Java Modeling: A UML workbook, Part 1

Introduction to sequence diagrams

Granville Miller

May 08, 2001

In this first installment of his new column, Granville Miller introduces one of the building blocks of the Unified Modeling Language: sequence diagramming. Sequence diagrams are used throughout the design process to demonstrate the internal interactions between actors and objects as a system executes over time. Follow along with Granville as he creates one of these diagrams, using a loan processing application as his example.

The Unified Modeling Language (UML) is a standard notation for modeling object-oriented systems. Introduced to the object-oriented programming community in stages between 1995 and 1997, UML was approved by the Object Management Group (OMG) in late 1997. Though it was controversial upon inception -- it was introduced amidst a flurry of protest and counterproposals -- UML has since become the industry standard for system notation. UML is now in version 1.4 and continues to evolve to meet the needs of object-oriented developers. (For more about the history of UML, see Resources.)

UML can be complex to learn, mainly because it attempts to provide modeling notation for such a broad array of situations. Each modeling notation takes the form of a diagram, and there are currently nine diagrams within the UML specification. Fortunately, learning UML can be a staged process; you can learn just one diagram at a time, and you do not have to embrace the full complexity of a diagram in your first attempt.

In this column, I'll step you through UML design and notation for Java-based application development. I'll introduce the essentials of the UML framework and other modeling technologies in a logical (and hopefully enjoyable) fashion, and you'll learn hands-on by modeling examples from the real world. In this first installment, we'll start with sequence diagramming, using a loan processing application as our example. Please note that I assume you are familiar with the Java language and have a basic knowledge of object-oriented development methods and terminology. Object-oriented concepts will be briefly explained, but in-depth discussion is beyond the scope of this column.
About sequence diagrams

UML does not proscribe any particular software development method or process; it merely standardizes the form of notation. Numerous development methods do, however, incorporate UML. One such method is the Rational Unified Process (RUP); another is feature-driven development (FDD). UML sequence diagrams, due to their intuitive nature and versatility, have become integral to the front-end modeling activities of these processes. Sequence diagrams are used to model the following:

About actor personalities

Actor personalities can be helpful in discovering and identifying actors who can participate in a use case scenario. An actor can have multiple personalities within a use case and across multiple use cases. So far, four different actor personalities have been identified as enhancements or stereotypes to the UML specification: initiator, server, receiver, and facilitator. Because actor personalities can be reflected in sequence diagrams, you should be familiar with their functions.

- **An initiator** is an external entity that sets a certain system behavior in motion. Initiators can request services or generate events. In sequence diagrams where actors are present, initiators start the sequence in motion.
- **External server** personalities provide services to others. Servers aid the system in achieving its goal by providing functionality or information externally. Many external systems including operating systems are server personalities. Servers tend to receive messages but probably will not generate them.
- **Receiver** personalities receive information from the system. They may provide services but they do so in a passive way. As a result, they may not provide value to the system but should provide value to other actors. An example of a receiver is a data warehouse or external backup system. Receivers generally receive messages from objects in the system but usually do not generate them.
- **A facilitator** is an actor that performs an action on behalf of another actor. An example of a facilitator is a video clerk who rents the videos on behalf of the video customer.

- Use case scenarios
- Protocols in a framework
- Subsystems
- Classes
- Method logic

A brief explanation of each of the above functions is in order.

**Use case scenarios**

For our example application, we'll be using sequence diagrams to model a single use case scenario. A use case is a single task performed by an actor interacting with your application toward a specified goal. An actor is any end user, organization, or system that interacts with, but is external to, your application. (See **About actor personalities** to learn about the four actor personalities; for an in-depth discussion of use case scenarios, see **Resources**.)

**Protocols in a framework**

A protocol sits between a framework and its interchangeable components called *ensembles*. Understanding the interactions required of a framework aids in the development of new ensembles. Sequence diagrams are often used to document these interactions.
**Subsystems**
Large projects are decomposed into smaller, more manageable pieces called *subsystems*. The interfaces between subsystems are vital to their proper integration into the larger whole that is the system. Sequence diagrams are used to specify the interactions between classes on the borders of these subsystems.

**Classes**
Some classes (such as `Socket` and `InetAddress`) require a sophisticated sequence of method invocations for proper interaction. These sequences form the protocols for interacting with such a class or set of classes. Sequence diagrams can be used to describe the uses of a class or interacting group of classes, thus describing the protocols required for interaction.

**Method logic**
Sequence diagrams are excellent for documenting method logic. In fact, some CASE tools, given a Java method, will automatically generate a sequence diagram. Sequence diagramming can be used to design a future method or to document the flow of an existing method.

**About the example application**
We'll learn about sequence diagramming with the help of an example loan processing application. Because this column is focused on modeling, not method, we want to move straight into diagramming, so we'll keep the details of the application fairly loose. The essential functions we'll diagram for the loan processing application are as follows:

**Use case: Submit loan request**
1. An applicant completes and submits a loan application to the bank via the Internet.
2. The system validates the information on the loan application, checking that it is correct and as complete as possible.
3. The system forwards a loan request to an external credit bureau for a credit report on the applicant.
4. The system calculates the applicant's credit score based on the returned credit report.

**Getting started**
The first step to creating a sequence diagram is to determine whether the diagram will represent an interaction with an external or internal entity. If you're modeling a use case scenario, your sequence diagrams will generally represent an interaction with an external entity. If you're modeling protocols in a framework, the diagrams might express either an internal or an external interaction. Diagrams for subsystems, classes, and individual method logic generally represent internal entities only. Whatever the case, the type of interaction you will be modeling determines the first (and left-most) element in the sequence diagram.

An interaction with an external entity indicates that an actor will be part of the interaction. An internal interaction might be initiated by an actor (if subsystem use cases are the basis for the interaction), but more likely it would be started by a generic class called `Sender`. If an actor starts the interaction, the actor falls under the category of initiator, one of four common actor personalities (see About actor personalities for details).
We'll focus on diagramming one scenario for our loan processing application: the *submit loan request* use case outlined above. Note the changes to our sequence diagram as an applicant completes an online loan application and submits it over the Internet. In this scenario, the applicant is external to the system and is, therefore, represented by an actor. We'll start by adding the actor, Applicant, to the diagram, as shown in Figure 1.

**Figure 1. Adding the applicant**

Adding the players

Once the initiator of the interaction is in place, the next step is to add the objects that it will interact with over the course of the scenario. The names of these objects should reflect the behavior of either classes or instances. (The choice of classes or instances gives a distinct meaning to the sequence chart, but I'll save a discussion of the difference between the two for next time.)

For the example scenario, we'll add two classes: LoanApplication and LoanRequest. A loan application is required when applying for a loan. It contains information about the applicant and the desired loan. A loan request is a form the bank sends out to a credit bureau upon receiving a loan application. It contains some information from the loan application, as well as a request for information about the applicant's credit history. The addition of these two classes to the sequence diagram is shown in Figure 2.
**Figure 2. Adding the two interacting classes**

![Sequence diagram](image)

**Connecting the dots -- er, dashes**

Sequence diagrams are intuitive to most software developers. They map objects and actors (horizontal axis) to time (vertical axis). Messages connect the objects, moving from one object to another down the vertical axis as the messages occur over time. These messages are connected to a vertical, dashed line, originating from the middle of the bottom of the object or actor. This line is called a *lifeline*.

On the horizontal axis, we represent messages with arrows sometimes called *call arrows* or *message arrows*. A message arrow points from the sender (tail) to the receiver (head). These arrows are used to capture the dynamic behavior of the system. Calls usually start on the left and move toward the right. That is, the initial arrow in an interaction usually comes from the left. When we create a new instance of a class, we draw the arrow pointing to the class itself rather than to the lifeline. The first step in our scenario is to create a new loan application, so we'll draw an arrow between the *Applicant* and the *LoanApplication*. Since creating a new instance in Java involves calling a constructor, we could label this arrow with the constructor name and possibly its arguments.

We're still in the analysis phase of the software development life cycle, so we want to include as much analysis information as possible. One of our business analysts mentions that we call the act of creating a new loan application "completing the loan application." If we wanted to remain true to this sequence diagram in construction, we could implement *complete* as a public static method that calls the *LoanApplication* constructor, as seen in Figure 3.
Diagramming an activation

When a message is received by a class or instance, it creates a box on the lifeline of the receiving object; this is called an activation. An activation represents the flow of control in the method of the receiver. When a message results in the creation of an object, the first activation represents the logic of the constructor. Subsequent messages will result in the creation of new activations.

Once a message is received, the receiving object can, in turn, send messages to itself or to other objects. This is shown by the tails of the arrows, which represent messages originating in the activation and terminating in new activations. When an object calls itself, the new activation is placed on top of the old one.

In our scenario, the applicant interacts with the loan application twice, first to complete it and second to submit it. When the LoanApplication receives the submit message, it validates itself by sending a validate message to itself. If it is valid, it creates a new LoanRequest to be sent to the credit bureau. Figure 4 shows the validation of the LoanApplication.
As the arrow flies: Indicating time passages

We use a slanted arrow to indicate the passage of a substantial amount of time between when a message is sent and when it is received. This notation is used to show that a call is not atomic. Examples of calls that are not atomic are method invocations via CORBA or RMI, or messages sent over a network.

In our example, the credit bureau is an external system, an actor that has the server personality (see About actor personalities for details). Servers usually do not originate messages, but rather have messages sent to them -- in this case the request for a credit report, sent by the credit checker. The credit checker represents the credit bureau. It tracks and forwards requests to the credit bureau, tracks and receives the responses, and otherwise sets up connections between the loan processing application and the credit bureau. The credit bureau will receive the request and process it according to its own schedule. We use the slanted arrow to represent this passage of time, as shown in Figure 5 below.

At the end of an activation, the return to the caller is implicit. In some cases, however, you may want to make the return explicit. Explicit return calls are indicated by a dashed arrow whose tail is the receiver and whose head is the sender. The explicit return arrow is often labeled with the value returned by the call. For our example, we have added an explicit arrow between the CreditBureau and the CreditChecker. This arrow could be labeled creditReport, because that is the object that is returned from the requestCreditReport method.
Figure 5. Obtaining a CreditReport

What’s next
As I stated at the beginning of this column, sequence diagrams are useful for depicting the internal behavior of a system over time. In this installment, I've taken you through the first steps of building a sequence diagram by modeling the interactions between objects. In the next installment, I’ll introduce the two forms of sequence diagram (generic and instance) and explain the role of conditional logic in sequence diagramming, using examples drawn from simple Java methods. See you then!