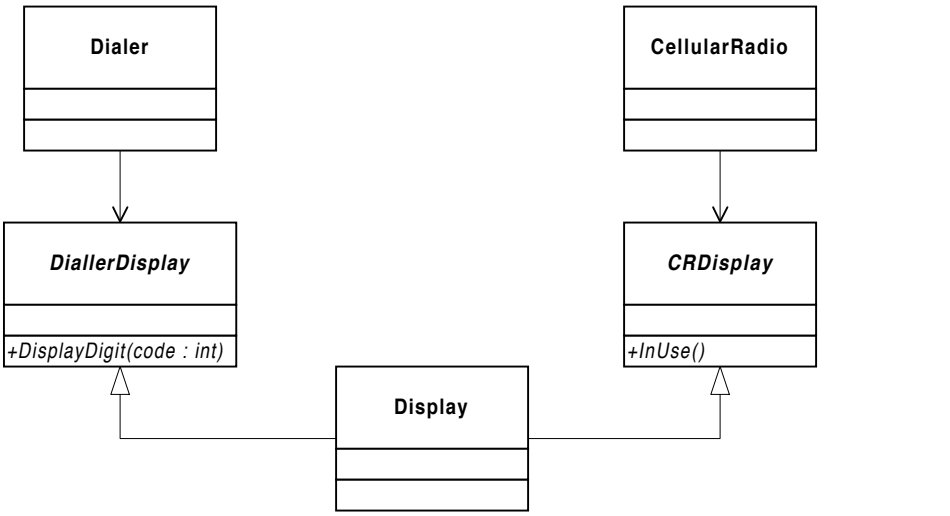

Figure 5: Interface Segregation of the Display

Iterating the dynamic model

Clearly these perturbation will have an effect upon the dynamic model. Thus, it will have to be changed as shown in Figure 6. This model shows how the adapters translate the `ButtonPressed` messages into something that the `Dialer` can understand. It also shows the segregation of the display interfaces. Note that the object named display appears twice, but with different class names. This indicates that display is derived from more than one class. Thus, the class name of the object tells the reader which interface the sender is depending upon.

Conclusion

We have completed two iterations of our static and dynamic model of a cellular phone. The first iteration was simply a guess. In the case of the first static model, the guess was pretty bad. However, after the second iteration we had resolved the disparity between the two models and had begun to explore more subtle design issues. In a real project, this iteration would continue until the designer was satisfied that both models were appropriately tuned.

Static models are necessary but insufficient for complete object oriented designs. A static model that is produced without the benefit of dynamic analysis is bound to be incorrect. The appropriate static relationships are a result of the dynamic needs of the application. UML Collaboration diagrams are a good way to depict dynamic models and compare them to the static models that must support them.

In future columns, we will continue to explore the wiles of UML. Among other things we will discuss UML's rich notation for finite state machines. We will explore how race conditions in concurrent systems can be detected with sequence diagrams. And we will demonstrate the facilities within UML that allow it to be extended.

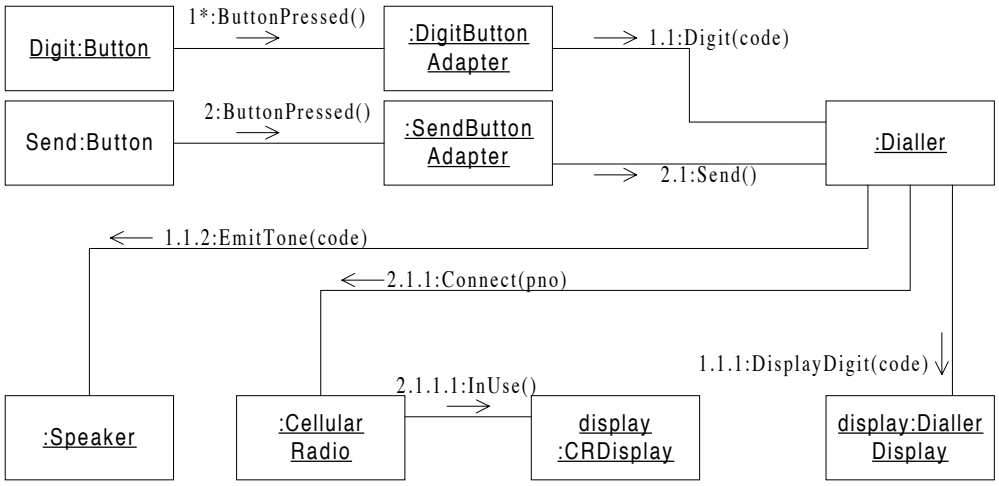

Figure 6: Iterated Dynamic Model