

Sequence Diagrams

Sequence diagrams emphasize the time-based flow of events. A sequence diagram shows the participating objects along the top of the diagram, with messages listed from top to bottom in order of execution. The basic sequence diagram has a frame around the outside. In the upper left corner of that frame is a box with a “dog-eared” corner containing `sd` followed by the name of the diagram. Figure 89 shows a simple sequence diagram.

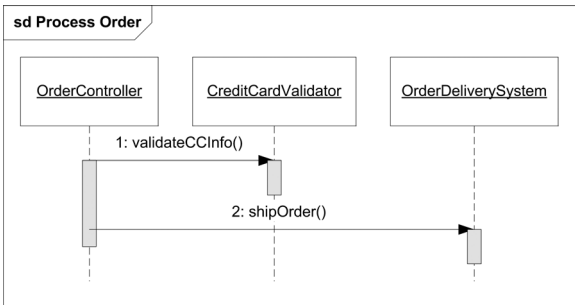


Figure 89. A simple sequence diagram

Interaction participants

Participants in a sequence diagram are indicated by rectangles along the top with the following notation:

object_name [selector] : class_name ref decomposition

where:

object_name

Specifies the name of the instance involved in the interaction

selector

Is an optional part of the syntax that can be used to choose a particular instance from a collection of instances, such as an array or list

class_name

Specifies the name of the type of the participant

Decomposition

Is an optional part of the name referencing another interaction diagram that shows further details about how this participant handles the messages it receives

You can use the keyword `self` to indicate that the participant is the classifier that owns the sequence diagram.

Object creation and deletion

Each object has a dashed lifeline running vertically down the diagram that shows when the object comes into existence and when it is destroyed. Objects created during the time covered by a sequence diagram are often shown directly above the message that creates them. Object creation is represented by a dashed line with an open arrow head extending from the creating object's lifeline to the newly created object's lifeline. You should use the «create» stereotype to indicate that the message causes the target object to be instantiated. Figure 90 shows object creation in sequence-diagram form.

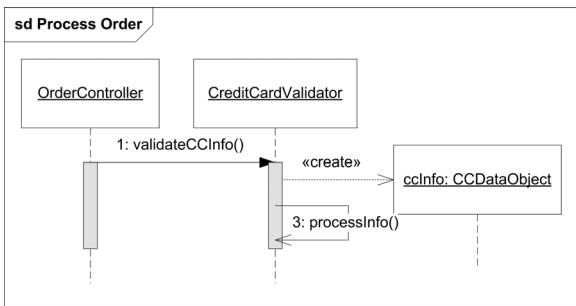


Figure 90. A sequence diagram showing object creation

Likewise, objects destroyed during the time covered by a sequence usually are not drawn beyond the message that causes the destruction. A destruction message is represented by a solid line with a filled arrow pointing to the target object's lifeline and stereotyped with «destroy». The lifeline of the object is terminated with an X, as shown in Figure 91.

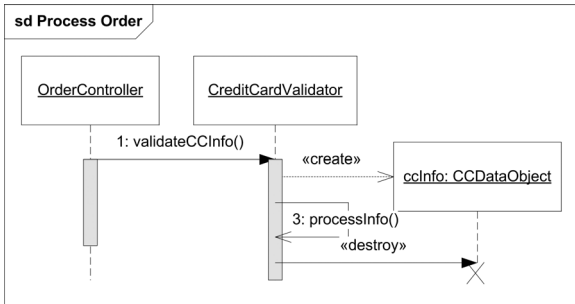


Figure 91. A sequence diagram showing object destruction

Object messages

Each message passed to an object invokes some type of response, called an *action*. The most basic type of message is a call message. Call indicates that the message is an invocation of an operation on the target object. An object can call an operation on itself, in which case, the operation is modeled as a link back to the object. A message that results in an operation invocation is named for the invoked operation. UML allows you to specify arguments to the operation as part of the message name; however, most UML tools do not support this.

The syntax for a message is:

attribute = *signal_or_operation_name*
 (*arguments*) : *return_value*

where:

attribute

Is an optional part of the syntax that provides a named holder (variable) for the return value from the call. You can then reference this attribute in later messages.

signal_or_operation_name

Specifies the name of the signal to send or the operation to invoke.

arguments

Indicates a comma-separated list of arguments to pass to the operation or signal.

return_value

Explicitly shows what the return value is.

Synchronous calls (meaning the caller is blocked until the target operation is complete) are represented with a filled arrow, as shown in Figure 92.

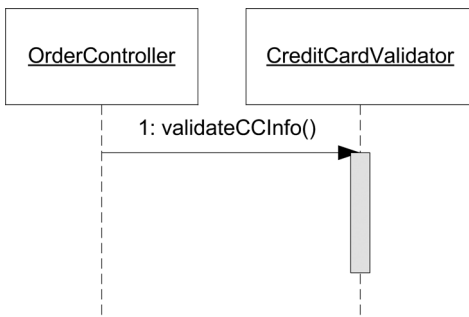


Figure 92. A sample call message

Show asynchronous calls (meaning the caller doesn't wait for the return value) with an open arrow, as shown in Figure 93.

Return is a special type of message that indicates the result of a previous call to an operation. Return is represented as a dashed line ending in an arrow that points to the object receiving the return value. The link is often named for the returned object. Figure 94 shows a return value from a call.

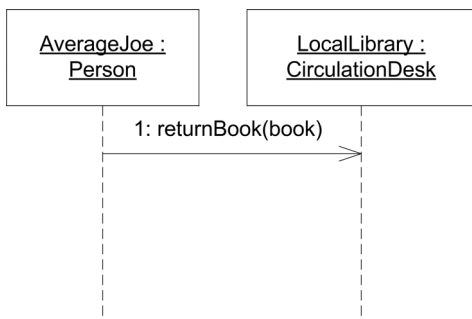


Figure 93. Sample asynchronous call

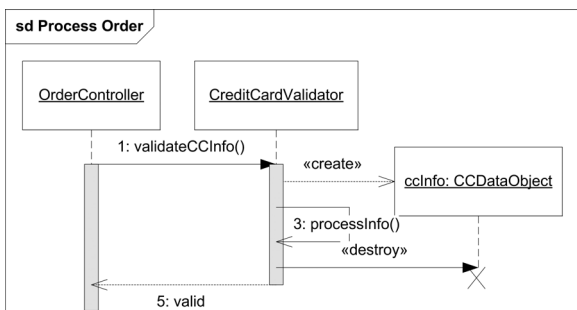


Figure 94. A sequence diagram showing return values

Return values can arrive out of order because asynchronous messages do not require the caller to wait for the result. To show that a return value may arrive after other messages are sent, simply draw the return arrow such that it intersects the original caller further down its lifeline than other messages. For example, you could send in three orders for takeout food before the first one is ready to be picked up, as shown in Figure 95.

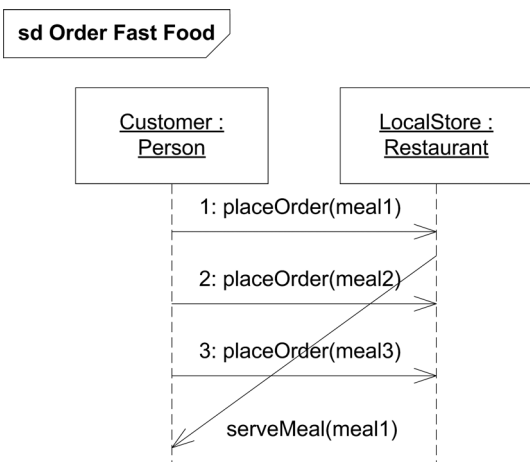


Figure 95. Asynchronous messages with an out of order response

Lost and found messages

UML 2.0 introduced the concept of messages that do not reach their destination (*lost* messages) or are from unknown sources (*found* messages). Note that lost and found are relative terms; the receiver or sender might only be unknown with respect to your current sequence diagram. Lost messages are commonly used to show how a system handles a network failure resulting in an undelivered message. Found messages are commonly used for modeling exception handling—you don't necessarily care who threw the exception; you simply want to show how it is handled.

To indicate a lost message, simply draw a filled dot at the end of a message arrow, as shown in Figure 96.

Conversely, a found message originates from a filled dot, as shown in Figure 97.

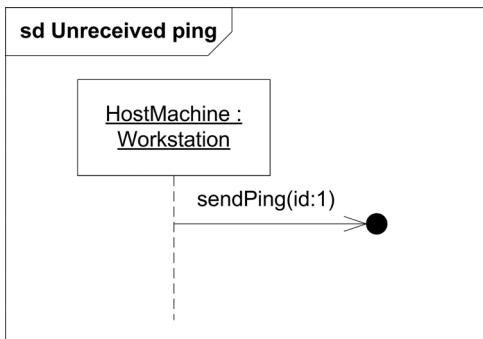


Figure 96. Example of a lost message

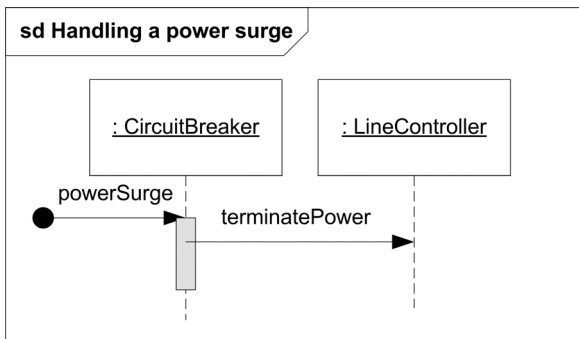


Figure 97. Example of a found message

State invariants

You can place invariant conditions across lifelines in a sequence diagram to indicate that the condition must be true for all following messages to execute. This should be considered similar to an assertion in Java or C++; it should not be used to model a conditional evaluation. See the next section, “Interaction operators,” for a better way to model conditional execution.

You indicate a state invariant by placing a Boolean expression between curly braces across object lifelines. The expression must evaluate to true for any messages below the expression to be evaluated. Figure 98 shows a sample state invariant.

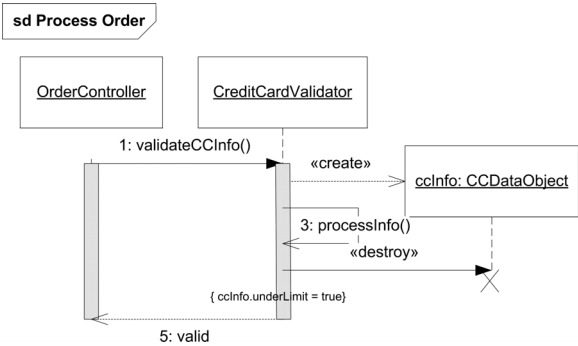


Figure 98. Example state invariant

Interaction operators

UML 2.0 greatly improves the ability of a sequence diagram to capture conditional and looping logic. The basic building block is a *combined fragment*. A combined fragment is a frame around a set of messages with an interaction operator written in the upper-left corner. Figure 99 shows a simple *critical* combined fragment (see later in this section for a definition of critical).

The following are the common interaction operators available in UML 2.0:

alt

An *alternative* is an if-then-else execution. To indicate the alternative flows, simply divide the combined fragment with a dashed line. Each flow can have a Boolean guard condition, written between square brackets, that must evaluate to true for the flow to execute. You may

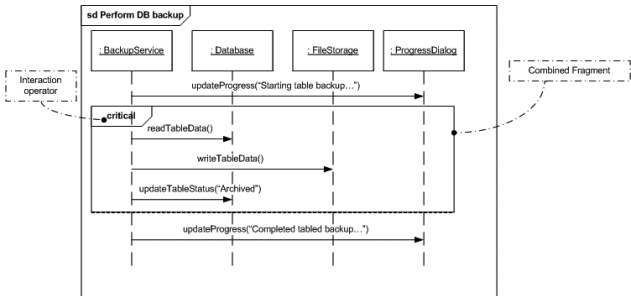


Figure 99. Sample combined fragment

use the reserved else guard condition to indicate a flow that should execute if none of the others evaluate to true. Figure 100 shows a simple example of alternatives.

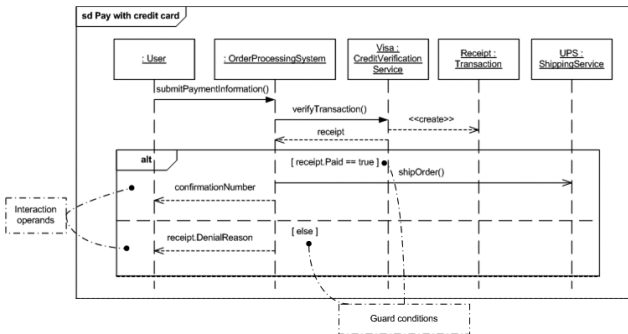


Figure 100. Example alternative operator

opt

An *option* is a combined fragment that executes only if its guard condition evaluates to true. Conceptually, options are similar to alternatives with only one flow.

break

A *break* is a combined fragment with a guard condition stipulating that if the condition evaluates to true and the fragment executes, then the enclosing interaction should terminate.

par

A *parallel* fragment indicates that the messages in the different regions can be interleaved and executed in parallel. Note that the relative ordering within a region must be maintained. See Figure 101 for an example of a parallel section.

neg

A *negative* fragment indicates a set of messages that are invalid in the current context. For example, you could indicate that you cannot call `Show()` on a window after `Dispose()` has been called.

critical

A *critical* region is a set of messages that must be treated as an atomic block, meaning they cannot be interrupted by another thread or process. `critical` is often used to model semaphores. See Figure 101 for an example of a critical section.

assert

An *assertion* is a fragment that is the only valid execution path. Assertions are typically associated with a state invariant to guarantee the state of a system. If the state invariant evaluates to false, the fragment will not execute (which is illegal because it's an assertion).

loop

A *loop* allows you to model messages that should be repeated a number of times. After the loop operator, you should express the minimum and maximum number of times that the messages should repeat with the following syntax:

```
loop ( min, max )
```

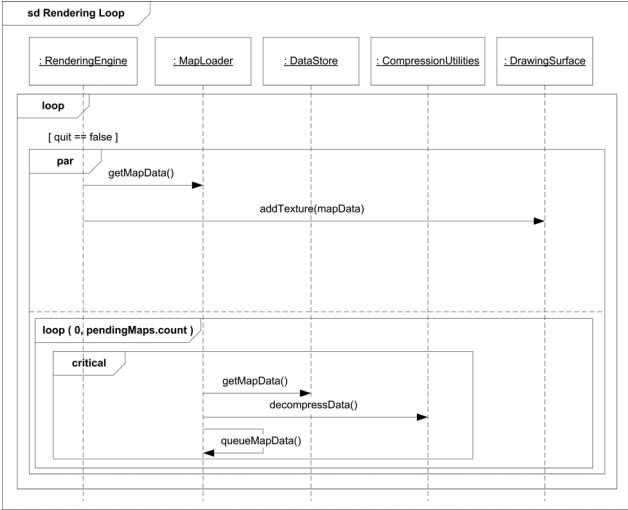


Figure 101. Example loop and parallel operators

where min and max are optional. To show unbounded, simply use an asterisk (*). An example of a loop is shown in Figure 101.