

CA314 – Object Oriented Analysis & Design - 6

File name: CA314_Section_06_Ver01

Author: L Tuohey

No. of pages: 15

Table of Contents

6. UML Interaction Diagrams (see ref 1, Chap. 9, 10)	3
6.1 Introduction.....	3
6.2 Collaborations	3
6.3 Collaboration diagrams	3
6.3.1 Activation of objects	4
6.3.2 Message numbering convention	5
6.4 Sequence diagrams.....	6
6.4.1 Main points	6
6.4.2 Additional points.....	9
6.5 Generic interaction diagrams	11
6.5.1 Conditional behaviour.....	11
6.5.2 Iteration	12
6.6 Concurrency	13
6.6.1 Three ways a new "thread of execution" may start up:	13
6.6.2 Modeling several threads of control	14

6. UML Interaction Diagrams (see ref 1, Chap. 9, 10)

6.1 Introduction

In the introductory case study we had an example of a *sequence diagram* being used to show how objects (in a class model) interacted to carry out a specific use case. In this section we look more closely at sequence diagrams and also at *collaboration diagrams* (a second kind of interaction diagram)

Reference 1 notes that one may not need to have interaction diagrams for all use cases; also, that the level of detail to include is a matter of judgement.

It is normal to revise class model while developing interaction diagrams but it should be ensured that the overall class model remains well-structured.

6.2 Collaborations

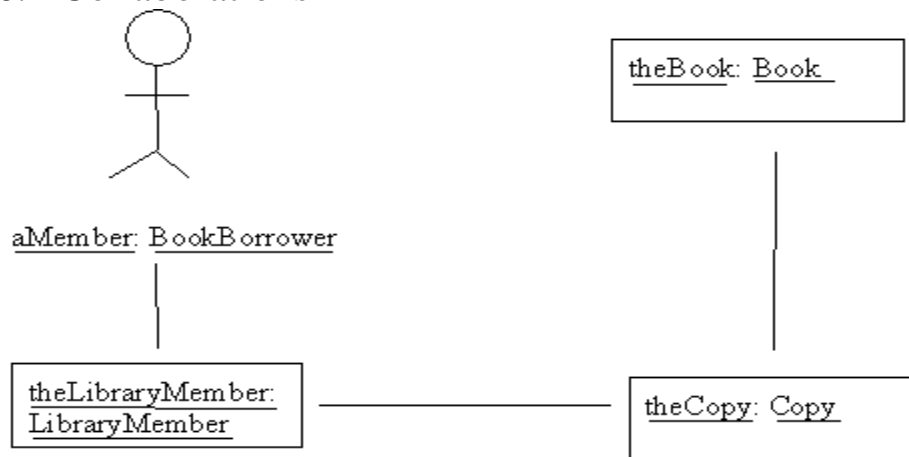


Figure 6.2-1 (from ref. 1): A simple collaboration, showing no interaction

As illustrated above, a collaboration consists of objects, actors and links. It can be thought of as an instance of part of a class model. Only relevant elements (*e.g. to a given use case*) need be shown. An actor that starts such a use case is called the initiator.

6.3 Collaboration diagrams

Collaboration diagrams are the second kind of interaction diagrams, sequence diagrams being the other kind. Collaboration diagrams are good for *showing links between objects* (see Figure 6.3-1 for an example, taken from Ref. 1).

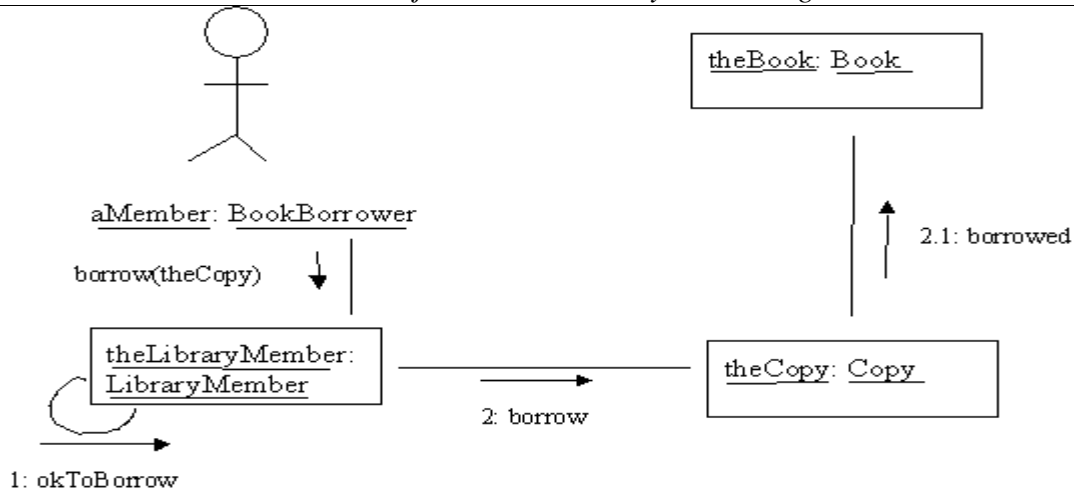


Figure 6.3-1: Interaction shown on a collaboration diagram

In comparison with Figure 6.2-1, the new elements are *messages*. Labeled arrows show messages sent *from* the object at the tail *to* the object at the head (called the target object)

Developing interaction diagrams (such as the collaboration diagram above) can help to identify needed associations between classes and operations within classes - e.g. *the target object must provide the appropriate operation to respond to a message*.

6.3.1 Activation of objects

We say that an object starts to compute when it is *activated* by receiving a message. Consider, for example, the following (simple) sequence of events for objects A, B, C and D:

CA314 - Object Oriented Analysis & Design

Event	Object with control	Live activation stack
Message received by A (from external source)	A	Empty
Message (1) received by B from A	B	A
Message (1.1) received by C from B	C	B A
Message (1.1.1) received by D from C	D	C B A
Message reply received by C from D	C	B A
Message reply received by B from C	B	A
Message reply received by A from B	A	Empty
A finishes computing	None	Empty

The above illustrates *flow of control* from one object to another. It may be helpful to think of *control* as a token that gets passed as a message along the links in a collaboration diagram.

When considering non-concurrent interactions, only actors can initiate activity. On the other hand, where concurrency is possible, one can have so-called *active objects* that can also initiate activity – see later in this section.

6.3.2 Message numbering convention

The UML *message numbering system* is illustrated in Figure 6.3-1. In general, the system is as follows:

- Initial message from actor to an object is not numbered
- First message from one object to another is number 1
- Whenever object O receives a message, the number of that message will be used as a prefix of all messages that are sent until O replies to that message.

Unfortunately, not every author follows this convention.

6.4 Sequence diagrams

6.4.1 Main points

We have had an example (Figure 2.9-2) of such a diagram in the introductory case study:

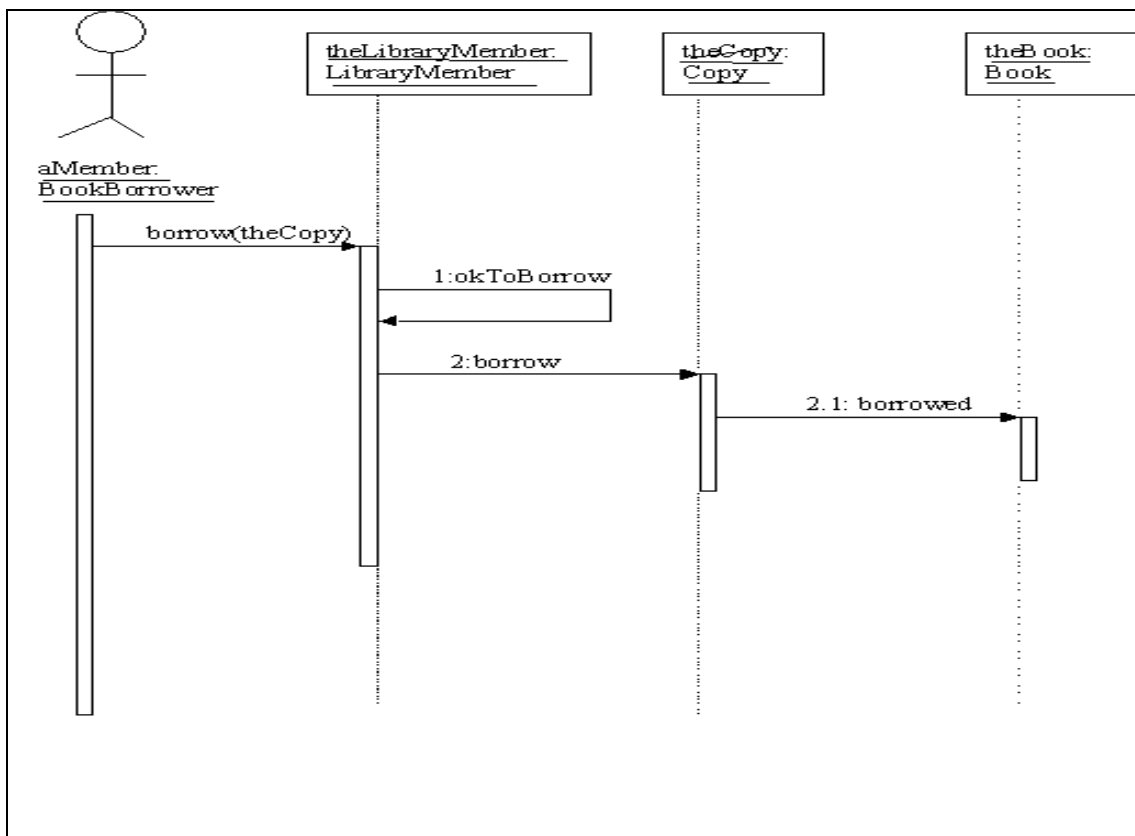


Figure 2.9-2: Interaction shown on sequence diagram (Figure 3.6 of ref. 1)

We use this example to explain further aspects of sequence diagrams in Figures 6.4-1 and 6.4-2.

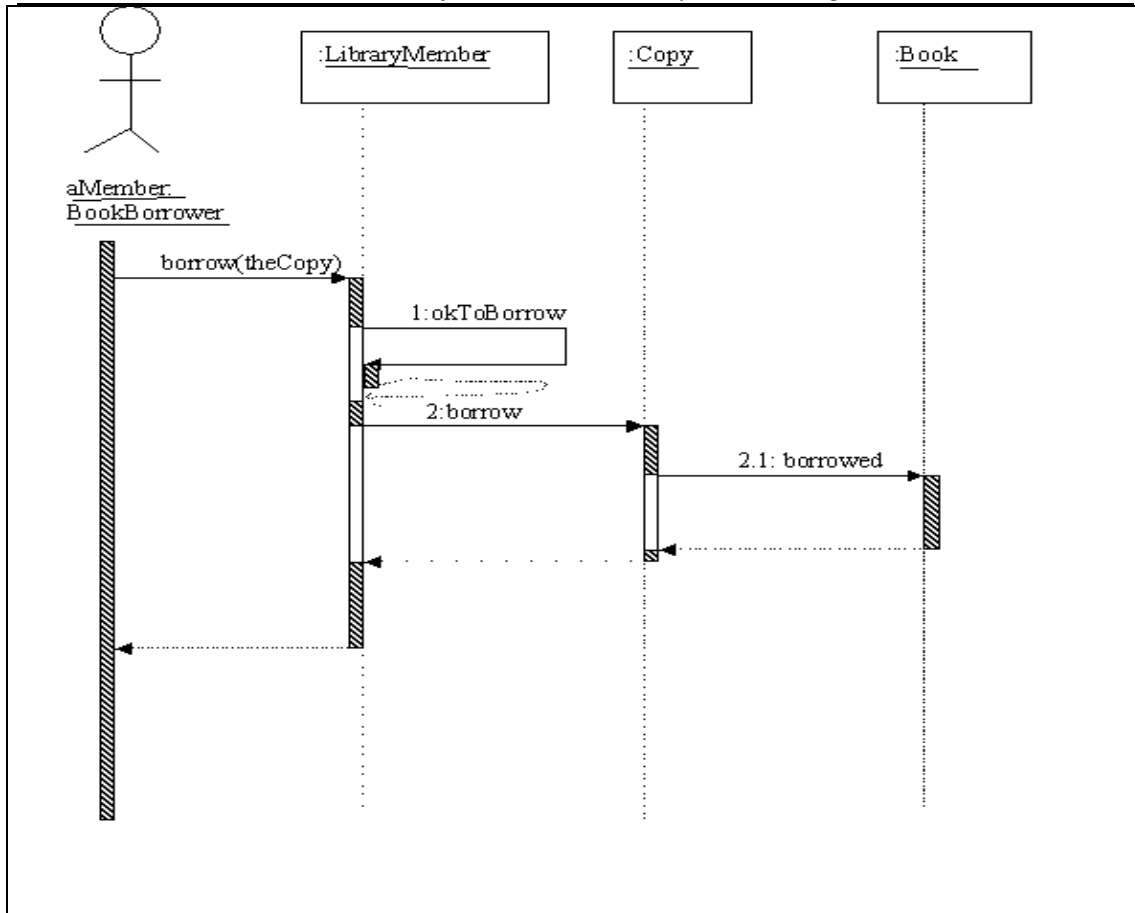


Figure 6.4-1: Interaction with optional features on sequence diagram (ref. 1)

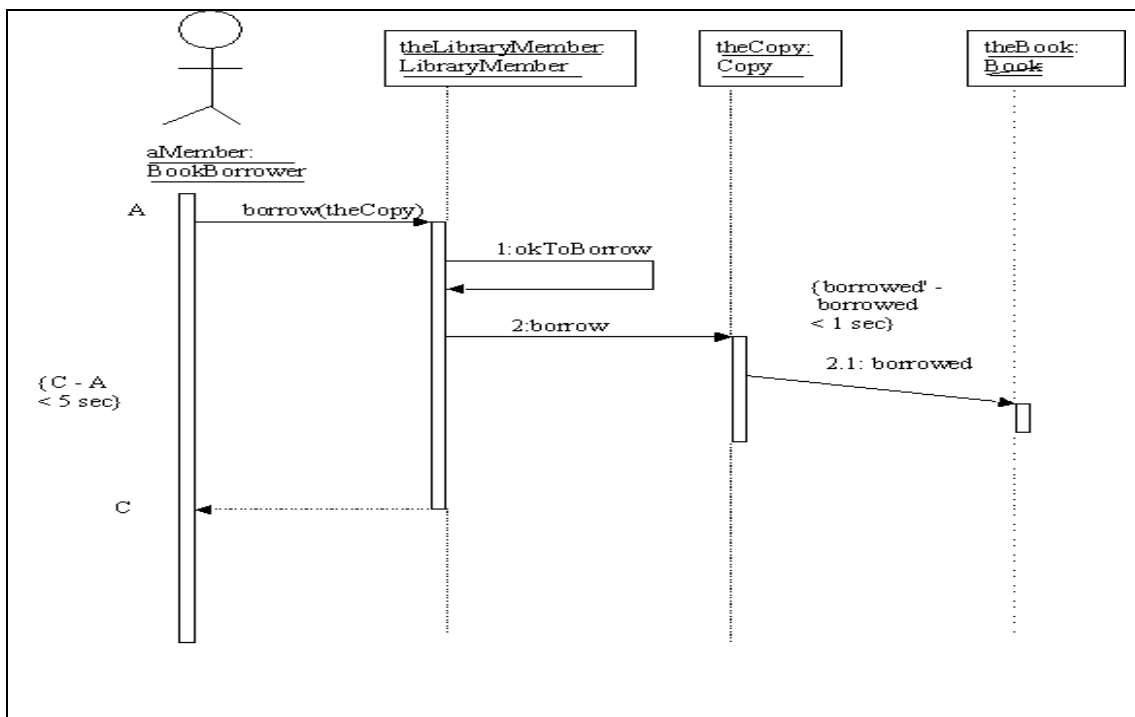


Figure 6.4-2: Showing time features on a sequence diagram (ref. 1)

With reference to the first diagram above (Figure 2.9-2):

- Diagram shows which messages are passed between objects and in what order they must occur - read the messages from top to bottom
- Diagram shows what happens when a library member borrows a copy of the book. It starts from the position of having a certain object of class "LibraryMember" called "theLibraryMember" and a certain object of class "Copy" called "theCopy" (corresponding to a person bringing a physical copy to the issue desk to borrow it)
- "theLibraryMember" acts (as decided earlier for this example) on behalf of the real library member so interaction begins with a message "borrow(theCopy)" passed to it
- Then, after a check whether allowed to borrow, object "theLibraryMember" sends message "borrow" to "theCopy". Finally, "theCopy" sends a message "borrowed" to "theBook" - we need to update system information on how many copies are on loan.

In principle, the order of objects does not matter. However, for (human) readability, try to arrange that most messages flow from left to right.

The second diagram (Figure 6.4-1) illustrates optional shading to highlight when an object is actually computing. Also it shows explicitly (dashed arrows) when responses to messages occur.

A complication occurs when an object sends a message to itself – *somewhat analogous to a recursive procedure call in a programming language*. In effect, the object is then associated with 2 (or more) live activations. This "nested activation" can be shown by offsetting the shading for the new activation as illustrated in Figure 6.4-1. If a diagram is becoming too cluttered, it may be better not to show some or all of such "internal" messages.

The third figure (Figure 6.4-2) illustrates two ways that times and timing constraints may be depicted. The first shows how a constraint on the time it takes the system to respond to an actor's message can be specified. The second shows how a constraint on the time it takes for a message to pass between objects can be stated (a sloping line is used to emphasise more than zero time; borrowed and borrowed' refer to times sent and received, respectively).

6.4.2 Additional points

Suppressing detailed behaviour:

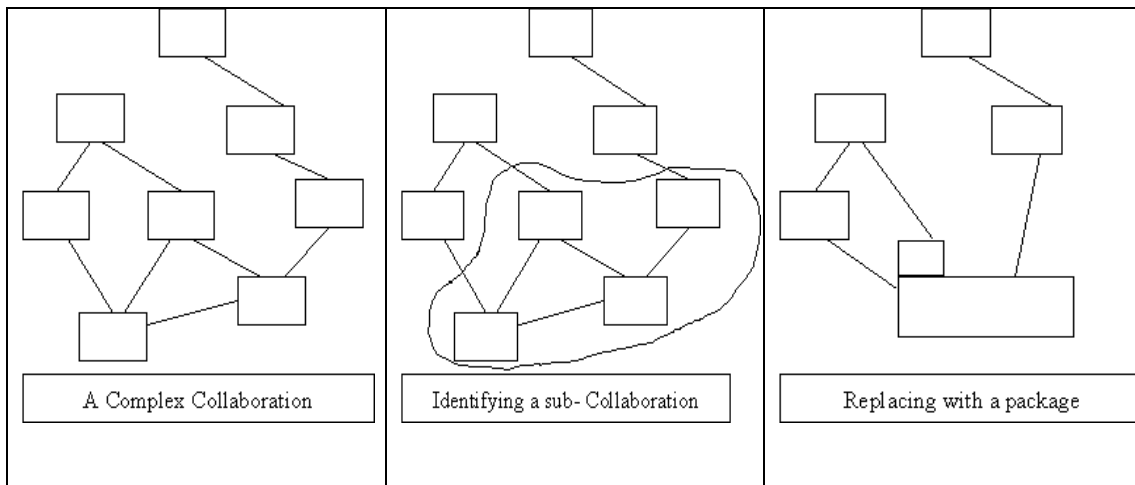


Figure 6.4-3: Using a package to simplify a collaboration (from ref. 1)

Figure 6.4-3 is essentially self-explanatory; its point being to suppress unnecessary detail. A "tab" is included on the package to identify it as such, and the package is given a name. (We will come back to packages in a later section).

Returned values; Creation and deletion of objects:

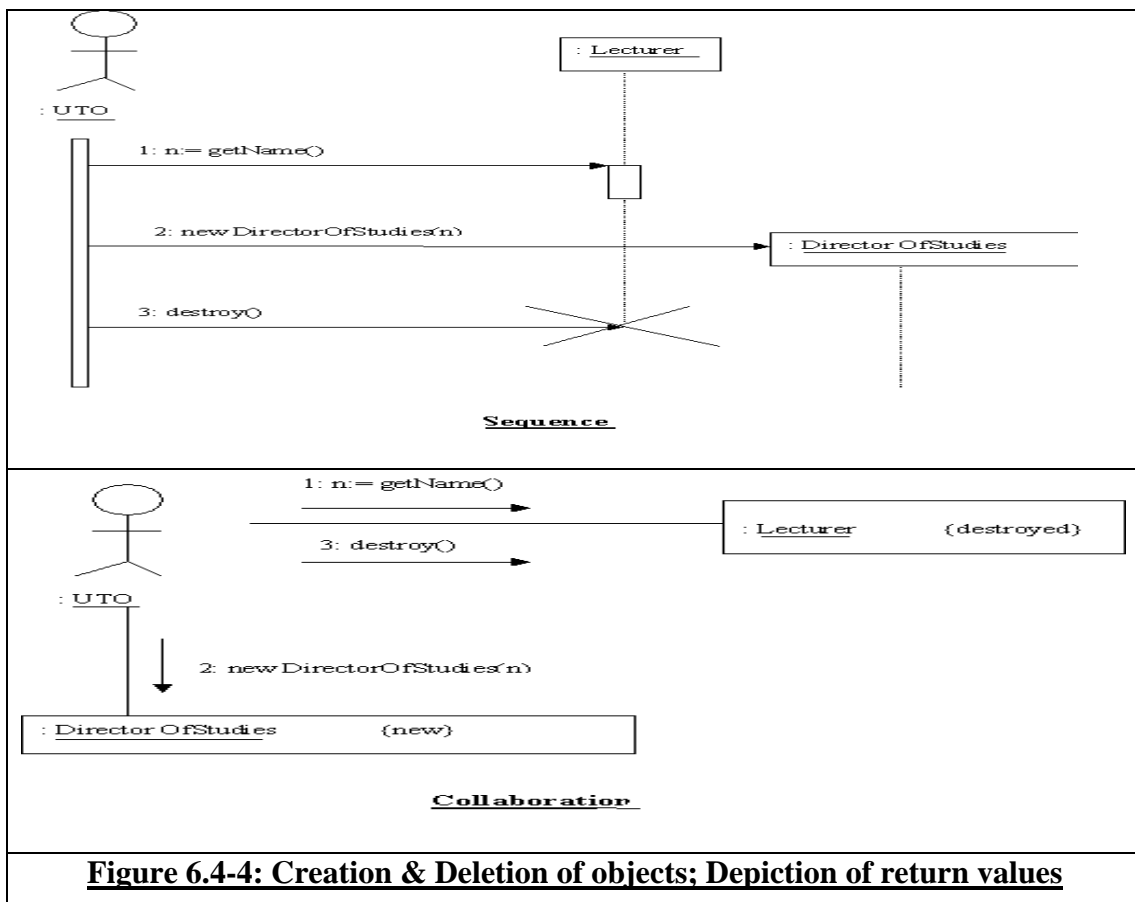


Figure 6.4-4: Creation & Deletion of objects; Depiction of return values

Note: Both parts of Figure 6.4-4 (taken from Ref. 1) illustrate a possible use case called "Promote Lecturer". We note the following,

- Assignment is used to name (and store) the returned value from a message
- Creation and Destruction of objects:
 - Collaboration diagram: Constraints {new}, {destroyed}, {transient} show objects created, destroyed, created & destroyed by an interaction
 - Sequence diagram: "Created" is shown by positioning of the object box (no longer at top). A large "X" is used to mark the end of its activation for a destroyed object. If not destroying itself, there is a message from the "destroying" object into the "X".
- Reference 1 notes that mechanisms for creating (and maybe initialising) and destroying objects vary between languages. Also, it refers to the issue of "garbage collection", whether automatic or programmer responsibility, and the danger of "memory leaks" due to failure to destroy no longer needed objects. The system design must specify where responsibilities for destroying objects lie.

A good design guideline - Law of Demeter:

"Only talk to your *immediate friends*"! This means that, in response to a message *m*, an object *O* should send messages only to

- (1) *O* itself
- (2) objects which are sent as arguments to the message *m*
- (3) objects which *O* creates as part of its reaction to *m*, or
- (4) objects which are directly accessible from *O*, that is, are using values of attributes of *O*.

If these guidelines are not followed then the design will be hard to maintain. Remember that these are just guidelines not a real law; and occasionally it may be best to break it them. Reference 1 provides some more detail and the topic is also dealt with on various internet sites.

A few miscellaneous points:

- In addition to their utility for use cases, interaction diagrams can be used to show how an operation is provided by an object, how a design *pattern* works, and how a complex component should be used (part of its documentation).
- There are some UML ambiguities or lacks of detail, including whether some features are allowed on both types of interaction diagram and on levels of granularity (ref. 1).

6.5 Generic interaction diagrams

There is a general question regarding interaction diagrams, namely whether and how to show all possible sequences of messages that occur or whether to show just one scenario. 'Generic' strictly requires all possible sequences to be shown but sometimes a looser interpretation may be preferable.

6.5.1 Conditional behaviour

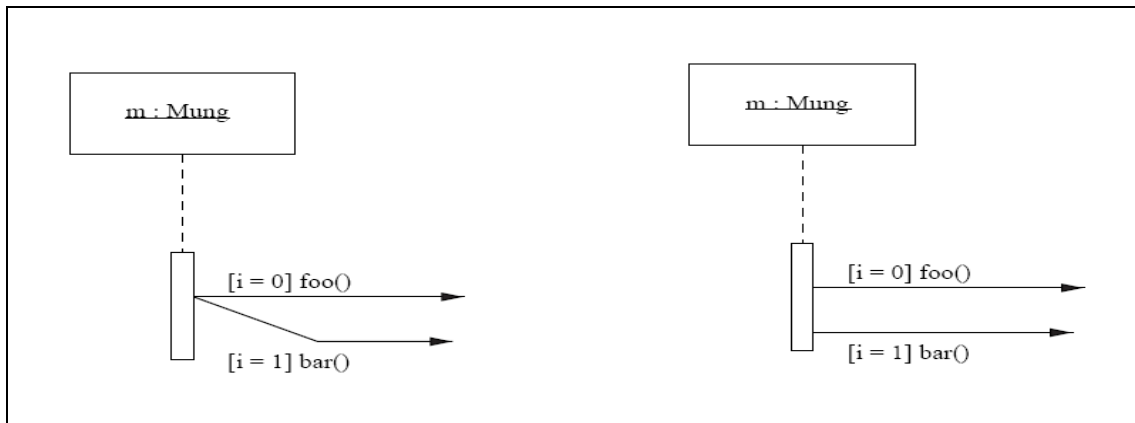


Figure 6.5-1: Two sequence diagram fragments (from ref. 1)

Recall that [...] is used to denote a condition (or guard) in front of a message. The 2 fragments in Figure 6.5-1 depict 2 possible solutions to the same problem. The *branching* solution (left) does not imply a time sequence - only one of the alternatives is followed during any one execution. The solutions are not equivalent.

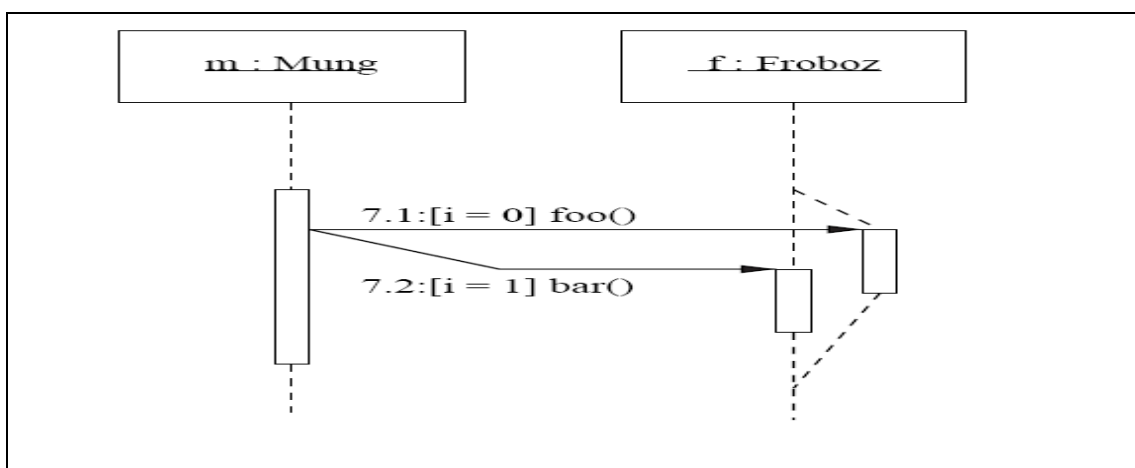


Figure 6.5-2: Fragment of a seq. diagram with branching lifeline (from ref. 1)

Figure 6.5-2 depicts how UML can show where different outcomes result from different conditional messages.

General consideration: Avoid over-complex diagrams even if UML does provide syntax!

6.5.2 Iteration

In the following figure (Figure 6.5-3), the * in

"3.1:*[i:=1..2]a()"

indicates that message "a" is sent repeatedly. The element "[i:=1..2]" is an *iteration clause* telling how many times "a" is to be sent; such a clause is optional.

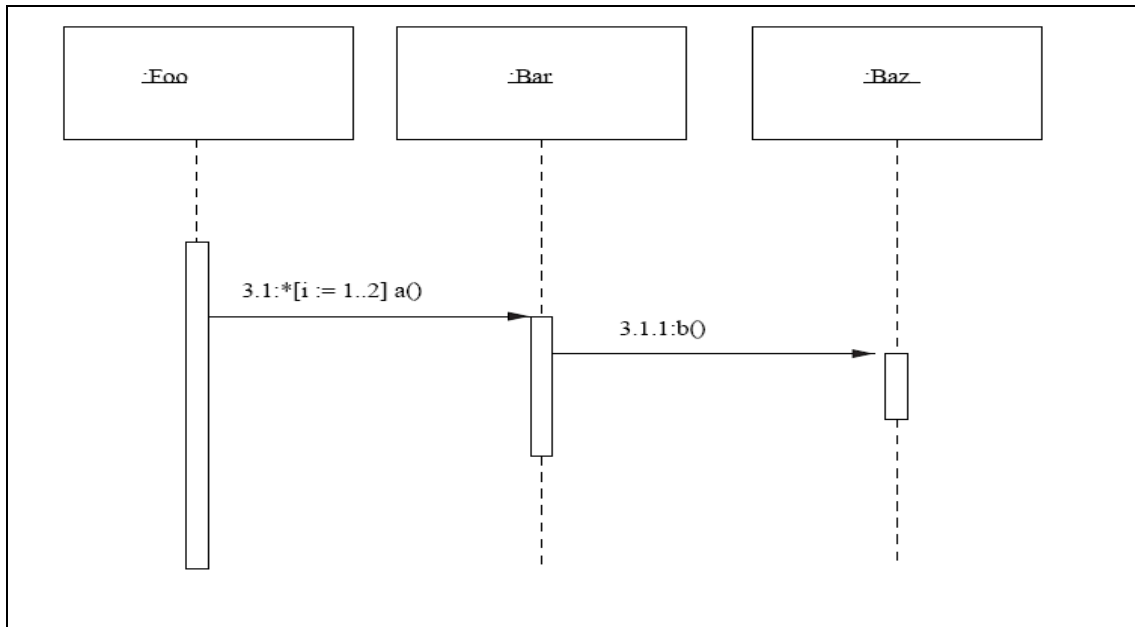


Figure 6.5-3: Seq. diag. fragment: iteration showing messages abab (from ref. 1)

Various forms of iteration clause are possible. For example,

"[item not found]"

would indicate that the corresponding message be sent repeatedly until item is found.

The following figure (Figure 6.5-4) presents an example of nested iteration:

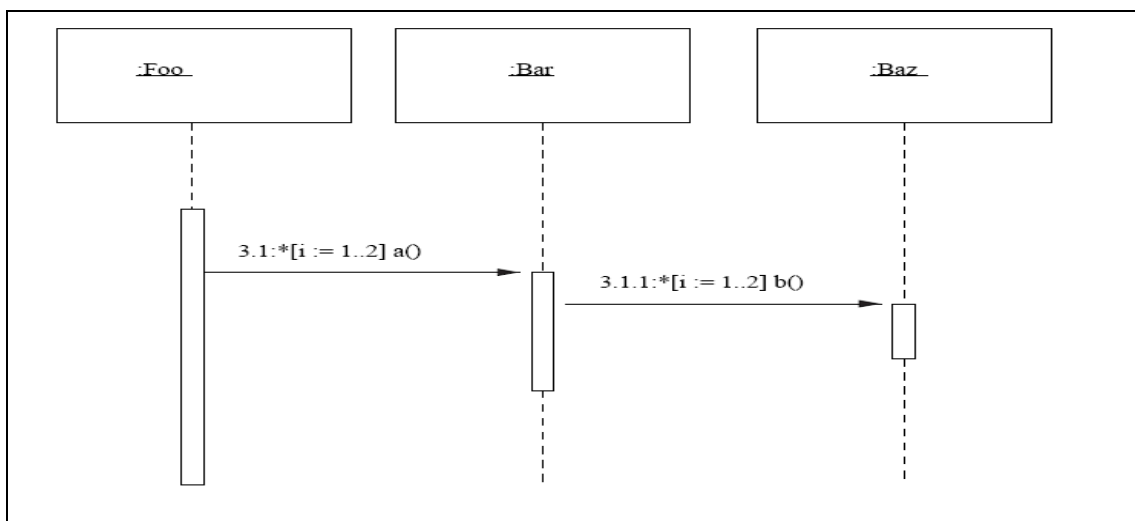
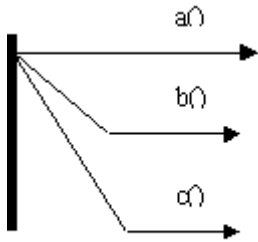


Figure 6.5-4: Seq. diag. frag.: iteration showing messages abbabb (from ref. 1)

6.6 Concurrency

6.6.1 Three ways a new "thread of execution" may start up:

<p>1. Single thread splits into several</p>	<p>For example, an object which is computing sends 3 messages at same time</p>  <p>Notice there are no conditions on these branches. On a collaboration diagram, one could use a numbering convention (e.g. 2.13. A, 2.13.B, etc) to distinguish simultaneous messages</p>
<p>2. An active object is one that owns its own thread of control</p>	<p>An active object sends a message while there is computation going on elsewhere Slightly heavier borders on active objects</p>
<p>3. Object sends an asynchronous message to another object</p>	<p>Can cause another object to become active without the sender having to stop computing</p>

Note: Avoid possible confusion between an active object and an object that becomes "activated" on receipt of a message.

6.6.2 Modeling several threads of control

Prior to section 6.6, we have considered cases where

- one object (at most) is computing at a time
- if an object sends a message it awaits a response
- have a single processor

In section 6.6, on the other hand, we are considering concurrent systems. For example,

- Distributed systems with several processors running in parallel
- Multi-threaded or multi-scheduling
- Reactive systems that receive inputs from environment in various ways

UML provides the following notations:

Interaction type	Symbol	Meaning
Synchronous or call	→	The 'normal' procedural situation. The sender loses control until the receiver finishes handling the message, then gets control back, which can optionally be shown as a return arrow.
Return	←	Not a message, but a return from an earlier message. Unblocks a synchronous send.
Asynchronous	⇒	The sender does not lose control; it sends the message and may continue immediately. The recipient of the message may also become active [i.e. activated], if it wasn't already

Figure 6.6-1: Variants of message sending in sequence diagrams (from ref. 1)

As an illustration, consider the following scenario:

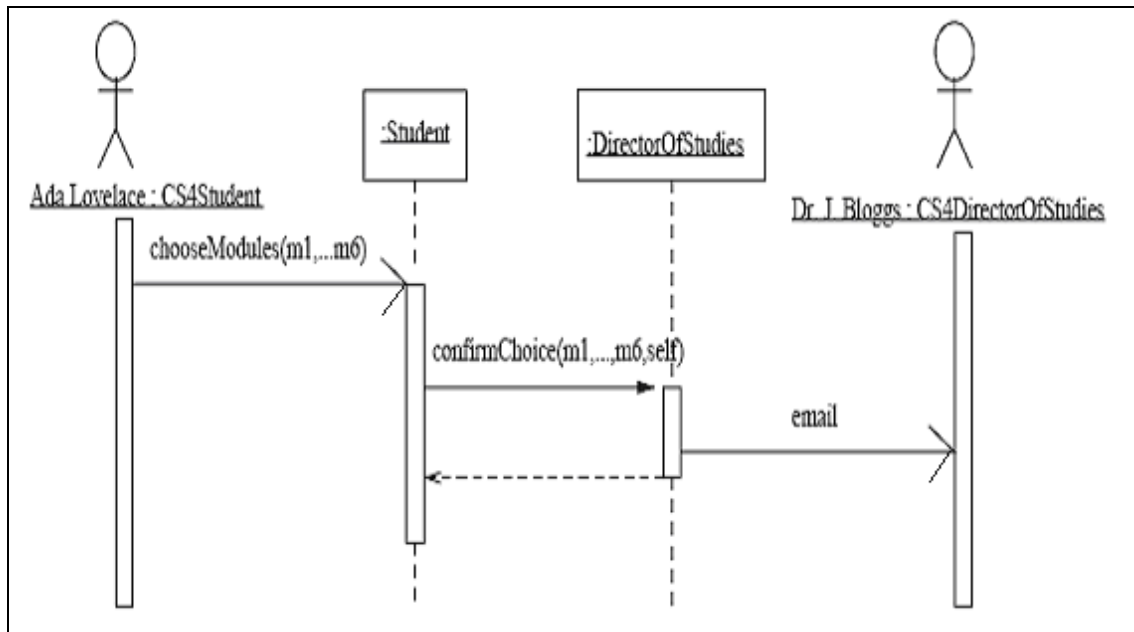


Figure 6.6-2: Asynchronous message passing (from ref. 1)

This sequence diagram “reads” as follows:

A student visits a web-site & chooses a selection of modules. Not necessarily immediately (asynchronous) the system does one of three things of which one is as in the diagram. Thus, if choices are not pre-approved for this student (but are legal), the system sends an e-mail to this student's Director of Studies. The system does not wait for a reply.

The first ("choose...") & last ("email") messages are asynchronous, while the second ("confirm...") is synchronous with its corresponding "return" displayed.

Note: Be advised that special issues arise in concurrent programming.

For information:



Augusta Ada King, Countess of Lovelace (10 December 1815, London – 27 November 1852, Marylebone, London), born **Augusta Ada Byron**, was the only legitimate child of poet Lord Byron. She is widely known in modern times simply as **Ada Lovelace**.

She is mainly known for having written a description of Charles Babbage's early mechanical general-purpose computer, the analytical engine. She is today appreciated as the "**first programmer**" since she was writing programs—that is, encoding an algorithm in a form to be processed by a machine—for a machine that Babbage had not yet built. She also foresaw the capability of computers to go beyond mere calculating or number-crunching while others, including Babbage himself, focused only on these capabilities. This article is licensed under the Creative Commons BY-SA License. It uses material from Wikipedia content.