6.4 Sequence Diagrams

A sequence diagram shows elements as they interact over time, showing an interaction or interaction instance. Sequence diagrams are organized along two axes: the horizontal axis shows the elements that are involved in the interaction, and the vertical axis represents time proceeding down the page. The elements on the horizontal axis may appear in any order.

6.4.1 Elements

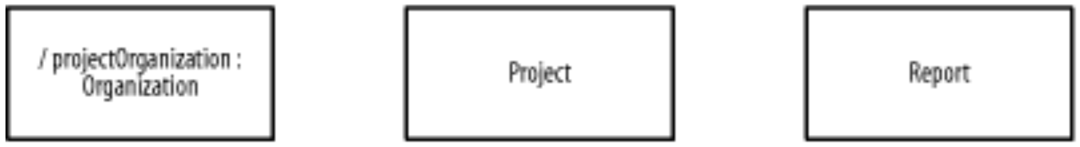
Sequence diagrams are made up of a number of elements, including class roles, specific objects, lifelines, and activations. All of these are described in the following subsections.

6.4.1.1 Class roles

In a sequence diagram, a class role is shown using the notation for a class as defined in Chapter 3, but the class name is preceded by a forward slash followed by the name of the role that objects must conform to in order to participate within the role, followed by a colon. Other classes may also be shown as necessary, using the notation for classes as defined in Chapter 3.

Class roles and other classes are used for specification-level collaborations communicated using sequence diagrams. Figure 6-8 shows the `projectOrganization` class role as well as the `Project` and `Report` classes.

Figure 6-8. A class role and two classes

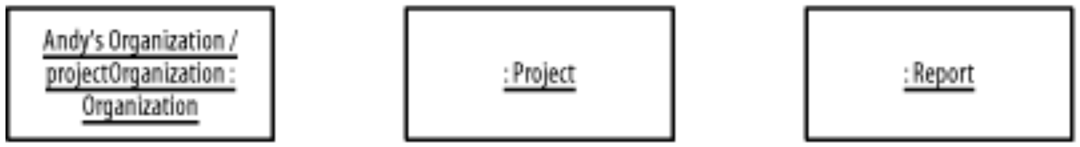


6.4.1.2 Specific objects

In a sequence diagram, a specific object of a class conforming to a class role is shown using the notation for objects as defined in Chapter 3, but the object name is followed by a forward slash followed by the name of the role followed by a colon followed by the class name, all fully underlined. Other objects may also be shown as necessary using the notation for objects, as defined in Chapter 3.

Specific objects conforming to class roles and other objects are used for instance-level collaborations communicated using sequence diagrams. Figure 6-9 shows that Andy's organization plays the role of an organization that contains a project that is the subject of the report. Figure 6-9 also shows anonymous `Project` and `Report` objects.

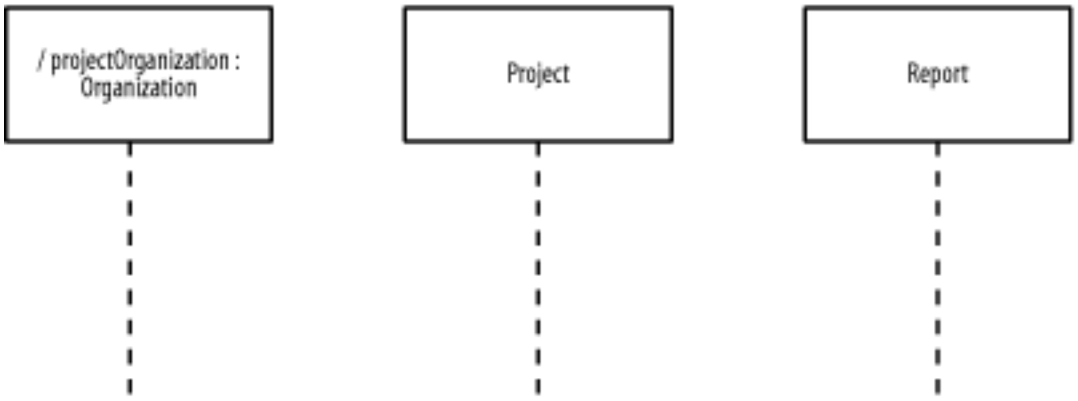
Figure 6-9. An object conforming to a class role



6.4.1.3 Lifelines

A lifeline, shown as a vertical dashed line from an element, represents the existence of the element over time. Figure 6-10 shows lifelines for the class role (`projectOrganization`) and classes (`Project` and `Report`) in Figure 6-8. Lifelines may also be shown for the objects in Figure 6-9.

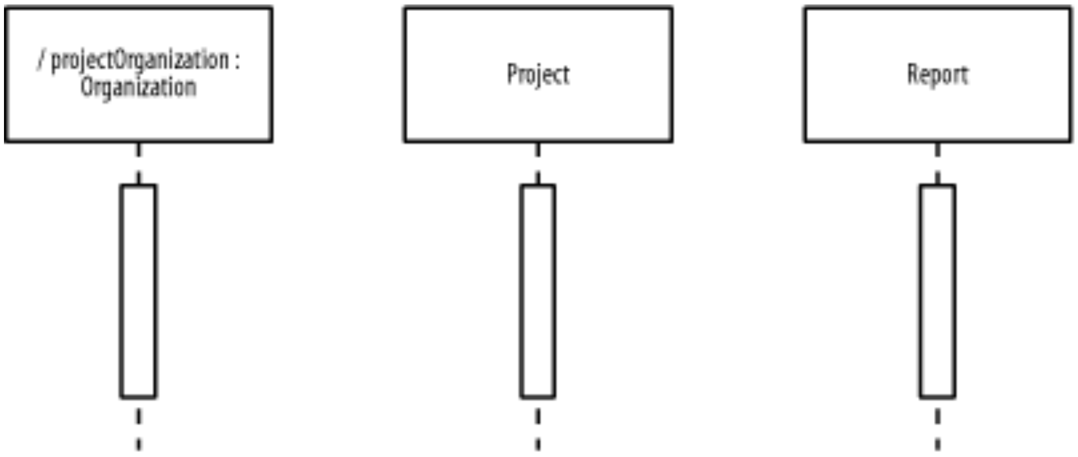
Figure 6-10. Lifelines



6.4.1.4 Activations

An optional activation, shown as a tall, thin rectangle on a lifeline, represents the period during which an element is performing an operation. The top of the rectangle is aligned with the initiation time, and the bottom is aligned with the completion time. Figure 6-11 shows activations for the class roles in Figure 6-8, where all the elements are simultaneously performing operations. Activations may also be shown for the objects in Figure 6-9.

Figure 6-11. Activations



6.4.2 Communication

In a sequence diagram, a communication, message, or stimulus is shown as a horizontal solid arrow from the lifeline or activation of a sender to the lifeline or activation of a receiver. In the UML, communication is described using the following UML syntax:

```
[guard] *[iteration] sequence_number : return_variable := operation_name


    (argument_list)
```

in which:

guard
: Is optional and indicates a condition that must be satisfied for the communication to be sent or occur. The square brackets are removed when no guard is specified.

iteration
: Is optional and indicates the number of times the communication is sent or occurs. The asterisk and square brackets are removed when no iteration is specified.

sequence_number
: Is an optional integer that indicates the order of the communication. The succeeding colon is removed when no sequence number is specified. Because the vertical axis represents time proceeding down the page on a sequence diagram, a sequence number is optional.

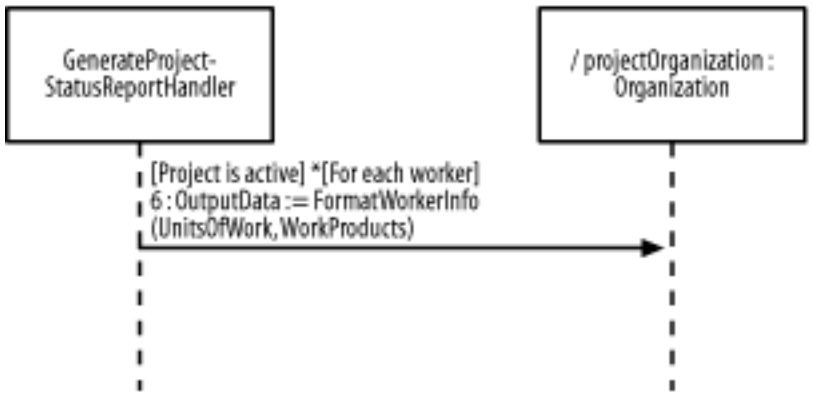
return_variable
: Is optional and indicates a name for the value returned by the operation. If you choose not to show a return variable, or the operation does not return a value, you should also omit the succeeding colon and equal sign.

operation_name
: Is the name of the operation to be invoked.

argument_list
: Is optional and is a comma-separated list that indicates the arguments passed to the operation. Each parameter may be an explicit value or a return variable from a preceding communication. If an operation does not require any arguments, the parentheses are left empty.

Figure 6-12 shows the communication occurring between the `GenerateProject-StatusReportHandler` class (which is responsible for managing the overall generation of the project status report) and the `projectOrganization` class role.

Figure 6-12. Sequence diagram communications


GenerateProject-StatusReportHandler
/ projectOrganization : Organization
[Project is active] *[For each worker] 6 : OutputData := FormatWorkerInfo (UnitsOfWork, WorkProducts)

Let's take a step-wise look at how the communication notation used in Figure 6-12 is built. To begin with, the communication invokes a `FormatWorkerInfo` operation that formats a worker's information:

```
FormatWorkerInfo
```

This operation requires a worker's units of work and work products, so we can update the communication to the following:

```
FormatWorkerInfo (UnitsOfWork, WorkProducts)
```

The operation also returns some output data as a string of formatted text, so we update the communication to reflect this:

```
OutputData := FormatWorkerInfo (UnitsOfWork, WorkProducts)
```

In our earlier description of this interaction and collaboration, this operation occurs as the sixth communication in an overall sequence of communications:

```
6 : OutputData := FormatWorkerInfo (UnitsOfWork, WorkProducts)
```

A project may involve more than one worker, thus the operation is to occur once for each worker: We indicate this using the notation for repetition:

```
*[For each worker] 6 : OutputData := FormatWorkerInfo (UnitsOfWork, WorkProdu
cts)
```

Finally, this operation is to occur only if a project is active, and we indicate this using guard notation:

```
[Project is active] *[For each worker] 6 : OutputData := FormatWorkerInfo


(UnitsOfWork, WorkProducts)
```

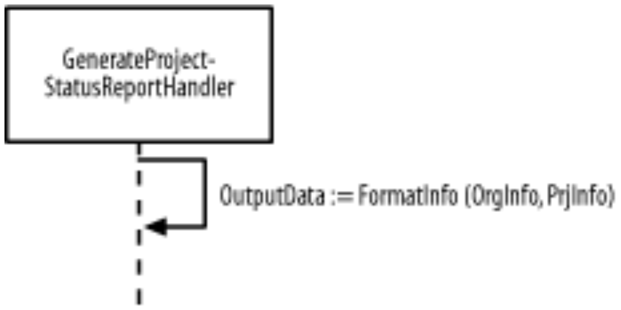
The UML also allows you to show communication using pseudocode or another language. For example, you can use the syntax of Java, C++, C#, or some other programming language. A communication may also be described in the left margin of the sequence diagram.

6.4.2.1 Reflexive communication

Similar to a reflexive association or link, as discussed in Chapter 3, an element may communicate with itself where a communication is sent from the element to itself. In the UML, a reflexive communication is shown as a horizontal solid arrow from the lifeline or activation of an element that loops back to the same lifeline or activation of the element.

Figure 6-13 shows a reflexive communication for step 6 of the `Generate Project-Status Report` interaction and collaboration description where the `GenerateProject-StatusReportHandler` class formats the organization and project information.

Figure 6-13. Sequence diagram reflexive communications

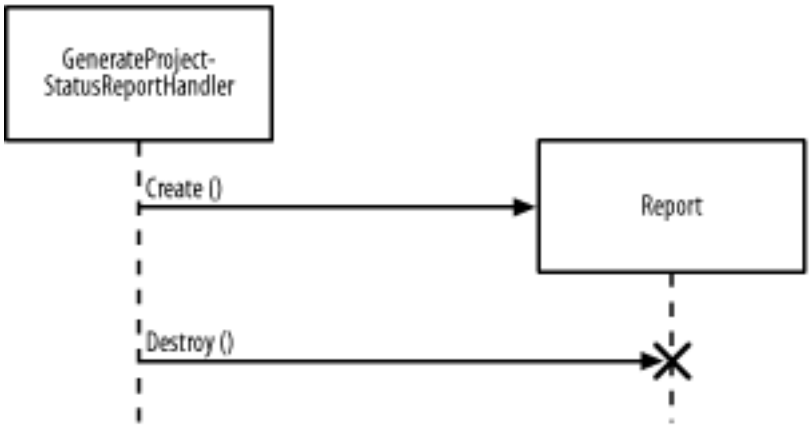

GenerateProject-StatusReportHandler
OutputData := FormatInfo (OrgInfo, PrjInfo)

6.4.2.2 Element creation and destruction

When an element is created during an interaction, the communication that creates the element is shown with its arrowhead to the element. When an element is destroyed during an interaction, the communication that destroys the element is shown with its arrowhead to the element's lifeline where the destruction is marked with a large "X" symbol.

Figure 6-14 shows a communication for step 3 of the `Generate Project-Status Report` interaction and collaboration description in which the `GenerateProject-StatusReportHandler` class creates a report. This figure also shows a communication for step 8 of the `Generate Project-Status Report` interaction and collaboration description where the `GenerateProject-StatusReportHandler` class destroys the report.

Figure 6-14. Sequence diagram creation and destruction communications

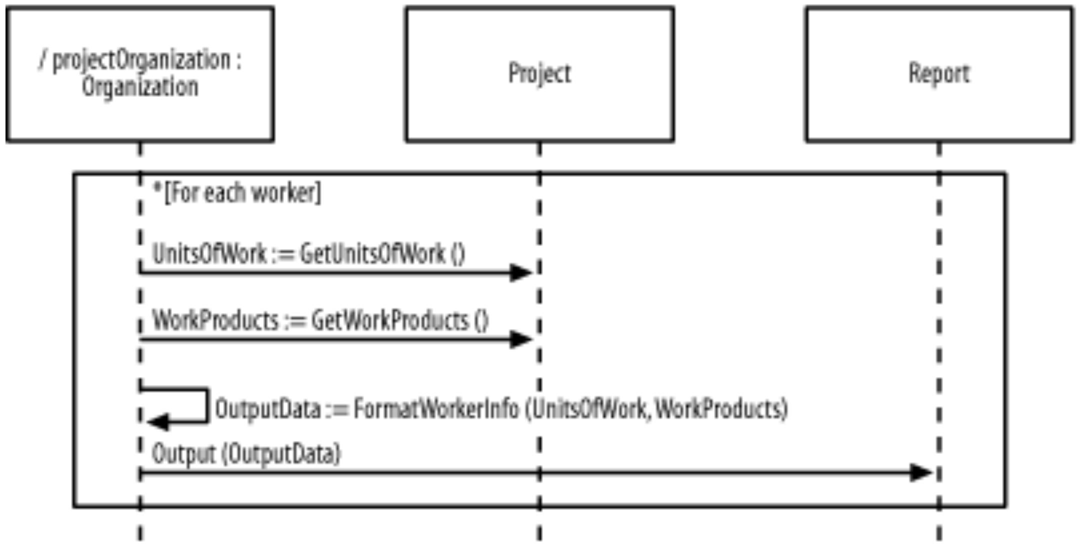


6.4.3 Repetition

In a sequence diagram, repetition (which involves repeating a set of messages or stimuli) within a generic-form interaction is shown as a set of communications enclosed in a rectangle.

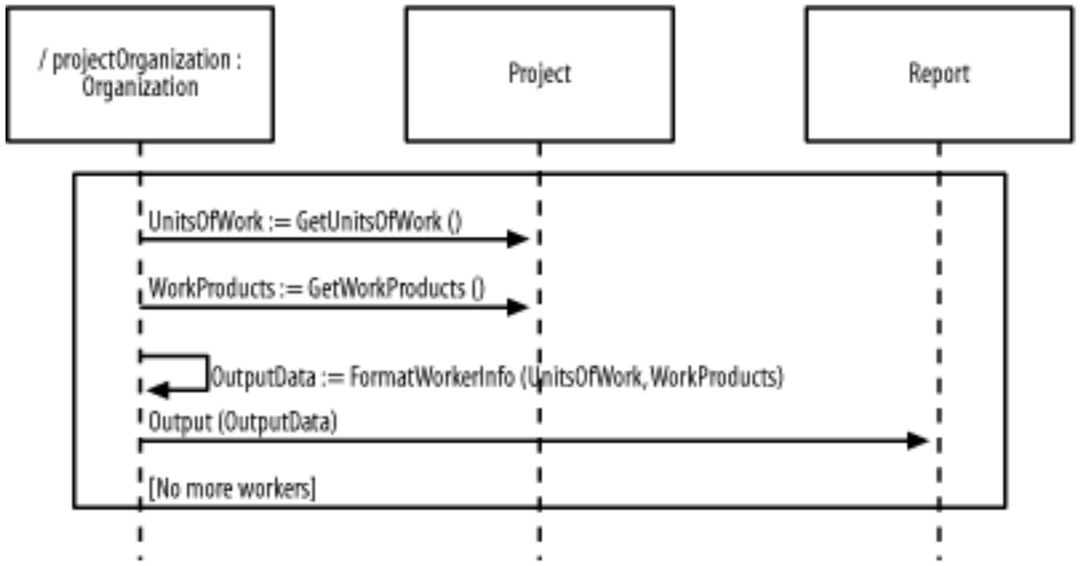
An iteration expression may be used at the top-inside or bottom-inside of the rectangle to indicate the number of times the communications inside the rectangle occur. Figure 6-15 shows step 6b of the `Generate Project-Status Report` interaction and collaboration description using an iteration expression in which the `GenerateProject-StatusReportHandler` class retrieves the worker's units of work and work products, formats this information, and outputs the formatted information to the report element. Note the use of `*[For each worker]` in the upper left, which indicates that the communications shown occur once for each worker involved in the project.

Figure 6-15. Sequence diagram repetition using an iteration expression within a generic-form interaction


/ projectOrganization : Organization
Project
Report
*[For each worker]
UnitsOfWork := GetUnitsOfWork ()
WorkProducts := GetWorkProducts ()
OutputData := FormatWorkerInfo (UnitsOfWork, WorkProducts)
Output (OutputData)

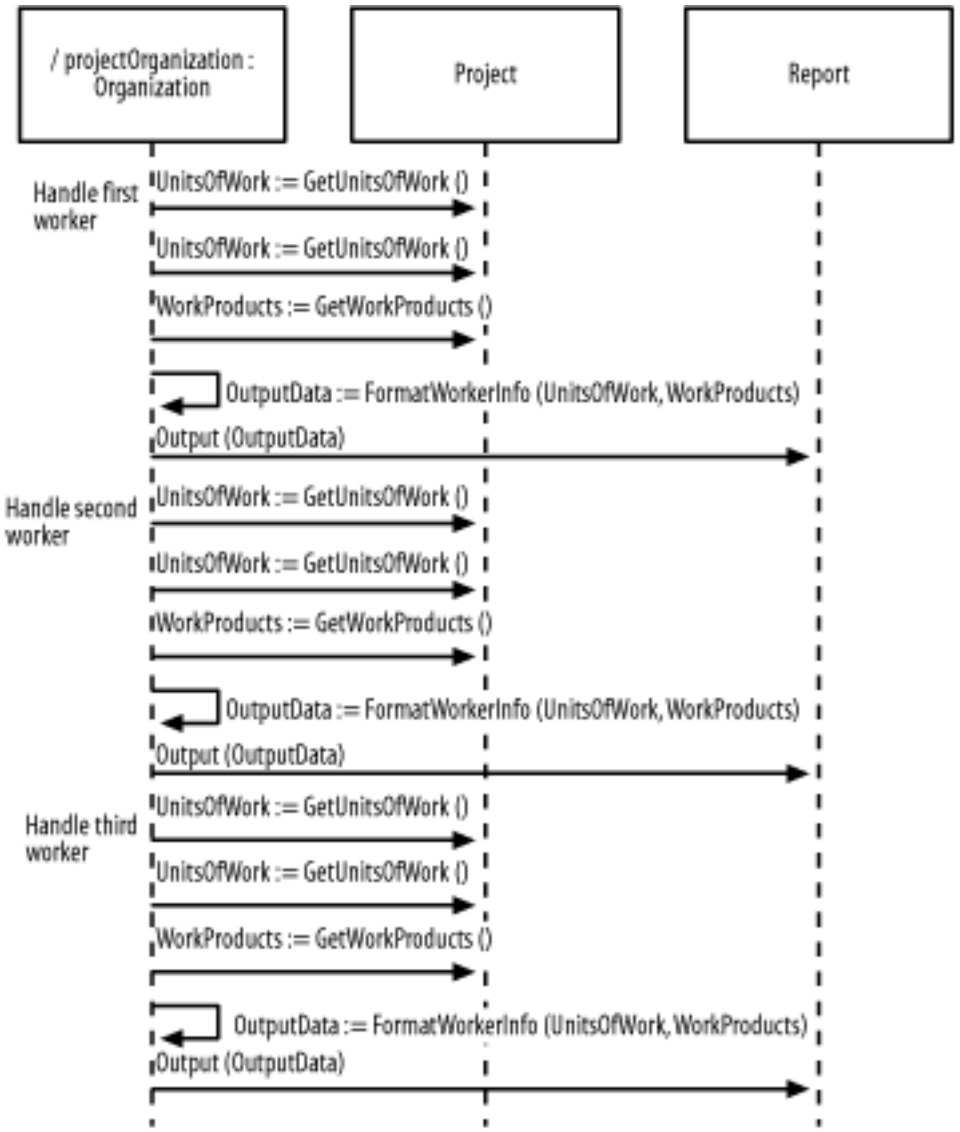
Alternatively, a guard expression may be used at the top-inside or bottom-inside of the rectangle to indicate the condition that must be satisfied in order to terminate the repetition. Figure 6-16 shows Figure 6-15 using a guard expression to express the same iteration as in Figure 6-15. Rather than specify explicitly that the set of communications is repeated for each worker, the guard expression specifies that the communications are to be repeated until no more workers remain to be processed.

Figure 6-16. Sequence diagram repetition using a guard expression within a generic-form interaction


/ projectOrganization : Organization
Project
Report
UnitsOfWork := GetUnitsOfWork ()
WorkProducts := GetWorkProducts ()
OutputData := FormatWorkerInfo (UnitsOfWork, WorkProducts)
Output (OutputData)
[No more workers]

Repetition within an instance-form interaction involves showing the actual set of messages or stimuli that are repeated. Figure 6-17 shows step 6b of the `Generate Project-Status Report` interaction and collaboration description (Figure 6-15 and Figure 6-16) for a project that contains exactly three workers, each with two units of work and one work product. Notice that I have also described the interaction in the left margin to make the diagram more readable.

Figure 6-17. Sequence diagram repetition within an instance-form interaction

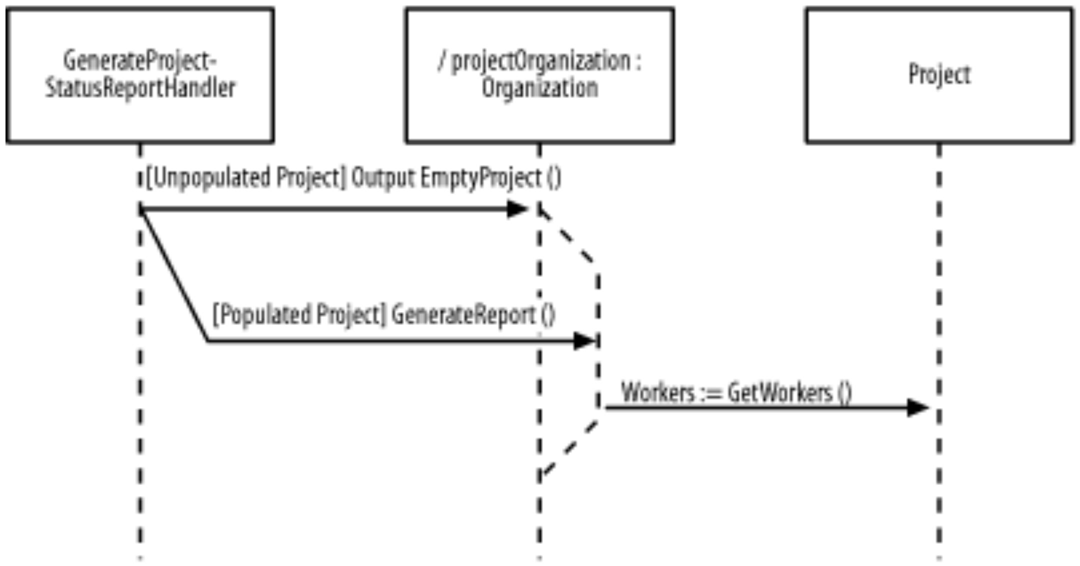

/ projectOrganization : Organization
Project
Report
Handle first worker
UnitsOfWork := GetUnitsOfWork ()
UnitsOfWork := GetUnitsOfWork ()
WorkProducts := GetWorkProducts ()
OutputData := FormatWorkerInfo (UnitsOfWork, WorkProducts)
Output (OutputData)
Handle second worker
UnitsOfWork := GetUnitsOfWork ()
UnitsOfWork := GetUnitsOfWork ()
WorkProducts := GetWorkProducts ()
OutputData := FormatWorkerInfo (UnitsOfWork, WorkProducts)
Output (OutputData)
Handle third worker
UnitsOfWork := GetUnitsOfWork ()
UnitsOfWork := GetUnitsOfWork ()
WorkProducts := GetWorkProducts ()
OutputData := FormatWorkerInfo (UnitsOfWork, WorkProducts)
Output (OutputData)

6.4.4 Conditionality

In a sequence diagram, conditionality (which involves communicating one set of messages or stimuli rather than another set of messages or stimuli) within a generic-form interaction is shown as multiple communications leaving a single point on a lifeline or activation, with the communications having mutually exclusive guard expressions. A lifeline may also split into two or more lifelines to show how a single element would handle multiple incoming communications, and the lifelines would subsequently merge together again.

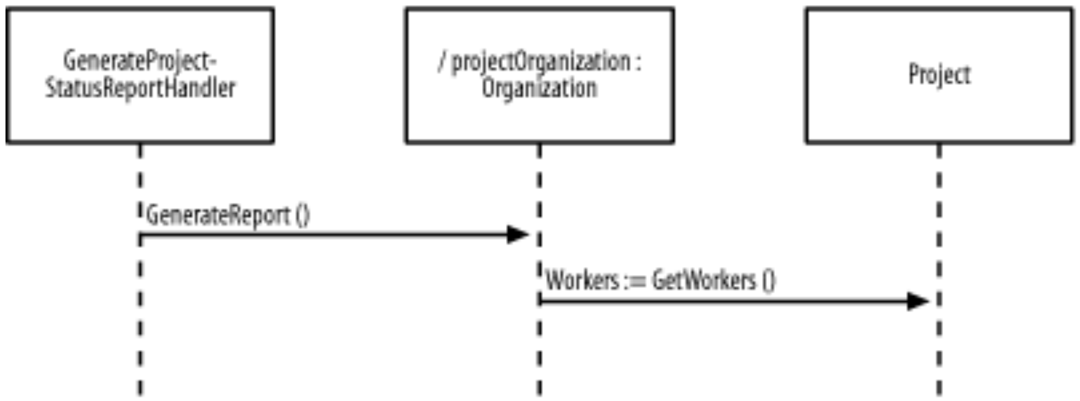
Figure 6-18 shows steps 5 and 6 of the `Generate Project-Status Report` interaction and collaboration description where the `GenerateProject-StatusReportHandler` class requests that the `projectOrganization` class role indicate that the project is empty if the project is a newly created or unpopulated project, and the `GenerateProject-StatusReportHandler` class requests that the `projectOrganization` class role continue generating information for the report element if the project is not a newly created or populated project. In this figure, only the first communication is shown for actually generating the report. If there are no other communications for actually generating the report, the `GenerateReport` communication may go to the same lifeline as the `OutputEmptyProject` communication. I use different lifelines in the figure because each lifeline represents a different path of execution.

Figure 6-18. Sequence diagram conditionality within a generic-form interaction


GenerateProject-StatusReportHandler
/ projectOrganization : Organization
Project
[Unpopulated Project] Output EmptyProject ()
[Populated Project] GenerateReport ()
Workers := GetWorkers ()

Conditionality within an instance-form interaction involves the set of messages or stimuli that are communicated for a specific condition. Figure 6-19 shows Figure 6-18 for a project that is populated.

Figure 6-19. Sequence diagram conditionality within an instance-form interaction


GenerateProject-StatusReportHandler
/ projectOrganization : Organization
Project
GenerateReport ()
Workers := GetWorkers ()