Chapter 15

INTERACTION DIAGRAM NOTATION

Cats are smarter than dogs. You can't get eight cats to pull a sled through snow.

—JeffVaidez

Objectives

Read basic UML interaction (sequence and collaboration) diagram notation.

Introduction

The following chapters explore object design. The language used to illustrate the designs is primarily interaction diagrams. Thus, it is advisable to at least skim the examples in this chapter and get familiar with the notation before moving on.

The UML includes **interaction diagrams** to illustrate how objects interact via messages. This chapter introduces the notation, while subsequent chapters focus on using them in the context of learning and doing object design for the NextGen POS case study.

Read the Following Chapters for Design Guidelines

This chapter introduces notation. To create well-designed objects, design principles must also be understood. After acquiring some familiarity with the notation of interaction diagrams, it is important to study the following chapters on these principles and how to apply them while drawing interaction diagrams.

15.1 Sequence and Collaboration Diagrams

The term *interaction diagram,* is a generalization of two more specialized UML diagram types; both can be used to express similar message interactions:

- collaboration diagrams
- sequence diagrams

Throughout the book, both types will be used, to emphasize the flexibility in choice.

Collaboration diagrams illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram, as shown in Figure 15.1.

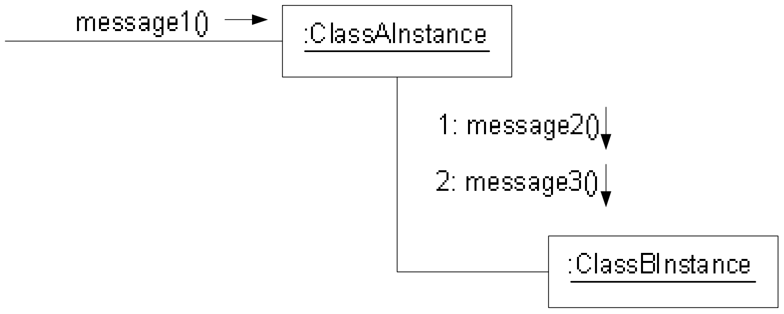


Figure 15.1 Collaboration diagram

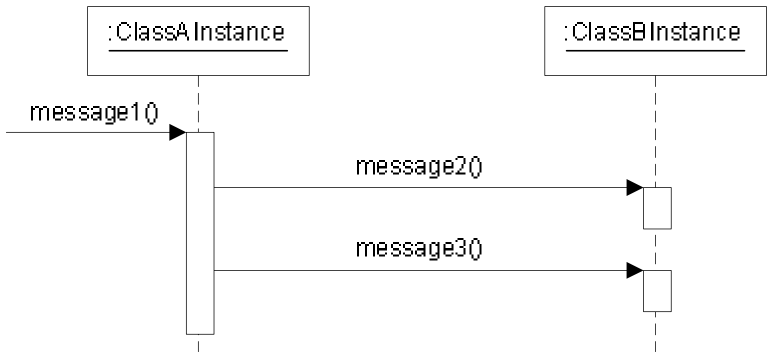


Figure 15.2 Sequence diagram.

Each type has strengths and weaknesses. When drawing diagrams to be published on pages of narrow width, collaboration diagrams have the advantage of allowing vertical expansion for new objects; additional objects in a sequence diagrams must extend to the right, which is limiting. On the other hand, collaboration diagram examples make it harder to easily see the sequence of messages.

EXAMPLE COLLABORATION 1JIAGRAM: MAKEfAYMENT

Most prefer sequence diagrams when using a CASE tool to reverse engineer source code into an interaction diagram, as they clearly illustrate the sequence of messages.

Type	Strengths	Weaknesses
sequence	clearly shows sequence or time ordering of messages simple notation	forced to extend to the right when adding new objects; consumes horizontal space
collaboration	space economical—flexibility to add new objects in two dimensions better to illustrate complex branching, iteration, and concurrent behavior	difficult to see sequence of messages more complex notation

15.2 Example Collaboration Diagram: makePayment

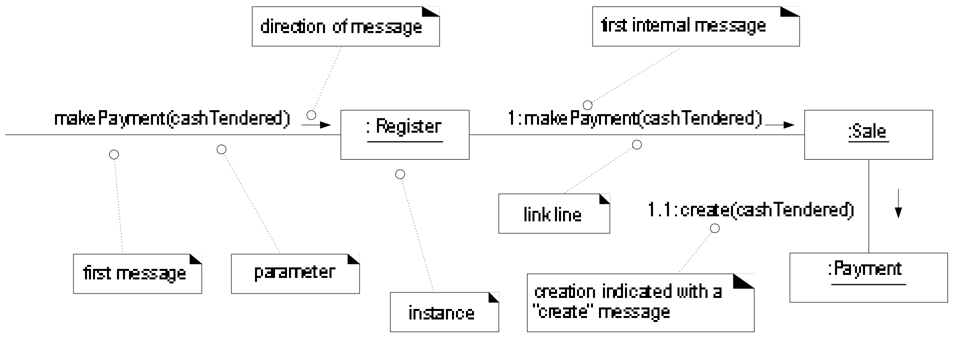

direction of message
first internal message
makePayment(cashTendered)
: Register
1: makePayment(cashTendered)
:Sale
link line
1.1: create(cashTendered)
:Payment
first message
parameter
instance
creation indicated with a "create" message

Figure 15.3 Collaboration diagram.

The collaboration diagram shown in Figure 15.3 is read as follows:

1. The message *makePayment is* sent to an instance of a *Register.* The sender is not identified.
2. The *Register* instance sends the *makePayment* message to a *Sale* instance.
3. The *Sale* instance creates an instance of a *Payment.*

15.3 Example Sequence Diagram: makePayment

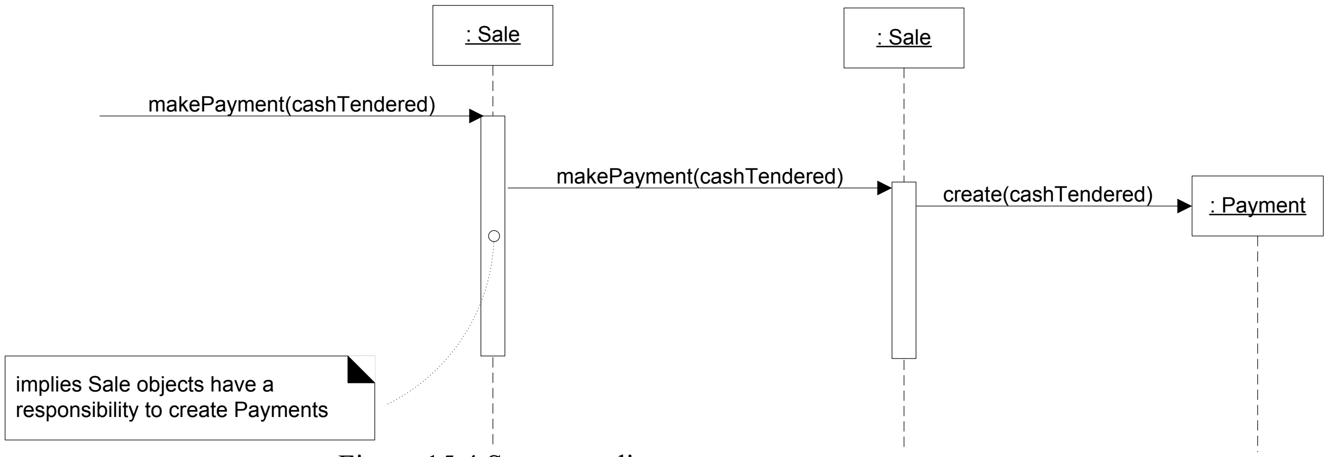

: Sale
: Sale
makePayment(cashTendered)
makePayment(cashTendered)
create(cashTendered)
: Payment
implies Sale objects have a responsibility to create Payments

Figure 15.4 Sequence diagram.

The sequence diagram shown in Figure 15.4 has the same intent as the prior collaboration diagram.

15.4 Interaction Diagrams Are Valuable

A common problem in object technology projects is a lack of appreciation for the value of doing object design via the medium of interaction diagrams. A related problem is doing them in a vague way, such as showing messages to objects that actually require much further elaboration; for example, showing the message *runSimulation* to some *Simulation* object, but not continuing on with the more detailed design, as though by virtue of a well-named message the design is magically complete.

Some non-trivial time and effort should be spent in the creation of interaction diagrams, as a reflection of thinking through details of the object design. For example, if the length of the timeboxed iteration is two weeks, perhaps a half or full day near the start of the iteration should be spent on their creation (and in parallel, class diagrams), before proceeding to programming. Yes, the design illustrated in the diagrams will be imperfect and is speculative, and it will be modified during programming, but it will provide a thoughtful, cohesive, common starting point for inspiration during programming.

Suggestion

Create interaction diagrams in pairs, not alone. The collaborative design will be improved, and the partners will learn quickly from each other.

Note that it is primarily during this step that the application of design skill is required, in terms of patterns, idioms, and principles. Relatively speaking, the creation of use cases, domain models, and other artifacts is easier than the assignment of responsibilities and the creation of well-designed interaction diagrams. This is because there is a larger number of subtle design principles and "degrees of freedom" that underlie a well-designed interaction diagram than most other OOA/D artifacts.

Making interaction diagrams (in other words, deciding on the details of the object design) is a very creative step in OOA/D.

Codified patterns, principles, and idioms can be applied to improve the quality of their design.

The design principles necessary for the successful construction of interaction diagrams *can* be codified, explained, and applied in a methodical fashion. This approach to understanding and using design principles is based on **patterns**—structured guidelines and principles. Therefore, after introducing the syntax of interaction diagrams, attention (in subsequent chapters) will turn to design patterns and their application in interaction diagrams.

15.5 Common Interaction Diagram Notation

Illustrating Classes and Instances

The UML has adopted a simple and consistent approach to illustrate **instances** vs. classifiers (see Figure 15.5):

- *For* any kind of UML element (class, actor, ...), an instance uses the same graphic symbol as the type, but the designator string is underlined.

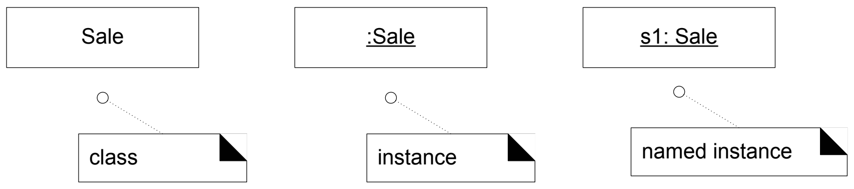


Figure 15.5 Class and instances.

Therefore, to show an instance of a class in an interaction diagram, the regular class box graphic symbol is used, but the name is underlined.

A name can be used to uniquely identify the instance. If none is used, note that a ":" precedes the class name.

Basic Message Expression Syntax

The UML has a standard syntax for message expressions:

```
return := message(parameter : parameterType) : returnType
```

Type information may be excluded if obvious or unimportant. For example:

```
spec := getProductSpect(id)
spec := getProductSpect(id:ItemID)
spec := getProductSpect(id:ItemID)    ProductSpecification
```

15.6 Basic Collaboration Diagram Notation

Links

A link is a connection path between two objects; it indicates some form of navigation and visibility between the objects is possible (see Figure 15.6). More formally, a link is an instance of an association. For example, there is a link—or path of navigation—from a *Register* to a *Sale,* along which messages may flow, such as the *makePayment* message.

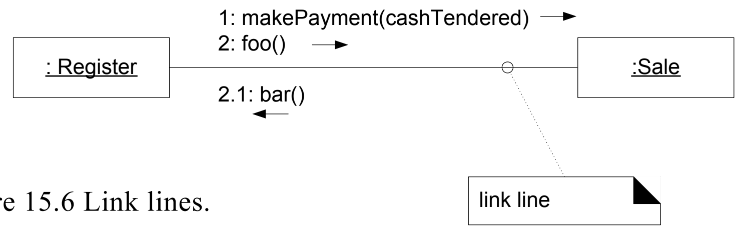


Figure 15.6 Link lines.

Note that multiple messages, and messages both ways, can flow along the same single link.

Messages

Each message between objects is represented with a message expression and small arrow indicating the direction of the message. Many messages may flow along this link (Figure 15.7). A sequence number is added to show the sequential order of messages in the current thread of control.

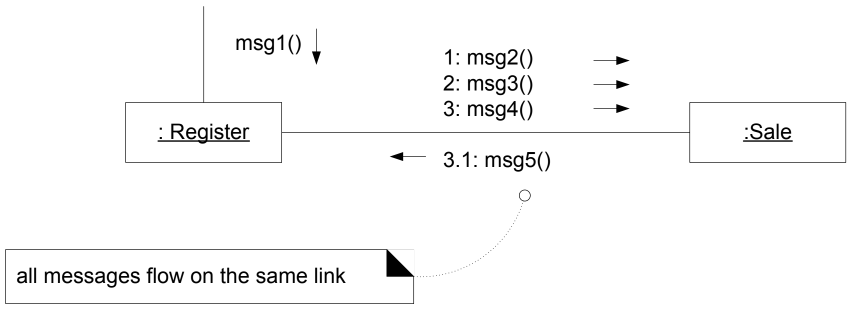


Figure 15.7 Messages.

Messages to "self" or "this"

A message can be sent from an object to itself (Figure 15.8). This is illustrated by a link to itself, with messages flowing along the link.

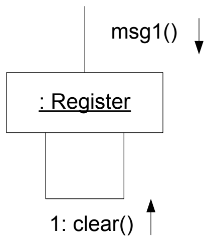


Figure 15.8 Messages to "this."

Creation of Instances

Any message can be used to create an instance, but there is a convention in the UML to use a message named *create* for this purpose. If another (perhaps less obvious) message name is used, the message may be annotated with a special feature called a UML stereotype, like so: *«create»*.

The *create* message may include parameters, indicating the passing of initial values. This indicates, for example, a constructor call with parameters in Java.

Furthermore, the UML property *{new}* may optionally be added to the instance box to highlight the creation.

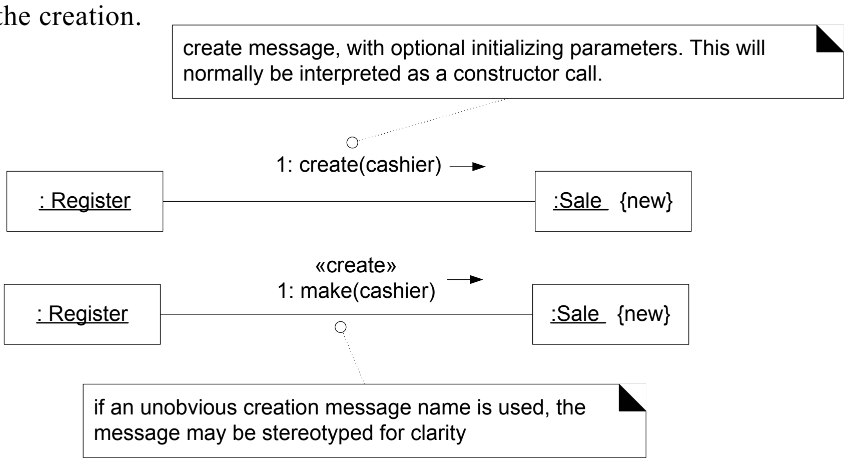
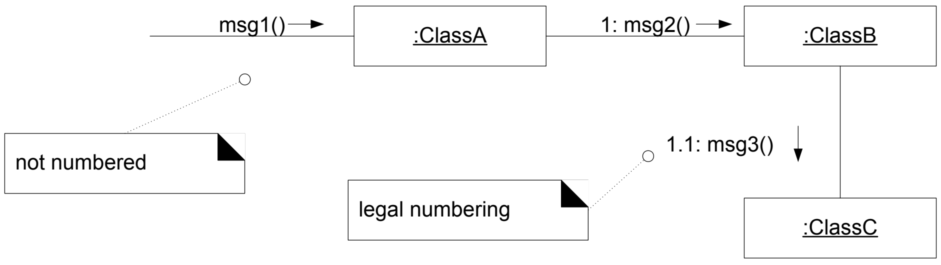

create message, with optional initializing parameters. This will normally be interpreted as a constructor call.
1: create(cashier)
: Register
:Sale {new}
«create»
1: make(cashier)
: Register
:Sale {new}
if an unobvious creation message name is used, the message may be stereotyped for clarity

Figure 15.9 Instance creation.

Message Number Sequencing

The order of messages is illustrated with **sequence numbers,** as shown in Figure 15.10. The numbering scheme is:

1. The first message is not numbered. Thus,*msg1()* is unnumbered.
2. The order and nesting of subsequent messages is shown with a legal num bering scheme in which nested messages have a number appended to them. Nesting is denoted by prepending the incoming message number to the out going message number.



In Figure 15.11 a more complex case is shown.

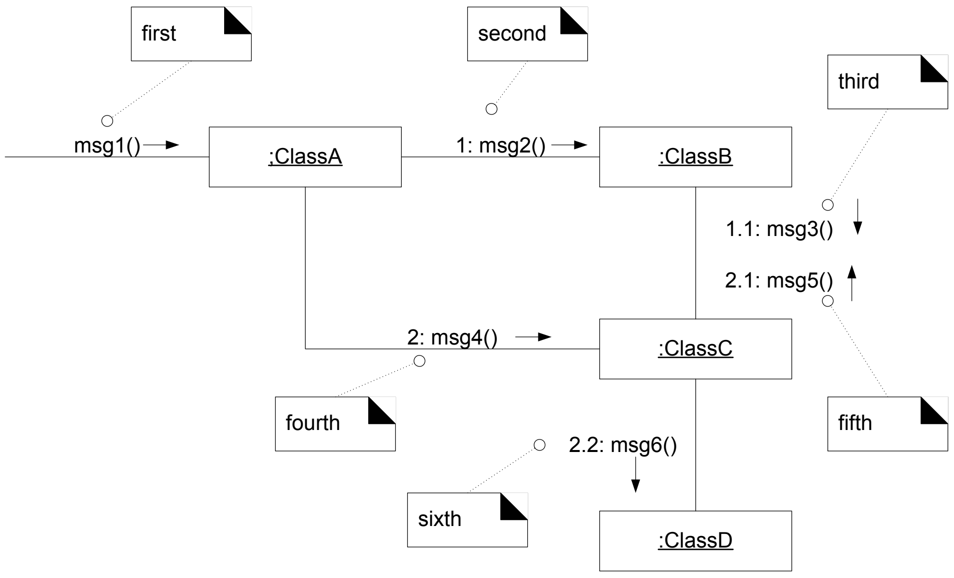


Figure 15.11 Complex sequence numbering.

Conditional Messages

A conditional message (Figure 15.12) is shown by following a sequence number with a conditional clause in square brackets, similar to an iteration clause. The message is only sent if the clause evaluates to *true*.

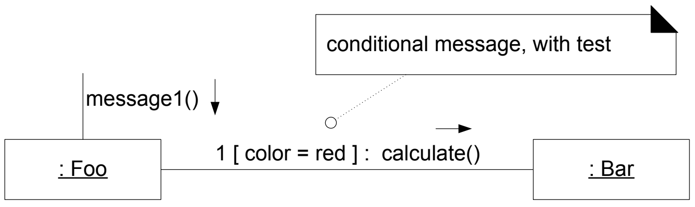

conditional message, with test
message1()
: Foo
1 [color = red] : calculate()
: Bar

Figure 15.12 Conditional message.

Mutually Exclusive Conditional Paths

The example in Figure 15.13 illustrates the sequence numbers with mutually exclusive conditional paths.

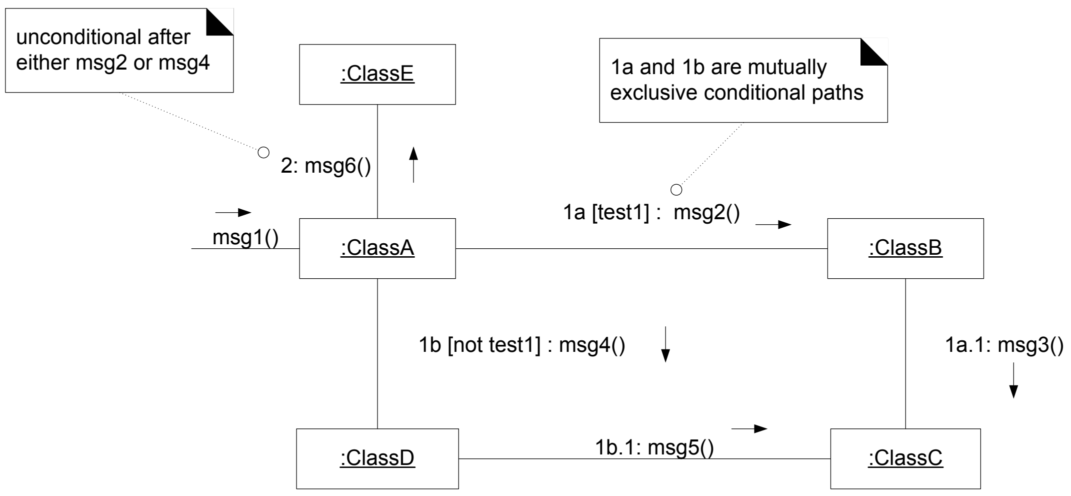


Figure 15.13 Mutually exclusive messages.

In this case it is necessary to modify the sequence expressions with a conditional path letter. The first letter used is a by convention. Figure 15.13 states that either *1a* or *1b* could execute after *msg1*. Both are sequence number 1 since either could be the first internal message.

Note that subsequent nested messages are still consistently prepended with their outer message sequence. Thus *Ib. 1* is nested message within *Ib.*

Iteration or Looping

Iteration notation is shown in Figure 15.14. If the details of the iteration clause are not important to the modeler, a simple '*' can be used.

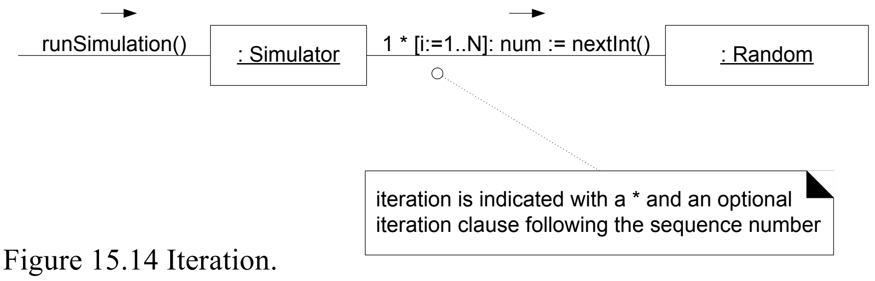

runSimulation()
: Simulator
1 * [i:=1..N]: num := nextInt()
: Random
iteration is indicated with a * and an optional iteration clause following the sequence number

Figure 15.14 Iteration.

Iteration Over a Collection (Multiobject)

A common algorithm is to iterate over all members of a collection (such as a list or map), sending a message to each. Often, some kind of iterator object is ultimately used, such as an implementation of *java.util.Iterator* or a C++ standard library iterator. In the UML, the term **multiobject** is used to denote a set of instances—a collection. In collaboration diagrams, this can be summarized as shown in Figure 15.15.

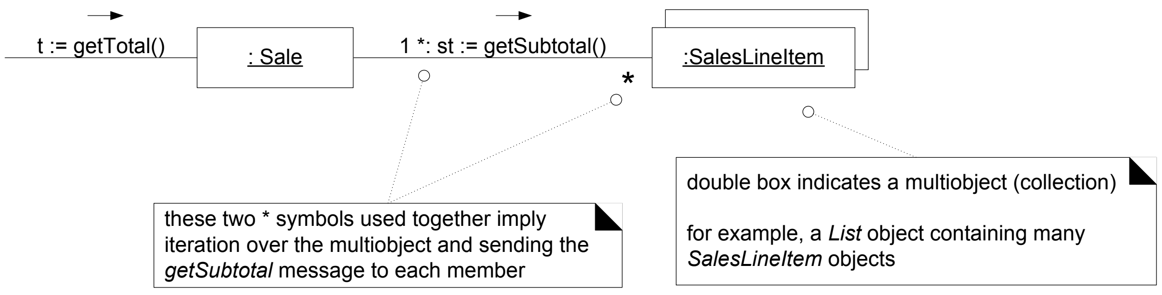

t := getTotal()
: Sale
1 *: st := getSubtotal()
:SalesLineItem
*
these two * symbols used together imply iteration over the multiobject and sending the *getSubtotal* message to each member
double box indicates a multiobject (collection)
for example, a *List* object containing many *SalesLineItem* objects

Figure 15.15 Iteration over a multiobject.

The "*" multiplicity marker at the end of the link is used to indicate that the message is being sent to each element of the collection, rather than being repeatedly sent to the collection object itself.

Messages to a Class Object

Messages may be sent to a class itself, rather than an instance, to invoke class or **static methods.** A message is shown to a class box whose name is not underlined, indicating the message is being sent to a class rather than an instance (see Figure 15.16).

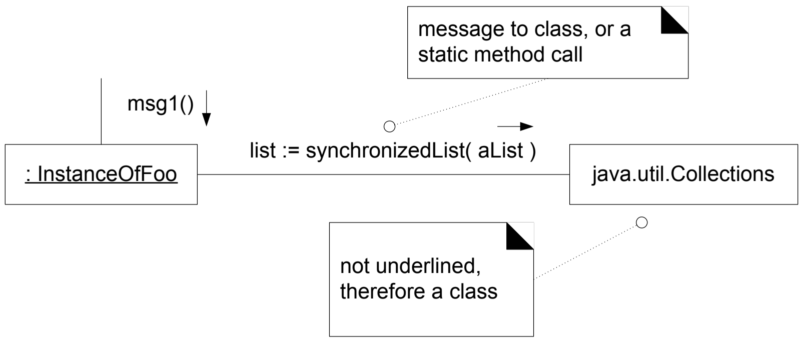

message to class, or a static method call
msg1()
: InstanceOfFoo
list := synchronizedList(aList)
java.util.Collections
not underlined, therefore a class

Figure 15.16 Messages to a class object (static method invocation).

Consequently, it is important to be consistent in underlining your instance names when an instance is intended, otherwise messages to instances versus classes may be incorrectly interpreted.

15.7 Basic Sequence Diagram Notation

Links

Unlike collaboration diagrams, sequence diagrams do not show links.

Messages

Each message between objects is represented with a message expression on an arrowed line between the objects (see Figure 15.17). The time ordering is organized from top to bottom.

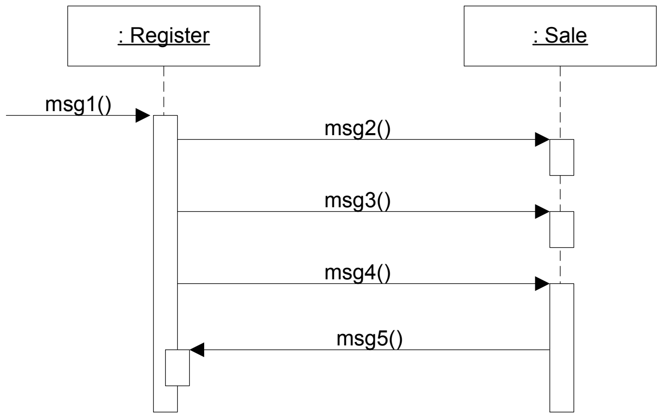


Figure 15.17 Messages and focus of control with activation boxes.

Focus of Control and Activation Boxes

As illustrated in Figure 15.17, sequence diagrams may also show the focus of control (that is, in a regular blocking call, the operation is on the call stack) using an **activation box.** The box is optional, but commonly used by UML practitioners.

Illustrating Returns

A sequence diagram may optionally show the return from a message as a dashed open-arrowed line at the end of an activation box (see Figure 15.18). Many practitioners exclude them. Some annotate the return line to describe what is being returned (if anything) from the message.

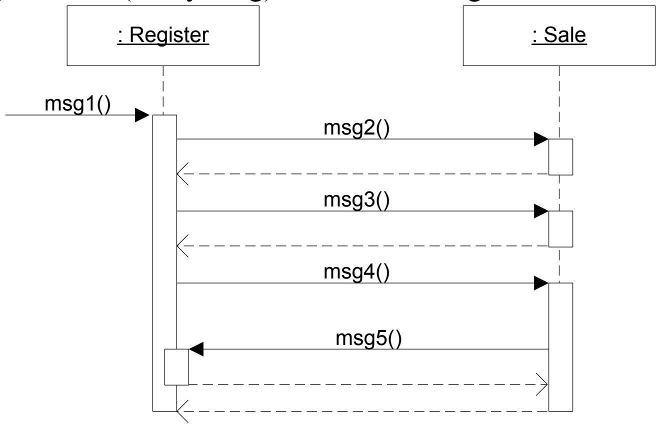


Figure 15.18 Showing returns.

Messages to "self" or "this"

A message can be illustrated as being sent from an object to itself by using a nested activation box (see Figure 15.19).

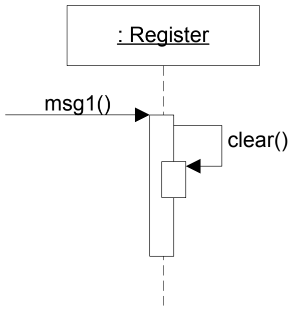


Figure 15.19 Messages to "this."

Creation of Instances

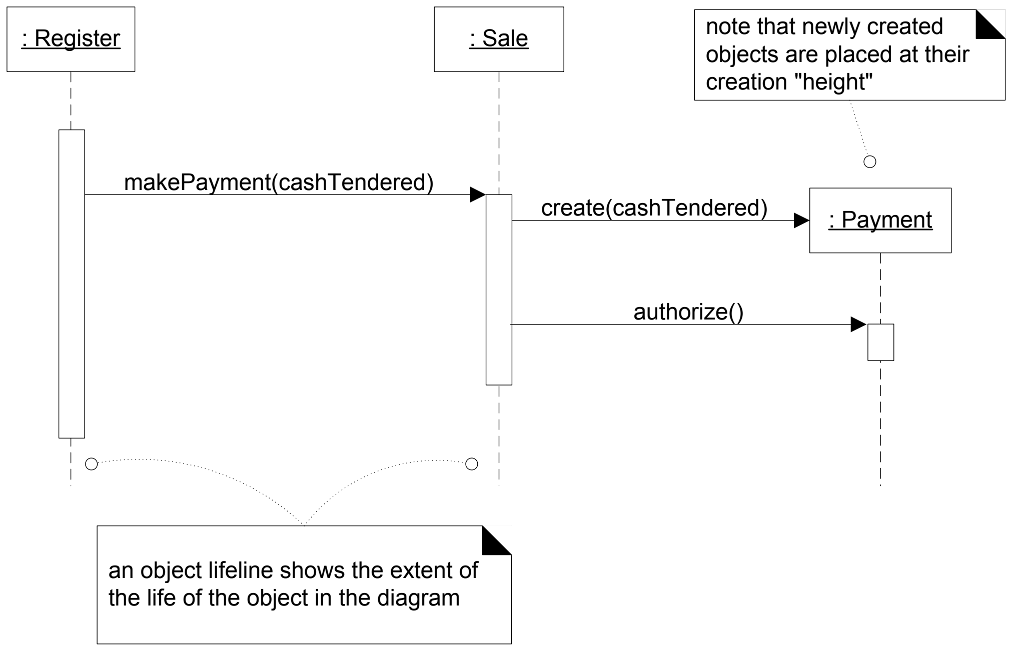

: Register
: Sale
note that newly created objects are placed at their creation "height"
makePayment(cashTendered)
create(cashTendered)
: Payment
authorize()
an object lifeline shows the extent of the life of the object in the diagram

Figure 15.20 Instance creation and object lifelines.

Object Lifelines and Object Destruction

Figure 15.20 also illustrates **object lifelines**—the vertical dashed lines underneath the objects. These indicate the extent of the life of the object in the diagram. In some circumstances it is desirable to show explicit destruction of an object (as in C++, which does not have garbage collection); the UML lifeline notation provides a way to express this destruction (see Figure 15.21).

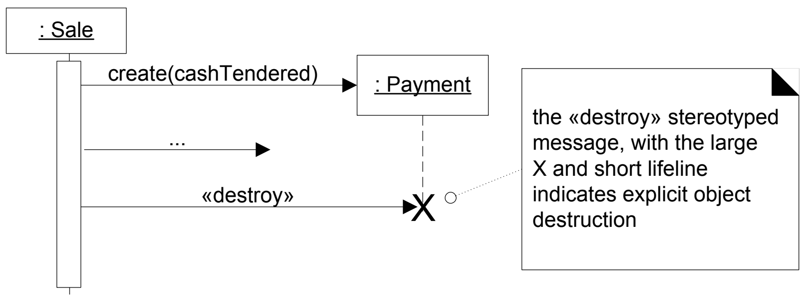

: Sale
create(cashTendered)
: Payment
...
«destroy»
the «destroy» stereotyped message, with the large X and short lifeline indicates explicit object destruction

Figure 15.21 Object destruction

Conditional Messages

A conditional message is shown in Figure 15.22.

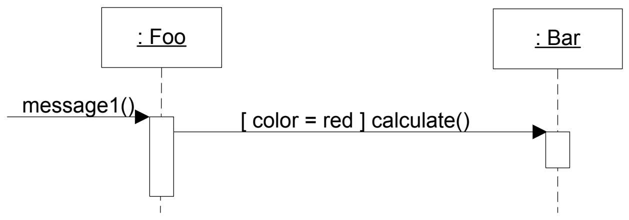


Figure 15.22 A conditional message.

Mutually Exclusive Conditional Messages

The notation for this case is a kind of angled message line emerging from a common point, as illustrated in Figure 15.23.

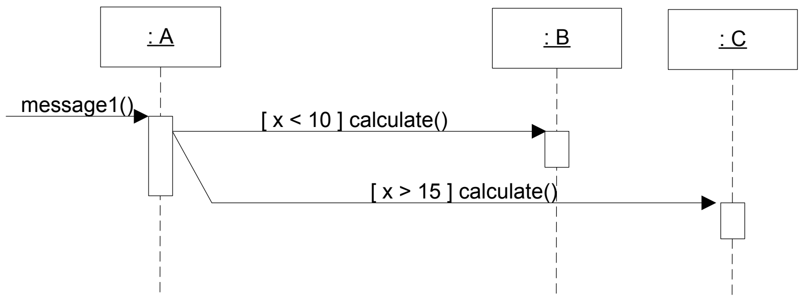


Figure 15.23 Mutually exclusive conditional messages.

Iteration for a Single Message

Iteration notation for one message is shown in Figure 15.24.

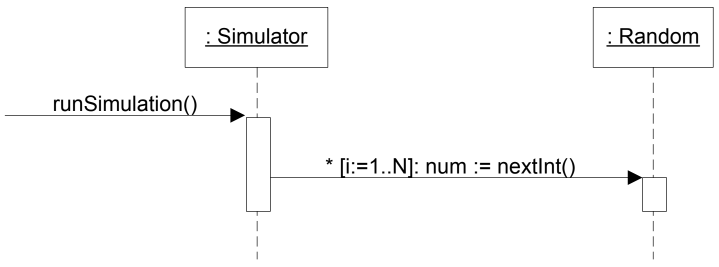


Figure 15.24 Iteration for one message.

Iteration of a Series of Messages

Notation to indicate iteration around a series of messages is shown in Figure 15.25.

Iteration Over a Collection (Multiobject)

In sequence diagrams, iteration over a collection is shown in Figure 15.26.

With collaboration diagrams the UML specifies a '*' multiplicity marker at the end of the role (next to the multiobject) to indicate sending a message to each element rather than repeatedly to the collection itself. However, the UML does not specify how to indicate this with sequence diagrams.

Messages to Class Objects

As in a collaboration diagram, class or static method calls are shown by not underlining the name of the classifier, which signifies a class object rather than an instance (see Figure 15.27).

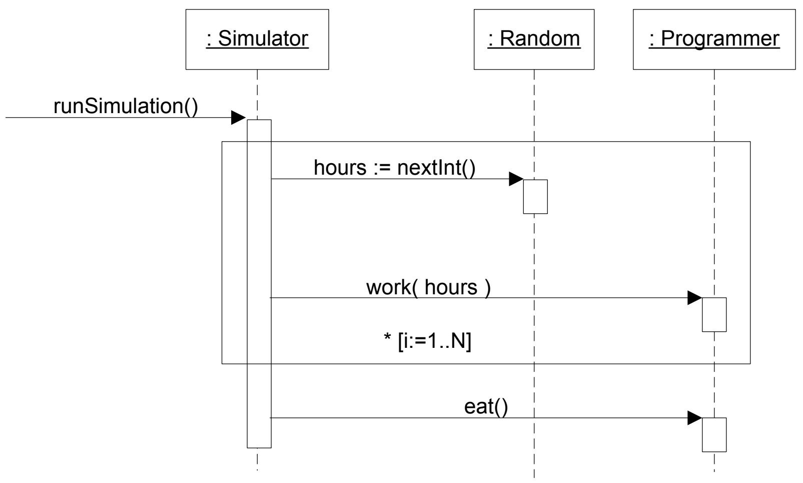


Figure 15.25 Iteration for a sequence of

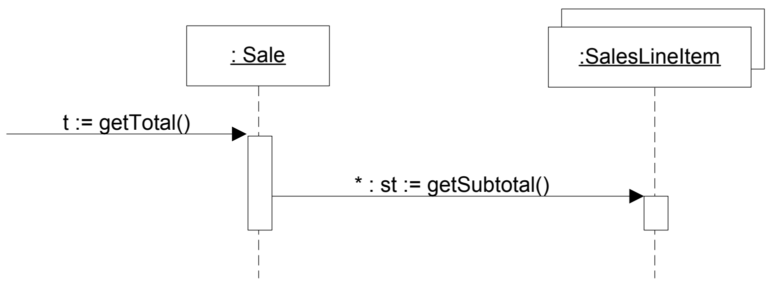


Figure 15.26 Iteration over a multiobject

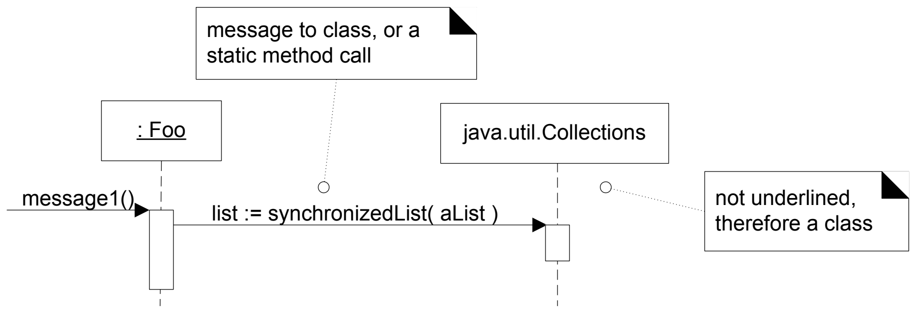

message to class, or a static method call
: Foo
java.util.Collections
message1()
list := synchronizedList(aList)
not underlined, therefore a class

Figure 15.27 Invoking class or static methods