Chapter 15

UML INTERACTION DIAGRAMS

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

—Jeff Valdez

Objectives

- Provide a reference for frequently used UML interaction diagram notation—sequence and communication diagrams.

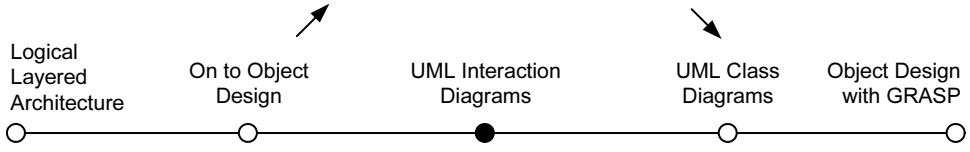
Introduction

The UML includes **interaction diagrams** to illustrate how objects interact via messages. They are used for **dynamic object modeling**. There are two common types: sequence and communication interaction diagrams. This chapter introduces the notation—view it as a *reference* to skim through—while subsequent chapters focus on a more important question: What are key principles in OO design?

In the following chapters, interaction diagrams are applied to help explain and demonstrate object design. Hence, it's useful to at least skim these examples before moving on.

What's Next?

Having introduced OO design (OOD), this chapter summarizes UML interaction diagrams for dynamic OO design. The next chapter summarizes UML class diagrams for static OO design.



15.1 Sequence and Communication Diagrams

The term **interaction diagram** is a generalization of two more specialized UML diagram types:

- sequence diagrams
- communication diagrams

Both can express similar interactions.

A related diagram is the **interaction overview diagram**; it provides a big-picture overview of how a set of interaction diagrams are related in terms of logic and process-flow. However, it's new to UML 2, and so it's too early to tell if it will be practically useful.

Sequence diagrams are the more notationally rich of the two types, but communication diagrams have their use as well, especially for wall sketching. Throughout the book, both types will be used to emphasize the flexibility in choice.

Sequence diagrams illustrate interactions in a kind of fence format, in which each new object is added to the right, as shown in Figure 15.1.

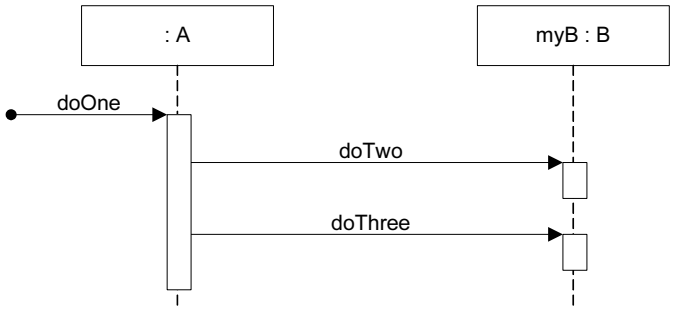


Figure 15.1 Sequence diagram.

What might this represent in code?[1] Probably, that class *A* has a method named *doOne* and an attribute of type *B*. Also, that class *B* has methods named *doTwo* and *doThree*. Perhaps the partial definition of class *A* is:

```
public class A
{
private B myB = new B();

public void doOne()
{
    myB.doTwo();
    myB.doThree();
}
// …
}
```

1. Code mapping or generation rules will vary depending on the OO language.

Communication diagrams illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram (the essence of their wall sketching advantage), as shown in Figure 15.2.

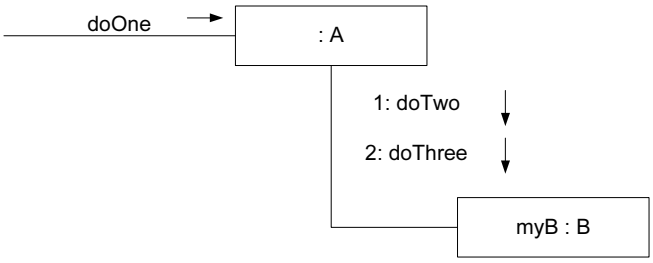


Figure 15.2 Communication diagram.

What are the Strengths and Weaknesses of Sequence vs. Communication Diagrams?

Each diagram type has advantages, and modelers have idiosyncratic preference—there isn't an absolutely "correct" choice. However, UML tools usually emphasize sequence diagrams, because of their greater notational power.

Sequence diagrams have some advantages over communication diagrams. Perhaps first and foremost, the UML specification is more sequence diagram centric—more thought and effort has been put into the notation and semantics. Thus, tool support is better and more notation options are available. Also, it is easier to see the call-flow sequence with sequence diagrams—simply read top to bottom. With communication diagrams we must read the sequence numbers, such as "1:" and "2:". Hence, sequence diagrams are excellent for documentation or to easily read a reverse-engineered call-flow sequence, generated from source code with a UML tool.

three ways to use UML p. 11

But on the other hand, communication diagrams have advantages when applying "UML as sketch" to draw on walls (an Agile Modeling practice) because they are *much* more space-efficient. This is because the boxes can be easily placed or erased anywhere—horizontal or vertical. Consequently as well, *modifying* wall sketches is easier with communication diagrams—it is simple (during creative high-change OO design work) to erase a box at one location, draw a new one elsewhere, and sketch a line to it. In contrast, new objects in a sequence diagrams must always be added to the right edge, which is limiting as it quickly consumes and exhausts right-edge space on a page (or wall); free space in the vertical dimension is not efficiently used. Developers doing sequence diagrams on walls rapidly feel the drawing pain when contrasted with communication diagrams.

Likewise, when drawing diagrams that are to be published on narrow pages (like this book), communication diagrams have the advantage over sequence diagrams of allowing *vertical* expansion for new objects—much more can be packed into a small visual space.

Type	Strengths	Weaknesses
sequence	clearly shows sequence or time ordering of messages large set of detailed notation options	forced to extend to the right when adding new objects; consumes horizontal space
communication	space economical—flexibility to add new objects in two dimensions	more difficult to see sequence of messages fewer notation options

Example Sequence Diagram: makePayment

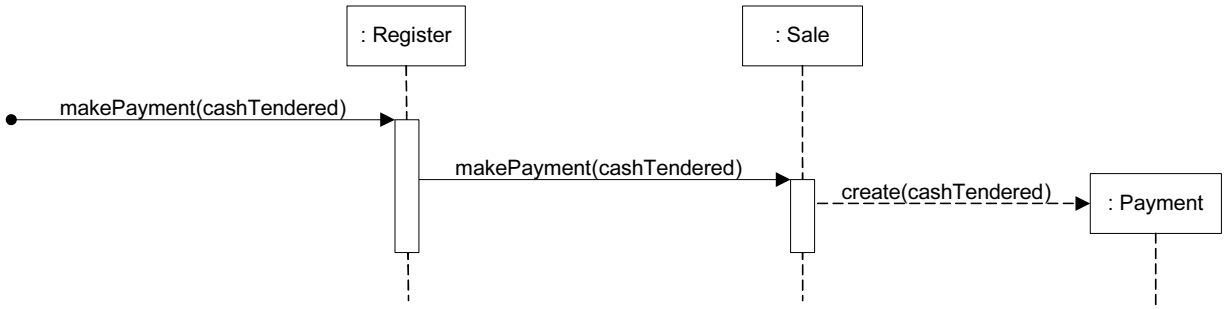


Figure 15.3 Sequence diagram.

The sequence 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*.

From reading Figure 15.3, what might be some related code for the *Sale* class and its *makePayment* method?

```
public class Sale
{
private Payment payment;

public void makePayment( Money cashTendered )
{
    payment = new Payment( cashTendered );
    //…
}
// …
}
```

Example Communication Diagram: makePayment

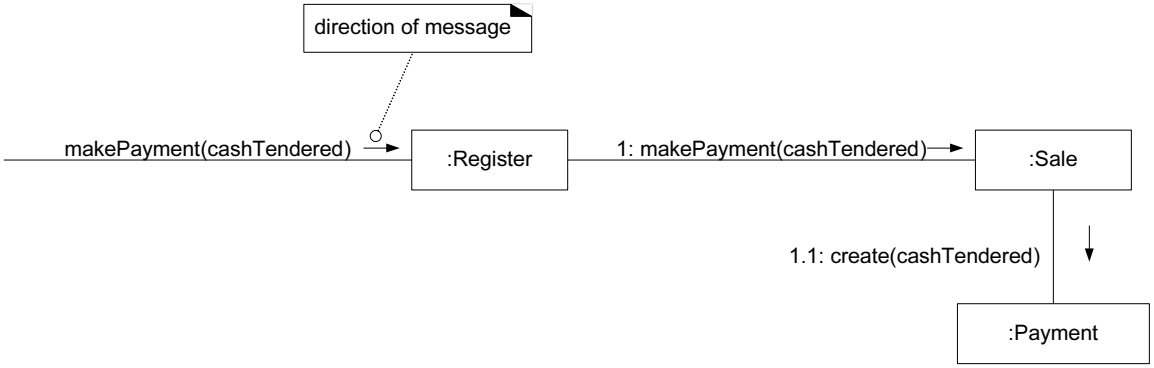


Figure 15.4 Communication diagram.

The communication diagram shown in Figure 15.3 has the same intent as the prior sequence diagram.

15.2 Novice UML Modelers Don't Pay Enough Attention to Interaction Diagrams!

Most UML novices are aware of class diagrams and usually think they are the only important diagram in OO design. Not true!

Although the static-view class diagrams are indeed useful, the dynamic-view interaction diagrams—or more precisely, *acts* of dynamic interaction modeling—are incredibly valuable.

> *Guideline*
>
> Spend time doing *dynamic* object modeling with interaction diagrams, not just static object modeling with class diagrams.

Why? Because it's when we have to think through the concrete details of what messages to send, and to whom, and in what order, that the "rubber hits the road" in terms of thinking through the true OO design details.

15.3 Common UML Interaction Diagram Notation

Illustrating Participants with Lifeline Boxes

In the UML, the boxes you've seen in the prior sample interaction diagrams are called **lifeline** boxes. Their precise UML definition is subtle, but informally they represent the **participants** in the interaction—related parts defined in the context of some structure diagram, such as a class diagram. It is not precisely accurate to say that a lifeline box equals an instance of a class, but informally and practically, the participants will often be interpreted as such. Therefore, in this text I'll often write something like "the lifeline representing a *Sale* instance," as a convenient shorthand. See Figure 15.5 for common cases of notation.

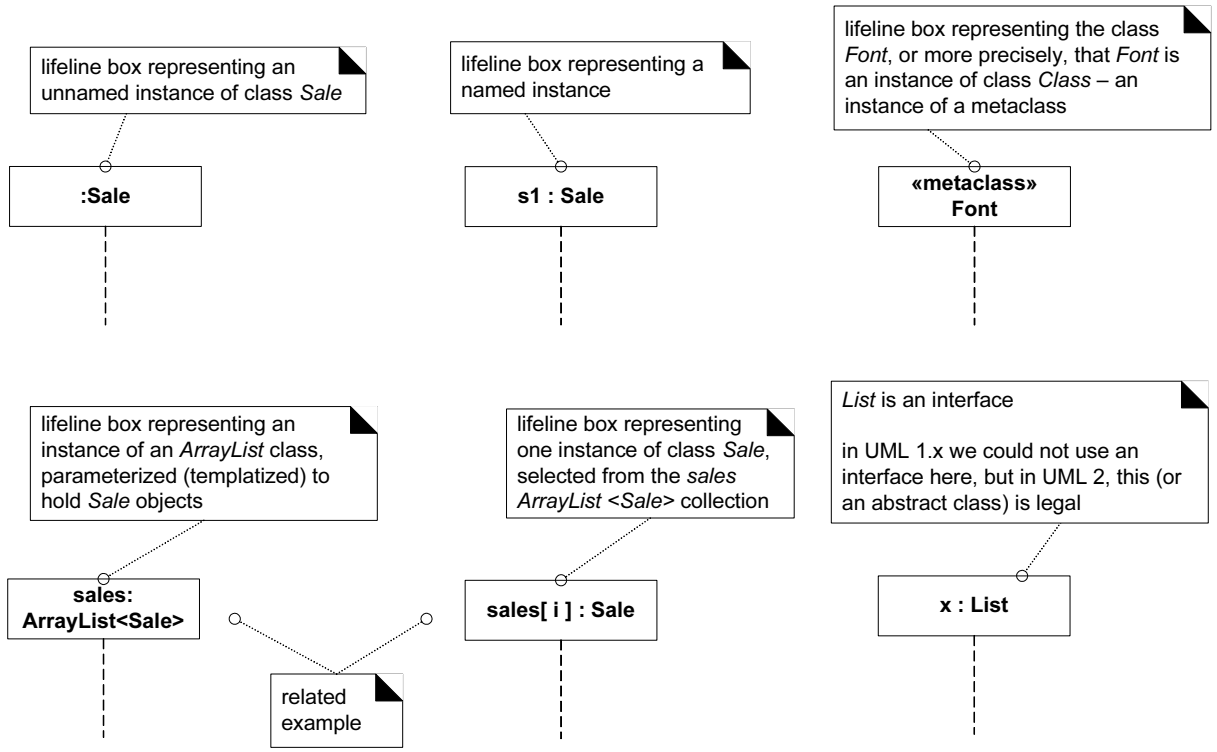


Figure 15.5 Lifeline boxes to show participants in interactions.

Basic Message Expression Syntax

Interaction diagrams show messages between objects; the UML has a standard syntax for these message expressions:[2]

2. An alternate syntax, such as C# or Java, is acceptable—and supported by UML tools.

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

Parentheses are usually excluded if there are no parameters, though still legal.

Type information may be excluded if obvious or unimportant.

For example:

```
initialize(code)
initialize
d = getProductDescription(id)
d = getProductDescription(id:ItemID)
d = getProductDescription(id:ItemID) : ProductDescription
```

Singleton Objects

Singleton p. 442

In the world of OO design patterns, there is one that is especially common, called the **Singleton** pattern. It is explained later, but an implication of the pattern is that there is only *one* instance of a class instantiated—never two. In other words, it is a "singleton" instance. In a UML interaction diagram (sequence or communication), such an object is marked with a '1' in the upper right corner of the lifeline box. It implies that the Singleton pattern is used to gain visibility to the object—the meaning of that won't be clear at this time, but will be upon reading its description on p. 442. See Figure 15.6.

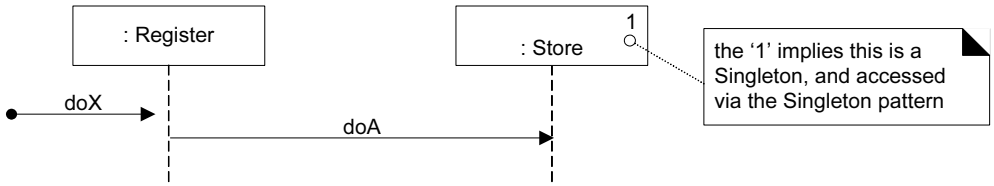


Figure 15.6 Singletons in interaction diagrams.

15.4 Basic Sequence Diagram Notation

Lifeline Boxes and Lifelines

lifeline boxes p. 226

In contrast to communication diagrams, in sequence diagrams the lifeline boxes include a vertical line extending below them—these are the actual lifelines. Although virtually all UML examples show the lifeline as dashed (because of UML 1 influence), in fact the UML 2 specification says it may be solid *or* dashed.

Messages

Each (typical synchronous) message between objects is represented with a message expression on a *filled-arrowed*[3] solid line between the vertical lifelines (see Figure 15.7). The time ordering is organized from top to bottom of lifelines.

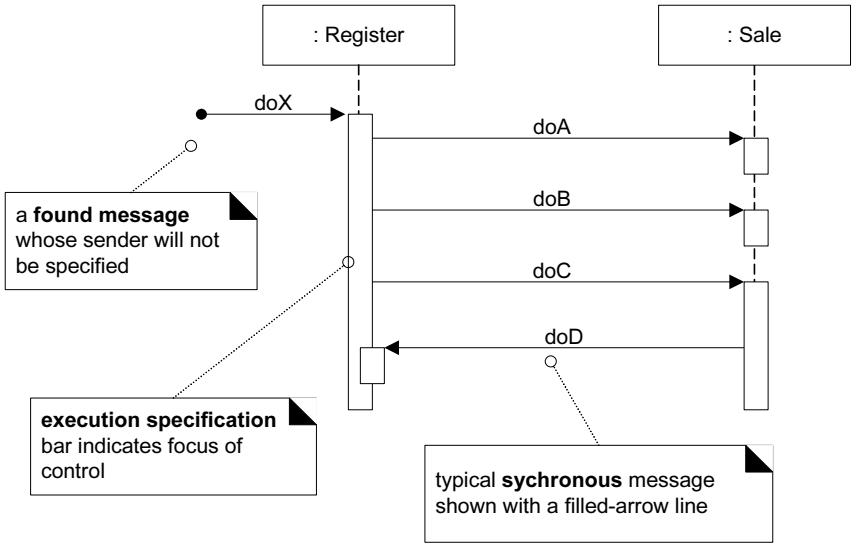


Figure 15.7 Messages and focus of control with execution specification bar.

In the example of Figure 15.7 the starting message is called a **found message** in the UML, shown with an opening solid ball; it implies the sender will not be specified, is not known, or that the message is coming from a random source. However, by convention a team or tool may ignore showing this, and instead use a regular message line without the ball, intending by convention it is a found message.[4]

Focus of Control and Execution Specification Bars

As illustrated in Figure 15.7, sequence diagrams may also show the focus of control (informally, in a regular blocking call, the operation is on the call stack) using an **execution specification** bar (previously called an **activation bar** or simply an **activation** in UML 1). The bar is optional.

Guideline: Drawing the bar is more common (and often automatic) when using a UML CASE tool, and less common when wall sketching.

3. An open message arrow means an asynchronous message in an interaction diagram.
4. Therefore, many of the book examples won't bother with the found message notation.

Illustrating Reply or Returns

There are two ways to show the return result from a message:

1. Using the message syntax *returnVar = message(parameter)*.
2. Using a reply (or return) message line at the end of an activation bar.

Both are common in practice. I prefer the first approach when sketching, as it's less effort. If the reply line is used, the line is normally labelled with an arbitrary description of the returning value. See Figure 15.8.

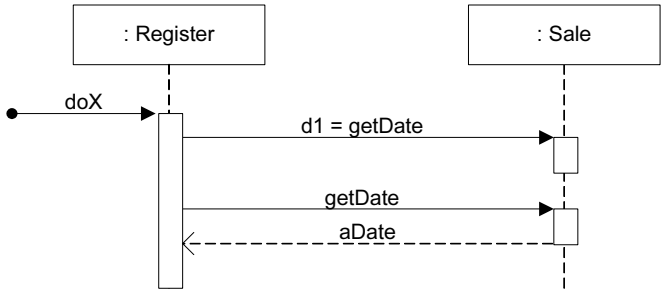


Figure 15.8 Two ways to show a return result from a message.

Messages to "self" or "this"

You can show a message being sent from an object to itself by using a nested activation bar (see Figure 15.9).

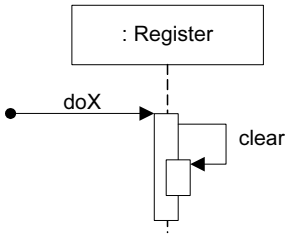


Figure 15.9 Messages to "this."

Creation of Instances

Object creation notation is shown in Figure 15.10. Note the UML-mandated *dashed* line.[5] The arrow is filled if it's a regular synchronous message (such as implying invoking a Java constructor), or open (stick arrow) if an asynchronous

5. I see no value in requiring a *dashed* line, but it's in the spec… Many author examples use a solid line, as early draft versions of the spec did as well.

call. The message name *create* is not required—anything is legal—but it's a UML idiom.

The typical interpretation (in languages such as Java or C#) of a *create* message on a dashed line with a filled arrow is "invoke the *new* operator and call the constructor".

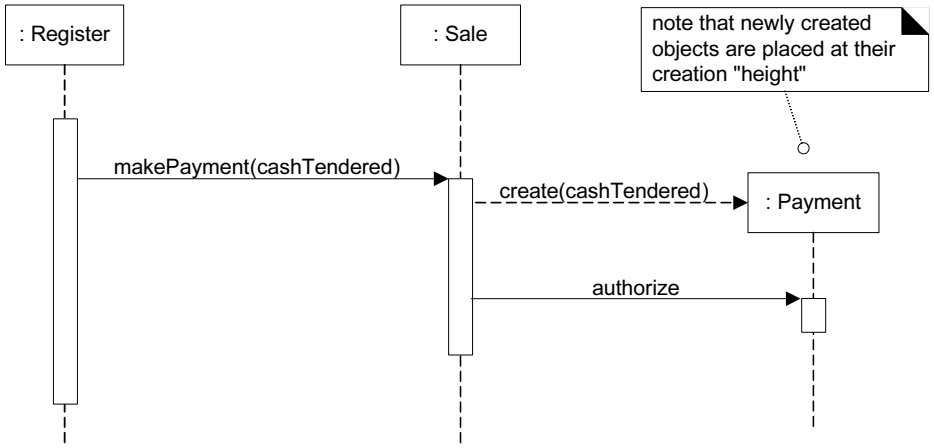


Figure 15.10 Instance creation and object lifelines.

Object Lifelines and Object Destruction

In some circumstances it is desirable to show explicit destruction of an object. For example, when using C++ which does not have automatic garbage collection, or when you want to especially indicate an object is no longer usable (such as a closed database connection). The UML lifeline notation provides a way to express this destruction (see Figure 15.11).

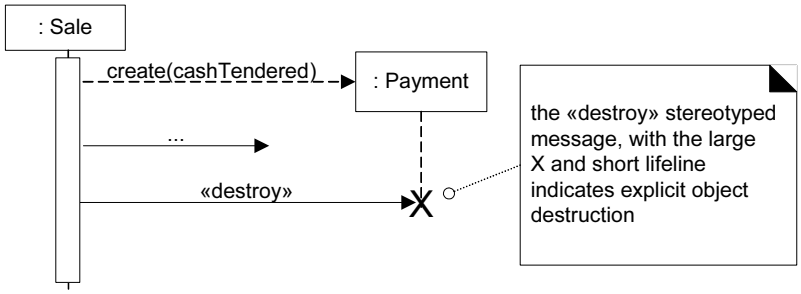


Figure 15.11 Object destruction.

Diagram Frames in UML Sequence Diagrams

To support conditional and looping constructs (among many other things), the UML uses **frames**.[6] Frames are regions or fragments of the diagrams; they have

an operator or label (such as *loop*) and a guard[7] (conditional clause). See Figure 15.12.

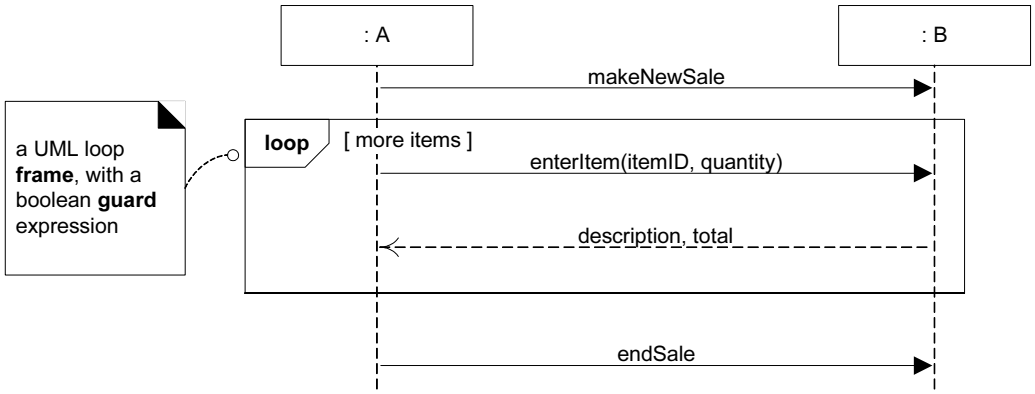


Figure 15.12 Example UML frame.

The following table summarizes some common frame operators:

Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write *loop(n)* to indicate looping n times. There is discussion that the specification will be enhanced to define a *FOR* loop, such as *loop(i, 1, 10)*
opt	Optional fragment that executes if guard is true.
par	Parallel fragments that execute in parallel.
region	Critical region within which only one thread can run.

Looping

The LOOP frame notation to show looping is shown in Figure 15.12.

Conditional Messages

An OPT frame is placed around one or more messages. Notice that the guard is

6. Also called **diagram frames** or **interaction frames**.
7. The *[boolean test]* guard should be placed *over* the lifeline to which it belongs.

placed *over* the related lifeline. See Figure 15.13.

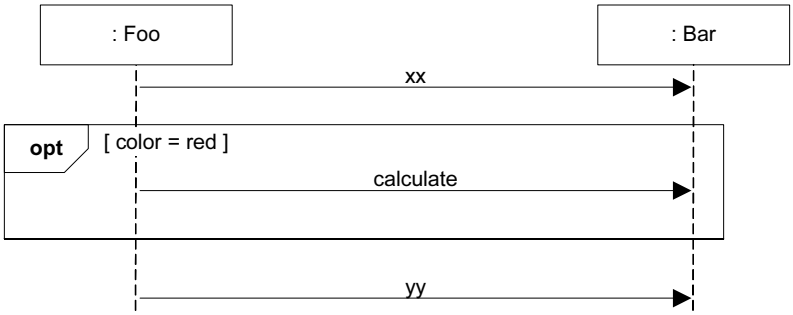


Figure 15.13 A conditional message.

Conditional Messages in UML 1.x Style—Still Useful?

The UML 2.x notation to show a single conditional message is heavyweight, requiring an entire OPT frame box around one message (see Figure 15.13). The older UML 1.x notation for *single* conditional messages in sequence diagrams is not legal in UML 2, but so simple that especially when sketching it will probably be popular for years to come. See Figure 15.14.

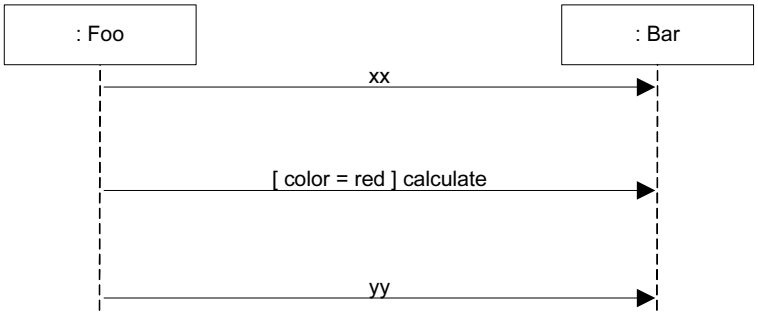


Figure 15.14 A conditional message in UML 1.x notation—a simple style.

Guideline: Use UML 1 style only for simple single messages when sketching.

Mutually Exclusive Conditional Messages

An ALT frame is placed around the mutually exclusive alternatives. See Figure 15.15.

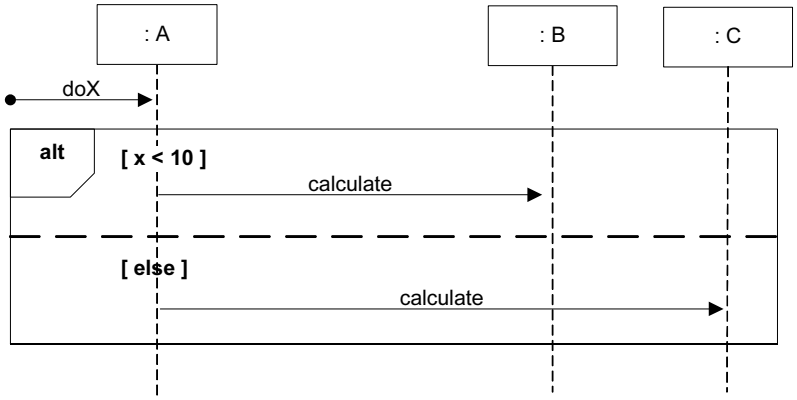


Figure 15.15 Mutually exclusive conditional messages.

Iteration Over a Collection

A common algorithm is to iterate over all members of a collection (such as a list or map), sending the same 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, although in the sequence diagram that low-level "mechanism" need not be shown in the interest of brevity or abstraction.

At the time of this writing, the UML specification did not (and may never) have an official idiom for this case. Two alternatives are shown—reviewed with the leader of the UML 2 interaction specification—in Figure 15.16 and Figure 15.17.

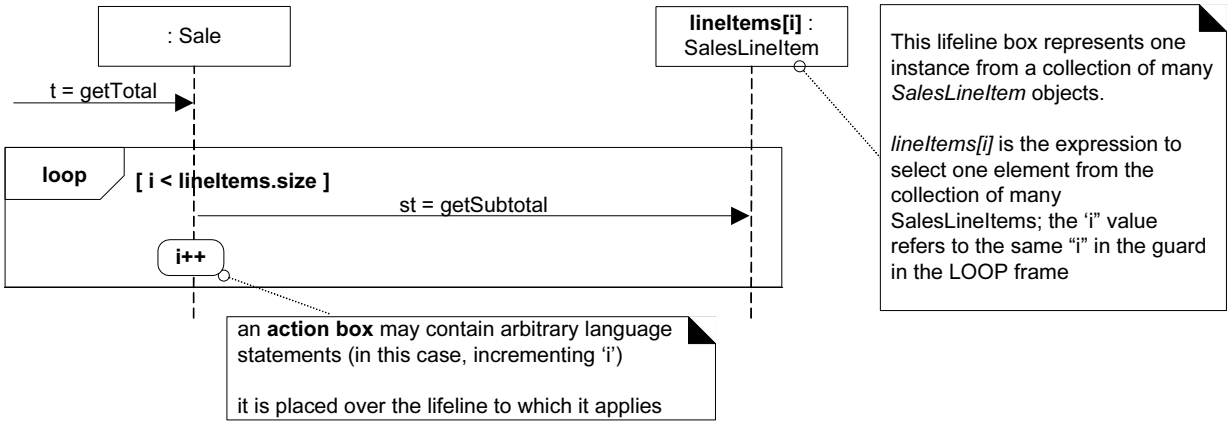


Figure 15.16 Iteration over a collection using relatively explicit notation.

Note the **selector** expression *lineItems[i]* in the lifeline of Figure 15.16. The selector expression is used to select one object from a group. Lifeline participants should represent one object, not a collection.

In Java, for example, the following code listing is a possible implementation that maps the explicit use of the incrementing variable *i* in Figure 15.16 to an idiomatic solution in Java, using its enhanced *for* statement (C# has the same).

```
public class Sale
{
private List<SalesLineItem> lineItems =
                            new ArrayList<SalesLineItem>();


public Money getTotal()
{
    Money total = new Money();
    Money subtotal = null;

    for ( SalesLineItem lineItem : lineItems )
    {
        subtotal = lineItem.getSubtotal();
        total.add( subtotal );
    }
    return total;
}
// …
}
```

Another variation is shown in Figure 15.17; the intent is the same, but details are excluded. A team or tool could agree on this simple style by convention to imply iteration over all the collection elements.[8]

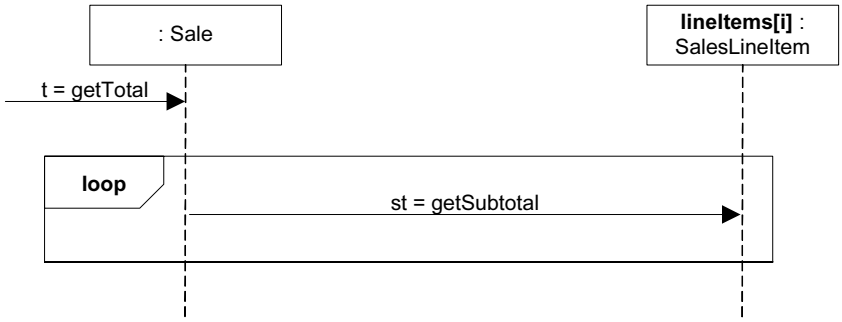


Figure 15.17 Iteration over a collection leaving things more implicit.

8. I use this style later in the book.

Nesting of Frames

Frames can be nested. See Figure 15.18.

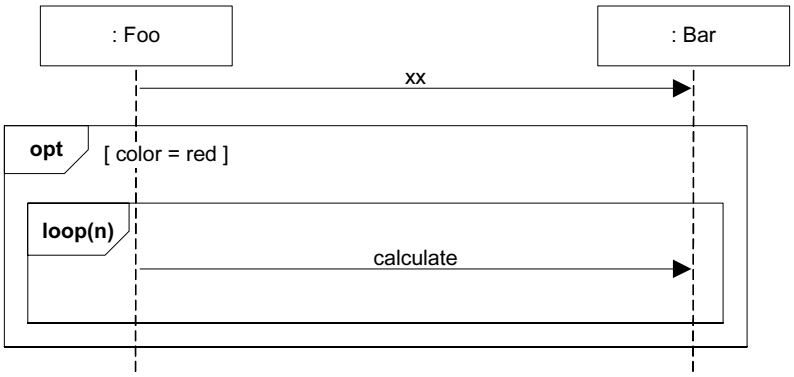


Figure 15.18 Nesting of frames.

How to Relate Interaction Diagrams?

Figure 15.19 illustrates probably better than words. An **interaction occurrence** (also called an **interaction use**) is a reference to an interaction within another interaction. It is useful, for example, when you want to simplify a diagram and factor out a portion into another diagram, or there is a reusable interaction occurrence. UML tools take advantage of them, because of their usefulness in relating and linking diagrams.

They are created with two related frames:

- a frame around an entire sequence diagram[9], labeled with the tag **sd** and a name, such as *AuthenticateUser*
- a frame tagged **ref**, called a **reference**, that refers to another named sequence diagram; it is the actual interaction occurrence

Interaction overview diagrams also contain a set of reference frames (interaction occurrences). These diagrams organized references into a larger structure of logic and process flow.

9. Interaction occurrences and *ref* frames can also be used for *communication* diagrams.

Guideline: Any sequence diagram can be surrounded with an *sd* frame, to name it. Frame and name one when you want to refer to it using a *ref* frame.

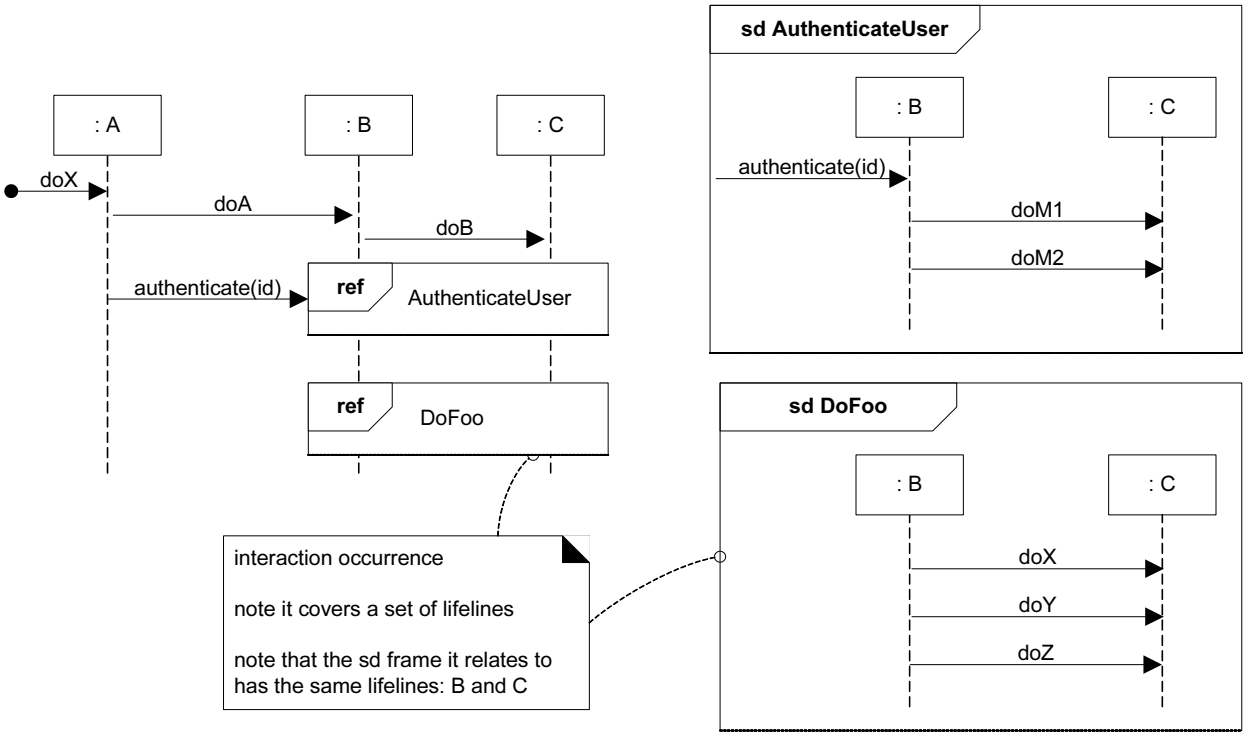


Figure 15.19 Example interaction occurrence, *sd* and *ref* frames.

Messages to Classes to Invoke Static (or Class) Methods

You can show class or static method calls by using a lifeline box label that indicates the receiving object is a class, or more precisely, an *instance* of a **metaclass** (see Figure 15.20).

What do I mean? For example, in Java and Smalltalk, all classes are conceptually or literally *instances* of class *Class*; in .NET classes are instances of class *Type*. The classes *Class* and *Type* are **metaclasses**, which means their instances are themselves classes. A specific class, such as class *Calendar*, is itself an

instance of class *Class*. Thus, class *Calendar* is an instance of a metaclass! It may help to drink some beer before trying to understand this.

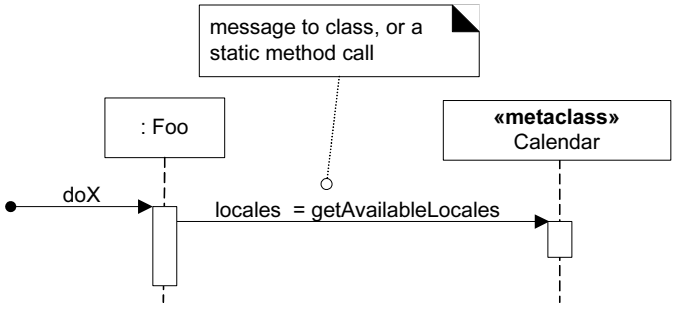


Figure 15.20 Invoking class or static methods; showing a class object as an instance of a metaclass.

In code, a likely implementation is:

```
public class Foo
{
public void doX()
{
    // static method call on class Calendar
    Locale[] locales = Calendar.getAvailableLocales();
    // …
}
// …
}
```

Polymorphic Messages and Cases

Polymorphism is fundamental to OO design. How to show it in a sequence diagram? That's a common UML question. One approach is to use multiple sequence diagrams—one that shows the polymorphic message to the abstract superclass or interface object, and then separate sequence diagrams detailing each polymorphic case, each starting with a *found* polymorphic message. Figure 15.21 illustrates.

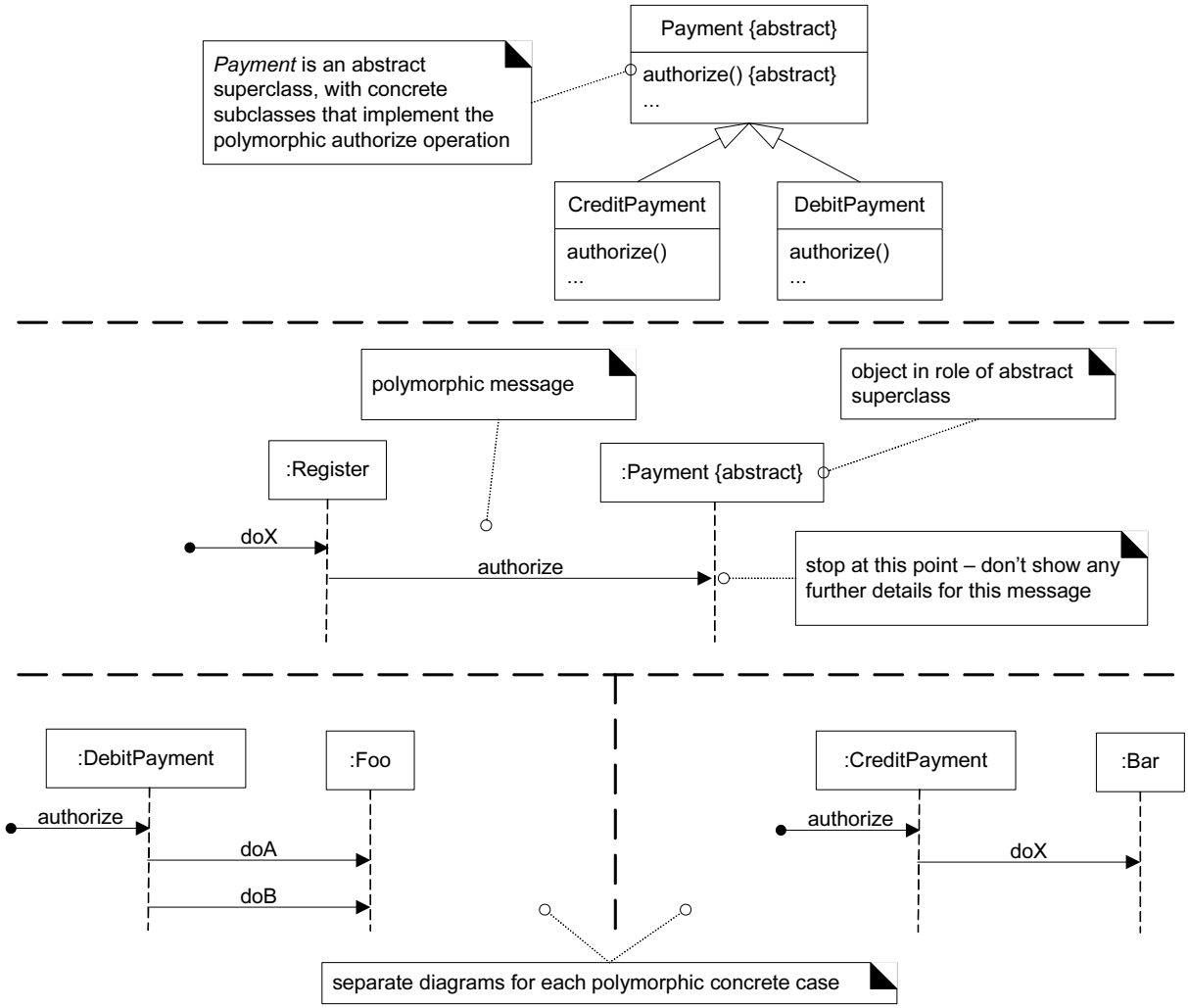


Figure 15.21 An approach to modeling polymorphic cases in sequence diagrams.

Asynchronous and Synchronous Calls

An **asynchronous message** call does not wait for a response; it doesn't *block*. They are used in multi-threaded environments such as .NET and Java so that new **threads** of execution can be created and initiated. In Java, for example, you may think of the *Thread.start* or *Runnable.run* (called by *Thread.start*) message as the asynchronous starting point to initiate execution on a new thread.

The UML notation for asynchronous calls is a stick arrow message; regular synchronous (blocking) calls are shown with a filled arrow (see Figure 15.22).

Guideline

This arrow difference is subtle. And when wall sketching UML, it is common to use a stick arrow to mean a synchronous call because it's easier to draw. Therefore, when reading a UML interaction diagram don't assume the shape of the arrow is correct!

An object such as the *Clock* in Figure 15.22 is also known as an **active object**—each instance runs on and controls its own thread of execution. In the UML, it may be shown with double vertical lines on the left and right sides of the lifeline box. The same notation is used for an **active class** whose instances are active objects.

active class p. 269

a stick arrow in UML implies an asynchronous call

a filled arrow is the more common synchronous call

In Java, for example, an asynchronous call may occur as follows:

```
// Clock implements the Runnable interface
Thread t = new Thread( new Clock() );
t.start();
```

the asynchronous *start* call always invokes the *run* method on the *Runnable* (*Clock*) object

to simplify the UML diagram, the *Thread* object and the *start* message may be avoided (they are standard "overhead"); instead, the essential detail of the *Clock* creation and the *run* message imply the asynchronous call

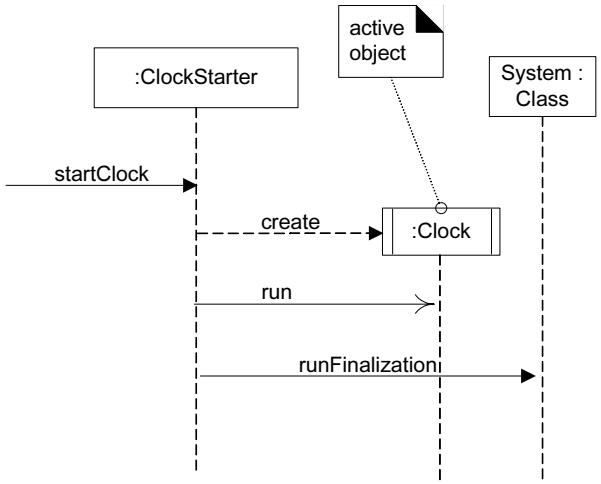


Figure 15.22 Asynchronous calls and active objects.

In Java, a likely implementation for Figure 15.22 follows. Notice that the *Thread* object in the code is excluded from the UML diagram, because it is simply a consistent "overhead" mechanism to realize an asynchronous call in Java.

```
public class ClockStarter
{
public void startClock()
{
    Thread t = new Thread( new Clock() );
    t.start(); // asynchronous call to the 'run' method on the Clock
    System.runFinalization(); // example follow-on message
}
// …
}
```

```
// objects should implement the Runnable interface
// in Java to be used on new threads

public class Clock implements Runnable
{
public void run()
{
    while ( true ) // loop forever on own thread
    {
        // …
    }
}
// …
}
```

15.5 Basic Communication 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 15.23). 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.

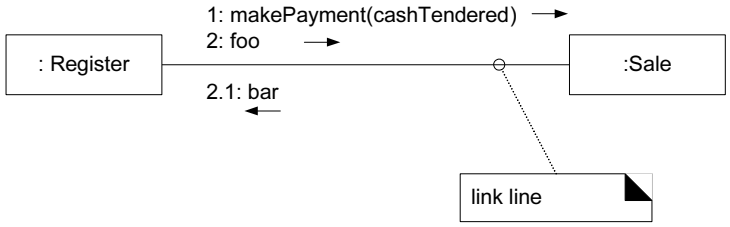


Figure 15.23 Link lines.

> *Note*
>
> Note that multiple messages, and messages *both* ways, flow along the same single link. There isn't one link line per message; all messages flow on the same line, which is like a road allowing two-way message traffic.

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 15.24). A sequence number is added to show the sequential order of messages in the current thread of control.

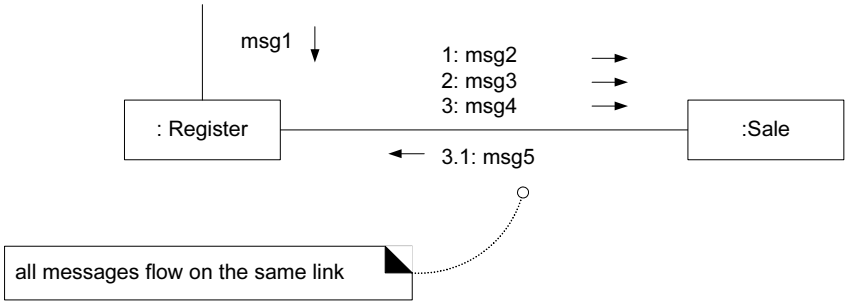


Figure 15.24 Messages.

Guideline

Don't number the starting message. It's legal to do so, but simplifies the overall numbering if you don't.

Messages to "self" or "this"

A message can be sent from an object to itself (Figure 15.25). This is illustrated by a link to itself, with messages flowing along the link.

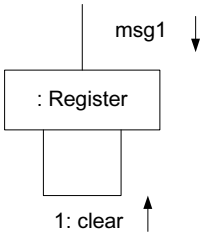


Figure 15.25 Messages to "this."

Creation of Instances

Any message can be used to create an instance, but the convention in the UML is to use a message named *create* for this purpose (some use *new*). See Figure 15.26. If another (less obvious) message name is used, the message may be annotated with 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 tagged value** *{new}* may optionally be added to the lifeline box to highlight the creation. Tagged values are a flexible extension mechanism in the UML to add semantically meaningful information to a UML element.

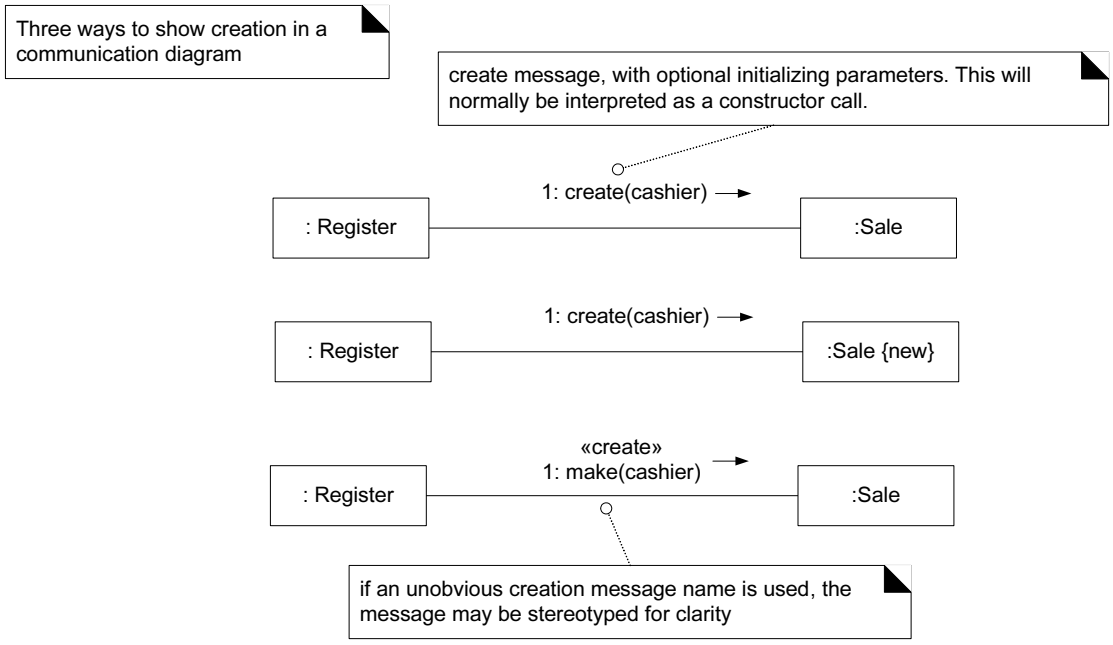


Figure 15.26 Instance creation.

Message Number Sequencing

The order of messages is illustrated with **sequence numbers**, as shown in Figure 15.27. The numbering scheme is:

1. The first message is not numbered. Thus, *msg1* is unnumbered.[10]
2. The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have a number appended to them. You denote nesting by prepending the incoming message number to the outgoing message number.

10.Actually, a starting number is legal, but it makes all subsequent numbering more awkward, creating another level of number-nesting deeper than otherwise necessary.

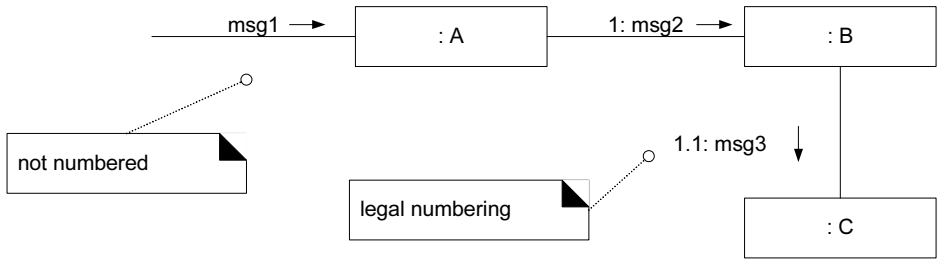


Figure 15.27 Sequence numbering.

Figure 15.28 shows a more complex case.

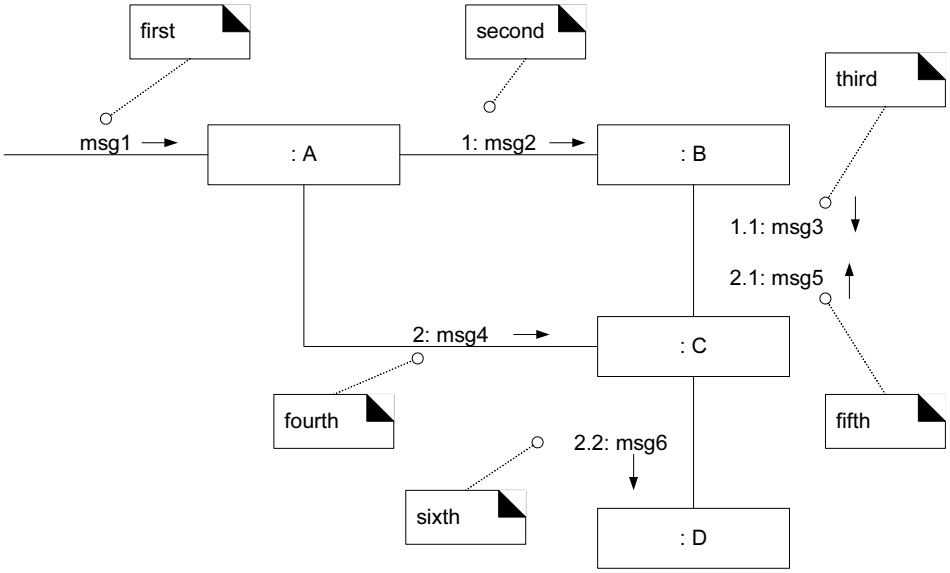


Figure 15.28 Complex sequence numbering.

Conditional Messages

You show a conditional message (Figure 15.29) 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*.

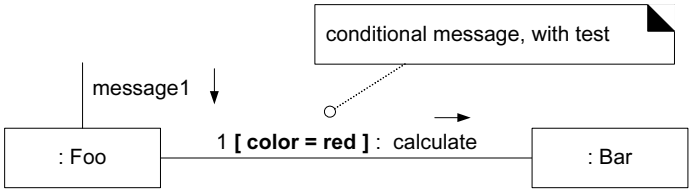


Figure 15.29 Conditional message.

Mutually Exclusive Conditional Paths

The example in Figure 15.30 illustrates the sequence numbers with mutually exclusive conditional paths.

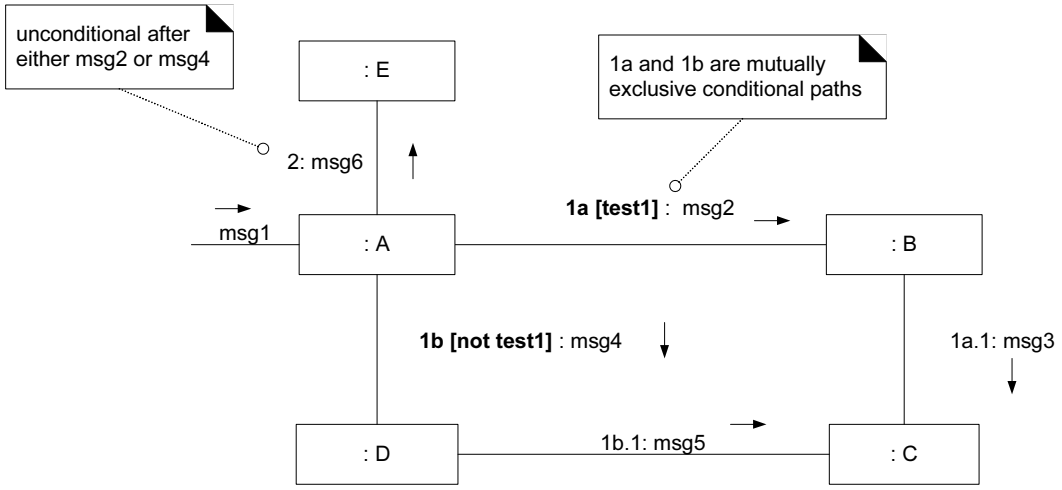


Figure 15.30 Mutually exclusive messages.

In this case we must modify the sequence expressions with a conditional path letter. The first letter used is **a** by convention. Figure 15.30 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 *1b.1* is nested message within *1b*.

Iteration or Looping

Iteration notation is shown in Figure 15.31. If the details of the iteration clause are not important to the modeler, a simple * can be used.

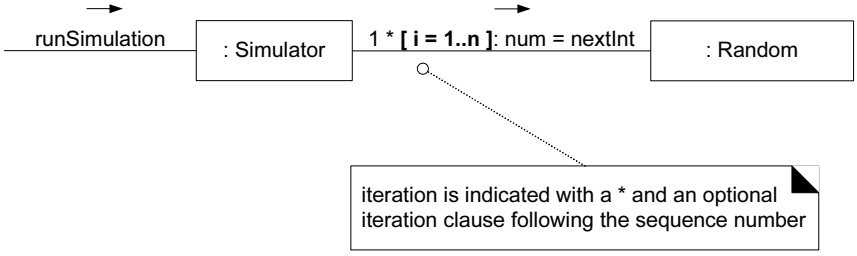


Figure 15.31 Iteration.

Iteration Over a Collection

A common algorithm is to iterate over all members of a collection (such as a list or map), sending the same message to each. In communication diagrams, this could be summarized as shown in Figure 15.32, although there is no official UML convention.

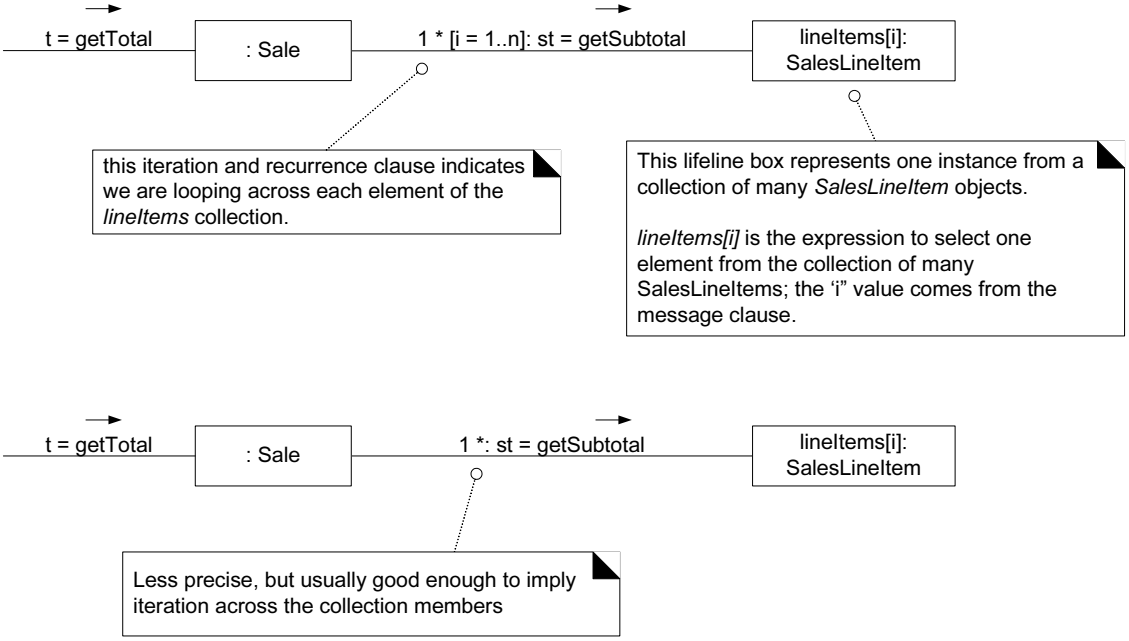


Figure 15.32 Iteration over a collection.

Messages to a Classes to Invoke Static (Class) Methods

See the discussion of metaclasses in the sequence diagram case on p. 236, to understand the purpose of the example in Figure 15.33.

Figure 15.33 Messages to a class object (static method invocation).

Polymorphic Messages and Cases

Refer to Figure 15.21 for the related context, class hierarchy, and example for sequence diagrams. As in the sequence diagram case, multiple communication diagrams can be used to show each concrete polymorphic case (Figure 15.34).

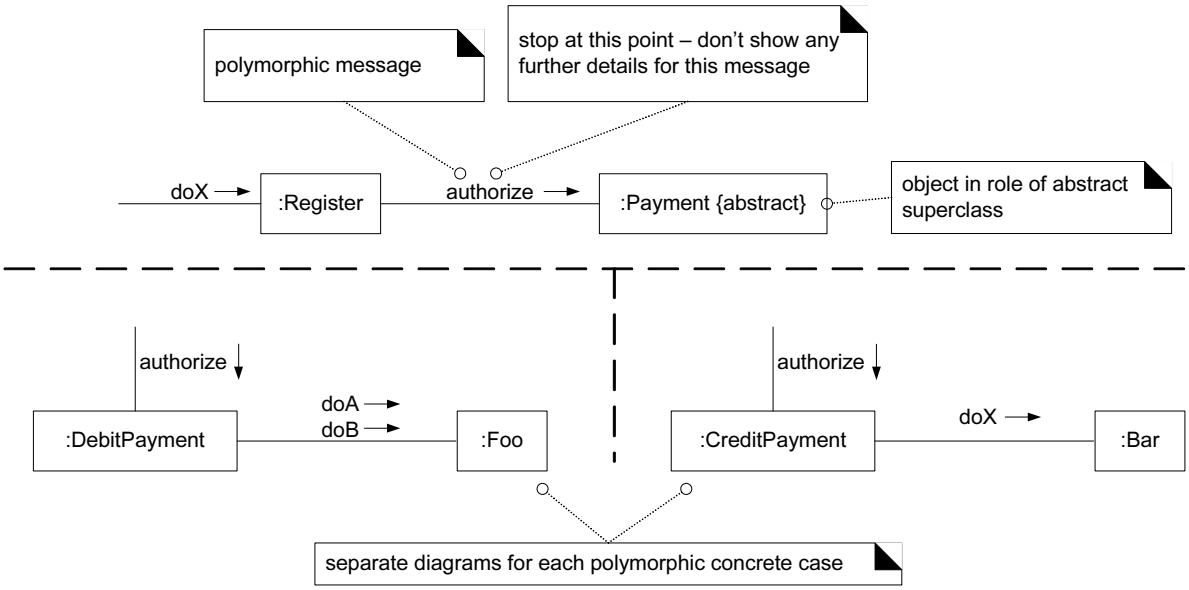


Figure 15.34 An approach to modeling polymorphic cases in communication diagrams.

Asynchronous and Synchronous Calls

As in sequence diagrams, asynchronous calls are shown with a stick arrow; synchronous calls with a filled arrow (see Figure 15.35).

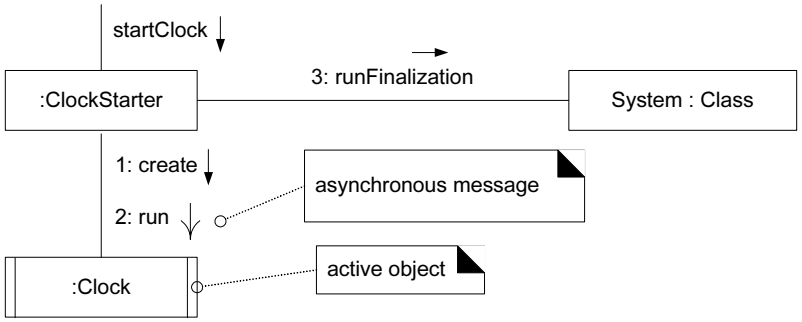


Figure 15.35 Asynchronous call in a communication diagram.