Applying UML and Patterns

**An Introduction to
Object-oriented Analysis
and Design
and Iterative Development**

Part III Elaboration Iteration I – Basic^2

Software Engineering

Chapter 15
UML Interaction Diagrams

Introduction

❑ The UML includes interaction diagrams to illustrate how objects interact via messages.

   ○ **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?
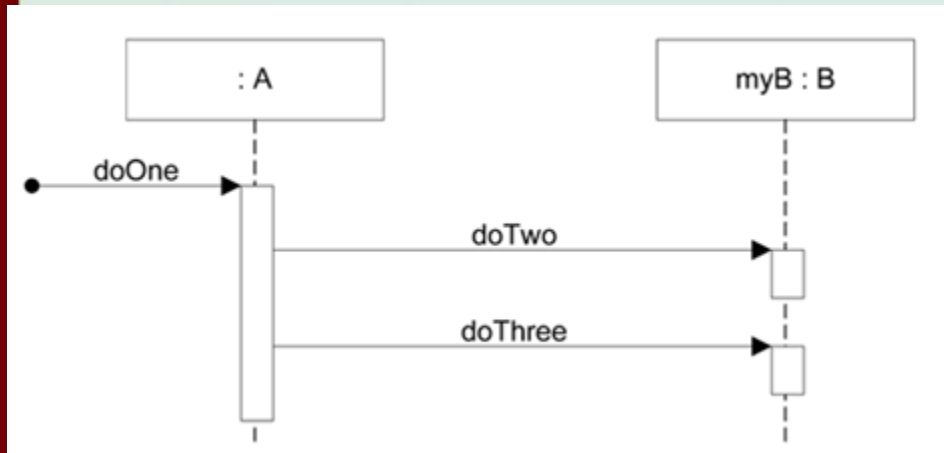
Sequence and Communication Diagrams

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

Sequence Diagram

❑ Sequence diagrams illustrate interactions in a kind of fence format, in which each new object is added to the right,
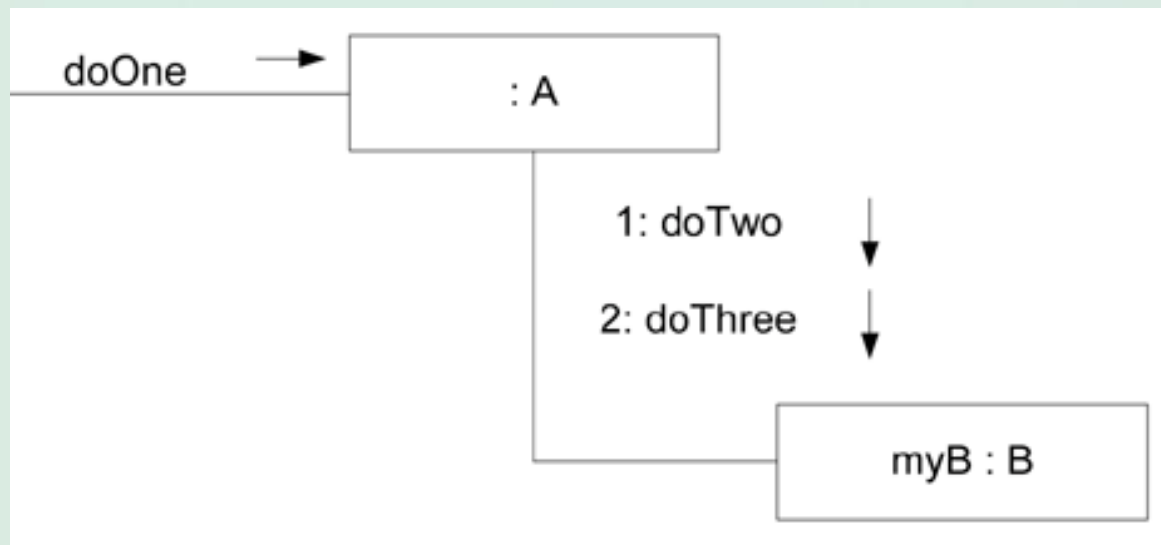


```
public class A {
    private B myB = new B();
    public void doOne() {
        myB.doTwo();
        myB.doThree();
    }
    // …
}
```

Software Engineering

Communication Diagram

❑ Communication diagrams illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram

Strengths and Weaknesses ₁

❑ Sequence diagrams have some advantages over communication diagrams

  ○ 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

  ○ 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:"

Software Engineering

Strengths and Weaknesses $_2$

❑ Advantages of communication diagrams

 ○ communication diagrams have advantages when applying "UML as sketch" to draw on walls (an Agile Modeling practice) because they are much more space-efficient.

 ○ boxes can be easily placed or erased anywhere horizontal or vertical.

 ○ 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)

Software Engineering

Strengths and Weaknesses ₃

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 economicalflexibility to add new objects in two dimensions	more difficult to see sequence of messages fewer notation options

Software Engineering

★ ★ ★

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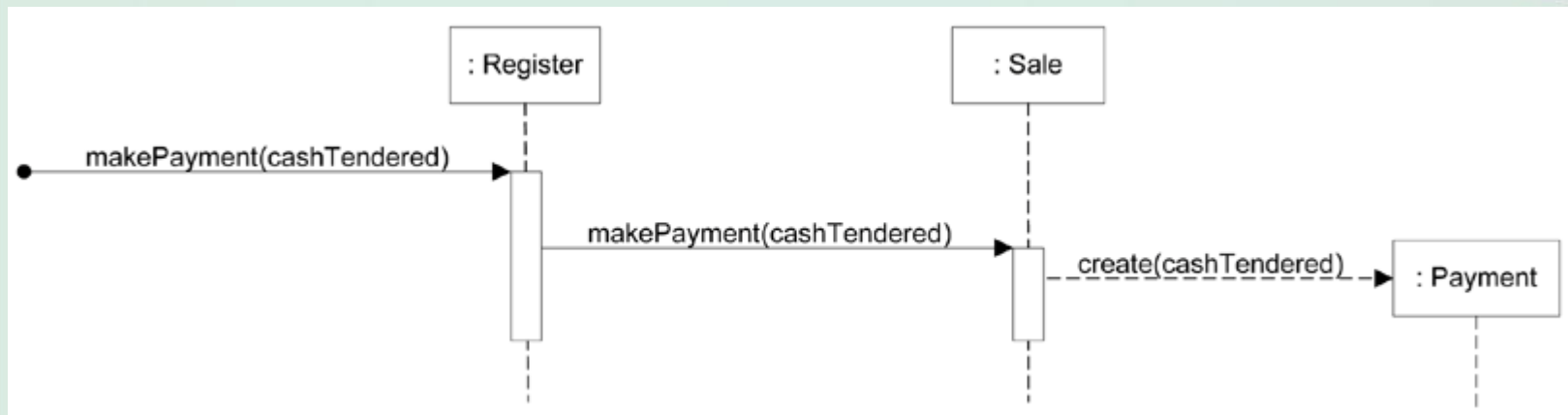
1. The message *makePayment* is sent to an instance of a Register. The sender is not identified.
2. The Register instance sends the *makePayment* message to a Sale instance.
3. The Sale instance creates an instance of a Payment.

What might be some related code for the Sale class and its *makePayment* method?

Software Engineering
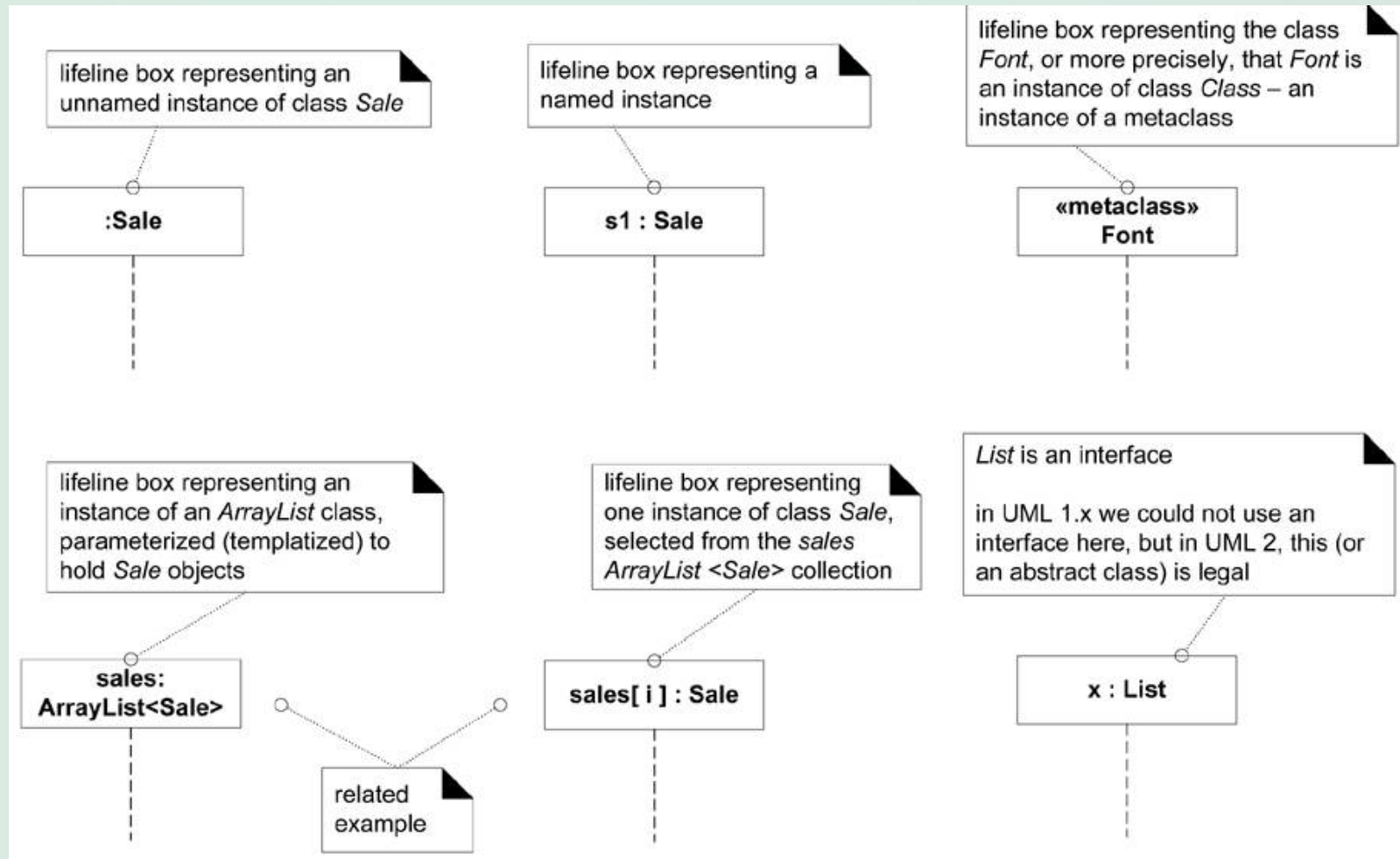
Example Communication Diagram
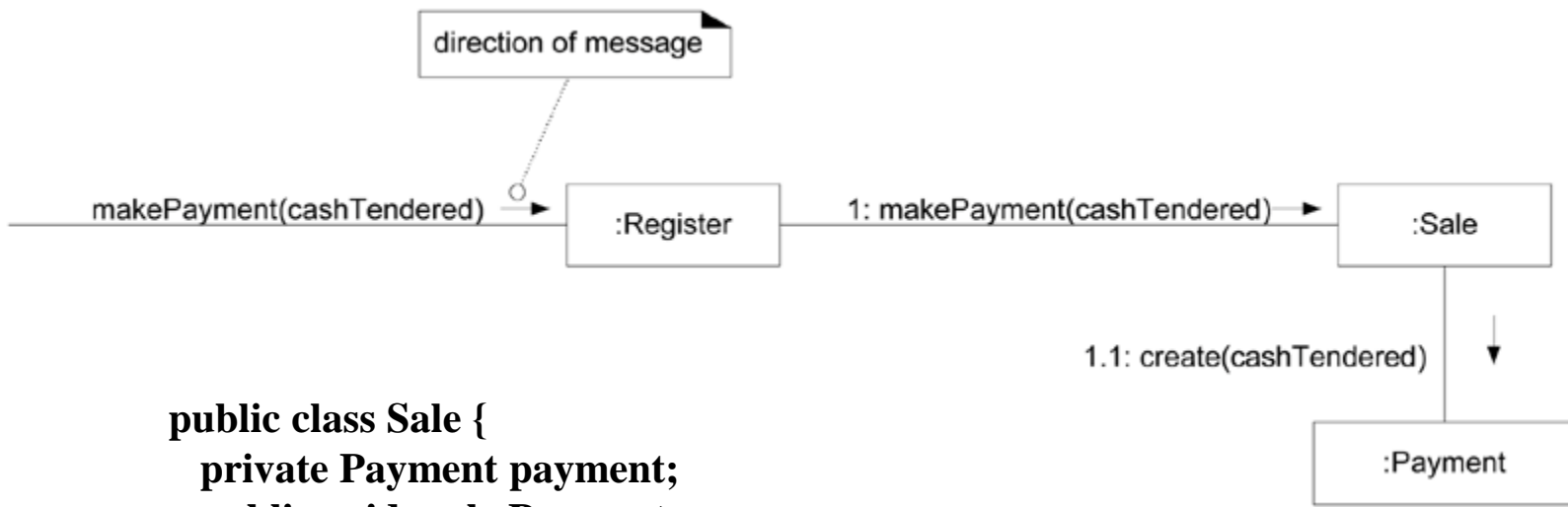

```
public class Sale {
    private Payment payment;
    public void makePayment
            ( Money cashTendered )  {
        payment = new Payment
            ( cashTendered );
        //…
    }
    // …
}
```

Software Engineering

Common UML Interaction Diagram Notation

Lifeline boxes to show participants in interactions

Basic Message Expression Syntax

❑ UML has a standard syntax for these message expressions

   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*

Software Engineering

Singleton Objects

❑ In the world of OO design patterns, there is one that is especially common, called the Singleton pattern
  ○ There is only one instance of a class instantiated - never two

: Register

: Store   1

the '1' implies this is a Singleton, and accessed via the Singleton pattern

doX

doA

Software Engineering

# Basic Sequence Diagram Notation

❑ **Lifeline Boxes and Lifelines**

  ○ 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 solid line between the vertical lifelines

  ○ The time ordering is organized from top to bottom of lifelines.

Software Engineering

Message and Focus of Control $_1$
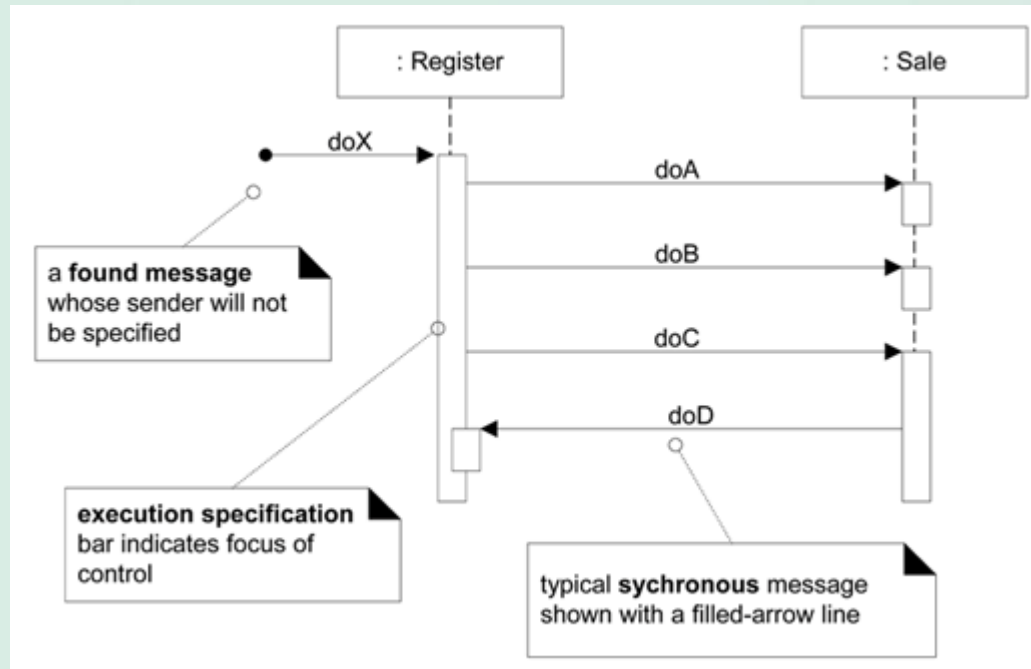
❑ found message:

  ○ the sender will not be specified, is not known, or that the message is coming from a random source

Message and Focus of Control $_2$

❑ Sequence diagrams may also show the focus of control 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.

Illustrating Reply or Returns

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

- ○ Using the message syntax
  - ◆ *returnVar = message(parameter).*
- ○ Using a reply (or return) message line at the end of an activation bar.
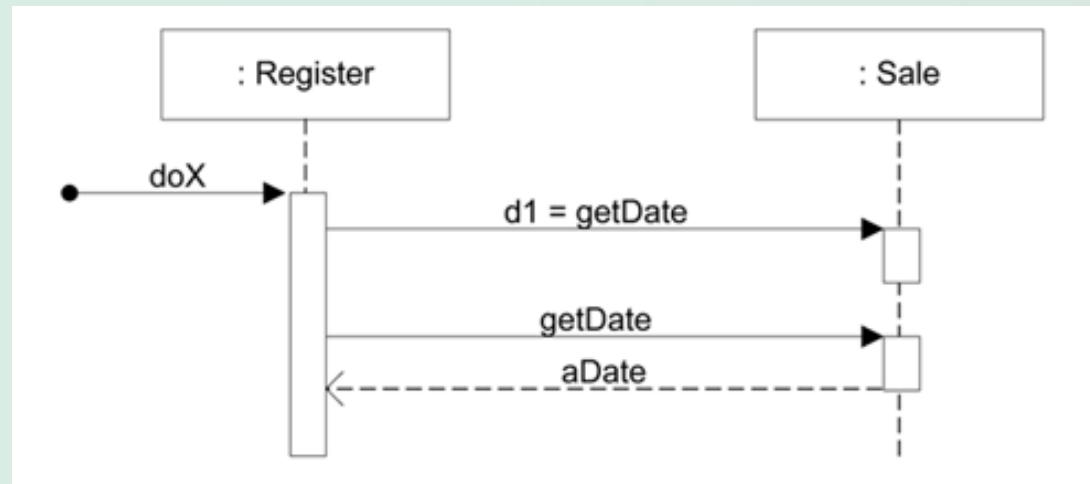
Creation of Instances

❑ 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 call.

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

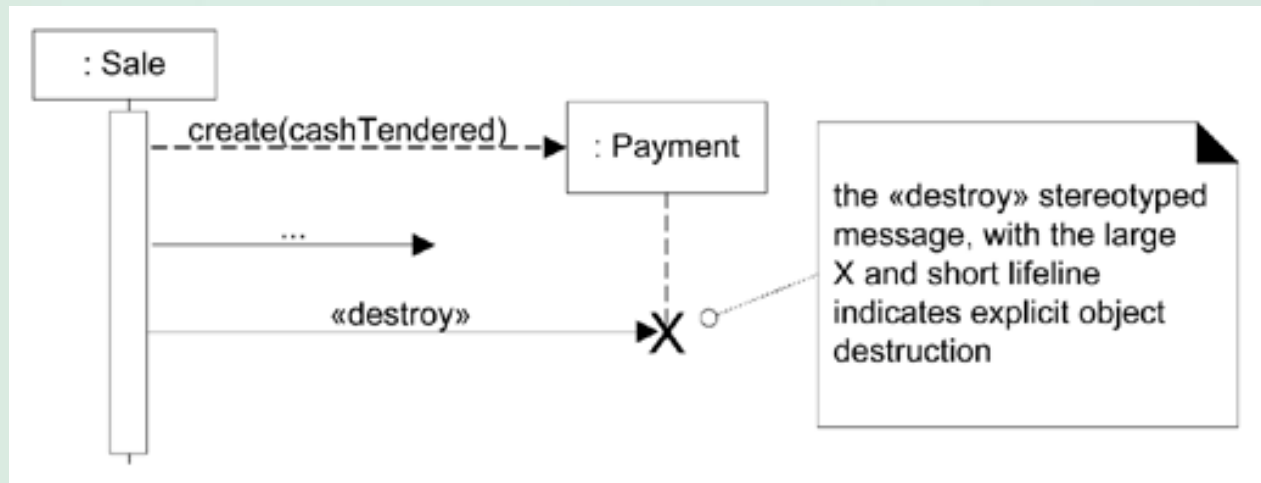❑ Object Lifelines and Object Destruction

Diagram Frames in UML Sequence Diagrams [1]

❑ To support conditional and looping constructs (among many other things), the UML uses frames.

  ○ Frames are regions or fragments of the diagrams;
  ○ they have an operator or label (such as loop) and a guard (conditional clause).
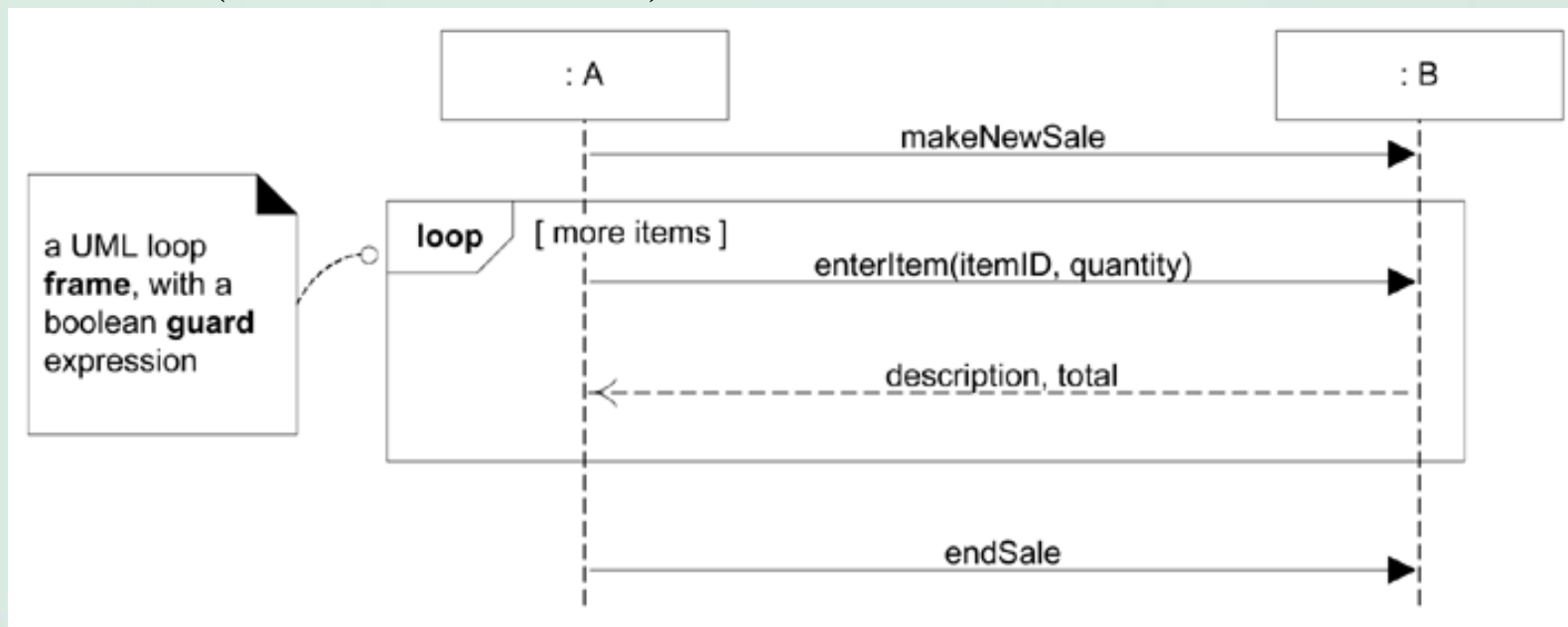

a UML loop **frame**, with a boolean **guard** expression

Software Engineering

★ ★

Diagram Frames in UML Sequence Diagrams ₂

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.

Software Engineering

Condition message

❑ Use UML 1 style only for simple single messages when sketching
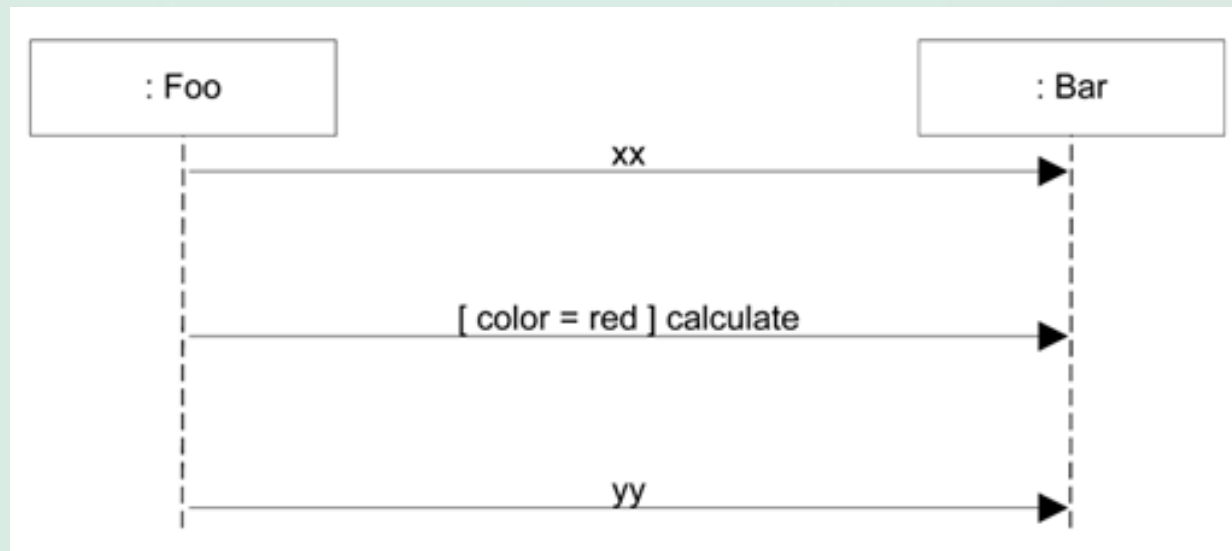
Mutually Exclusive Conditional Messages

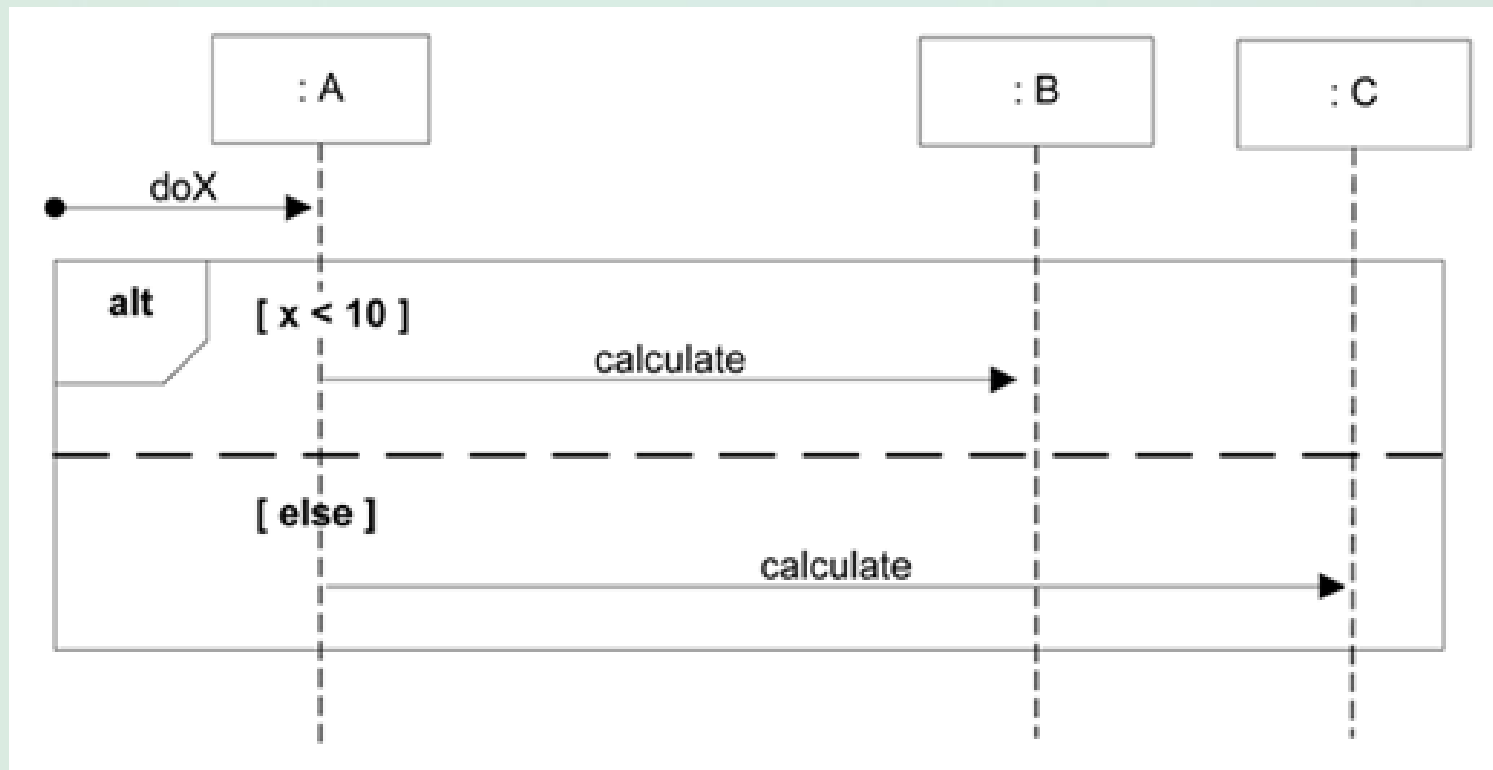❑ An ALT frame is placed around the mutually exclusive alternatives

★ ★ ★

Iteration Over a Collection $_1$

This lifeline box represents one instance from a collection of many *SalesLineItem* objects.

lineItems[i] is the expression to select one element from the collection of many SalesLineItems; the 'i' value refers to the same "i" in the guard in the LOOP frame

an **action box** may contain arbitrary language statements (in this case, incrementing 'i')

it is placed over the lifeline to which it applies

❑ 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.

Software Engineering

Iteration Over a Collection ₂

❑ The selector expression is used to select one object from a group. Lifeline participants should represent one object, not a collection.

```java
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;
  }
  // …
}
```

Software Engineering

Iteration Over a Collection ₃

❑ Another variation is shown below
  ○ 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

Relating Interaction Diagrams 1
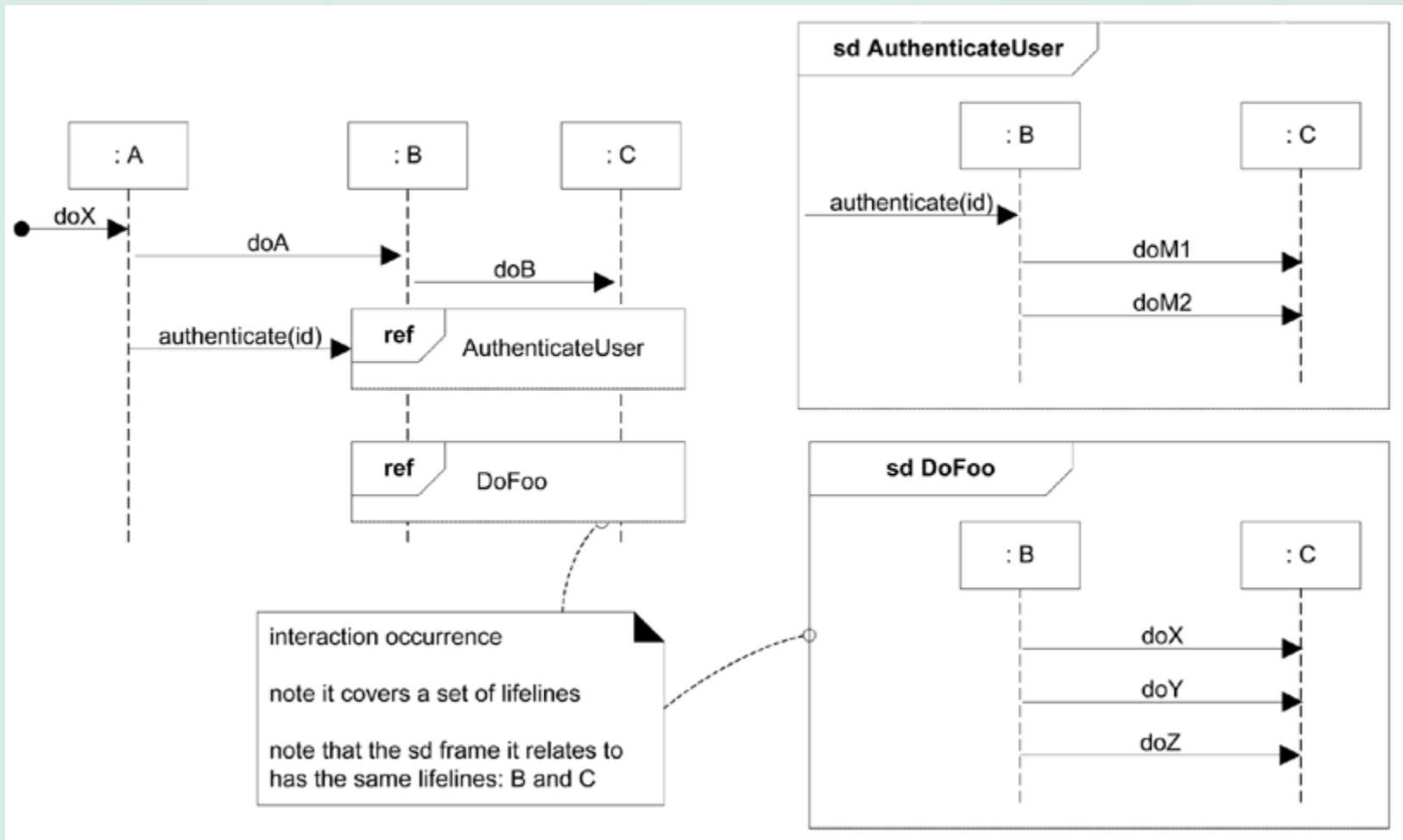
Relating Interaction Diagrams 2

❑ An interaction occurrence (also called an interaction use) is a reference to an interaction within another interaction.

○ 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.

Metaclass ₁

❑ 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

❑ 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.

Software Engineering

Metaclass $_2$
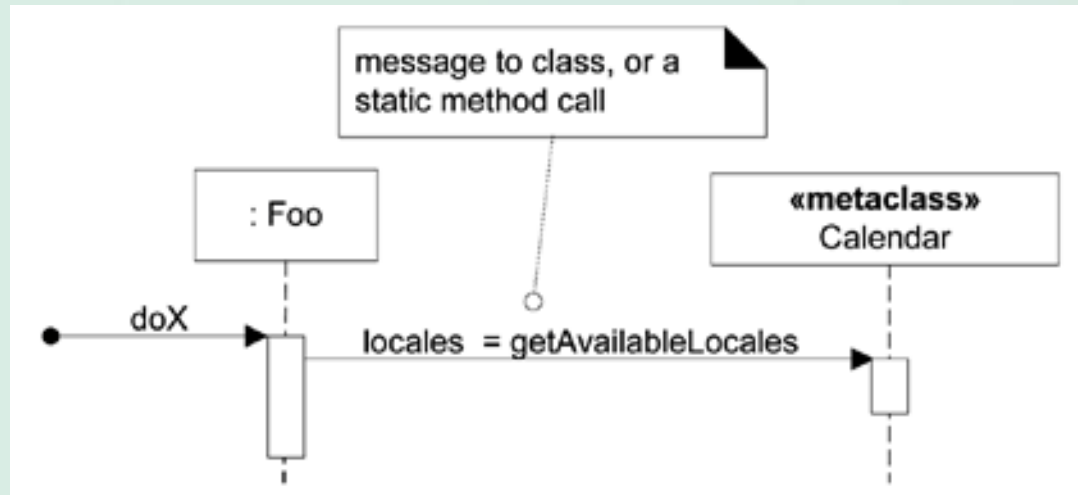

```
public class Foo {
    public void doX() {
        // static method call on class
        Calendar Locale[] locales = Calendar.getAvailableLocales();
            // …
    }
    // …
}
```

Software Engineering

Polymorphic Messages and Cases

❏ How to show it in a sequence diagram? That's a common UML question.

○ 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
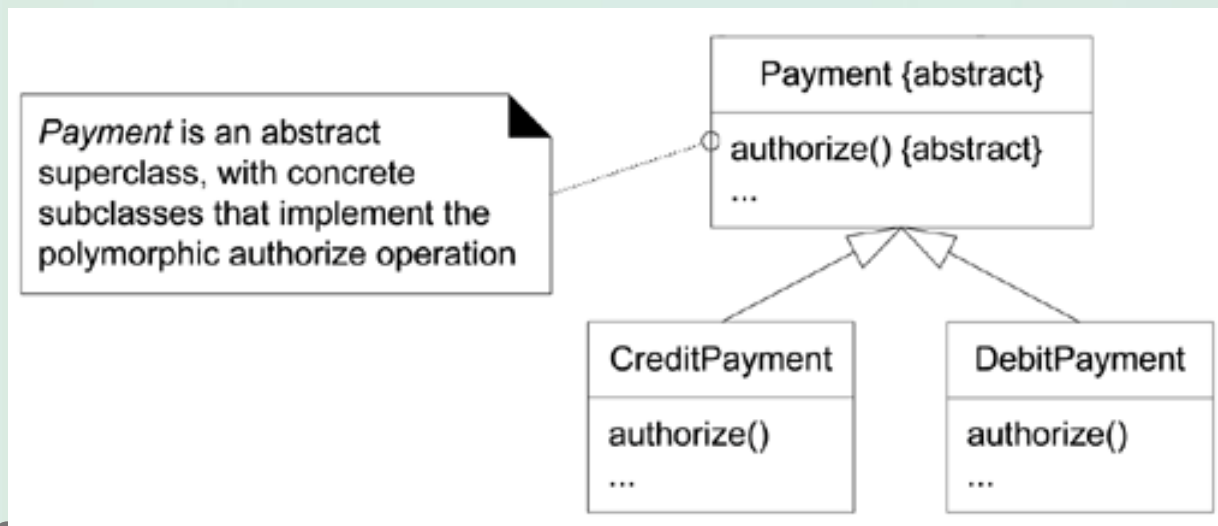
Asynchronous and Synchronous Calls

❑ 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

Software Engineering

Conti.

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

Software Engineering
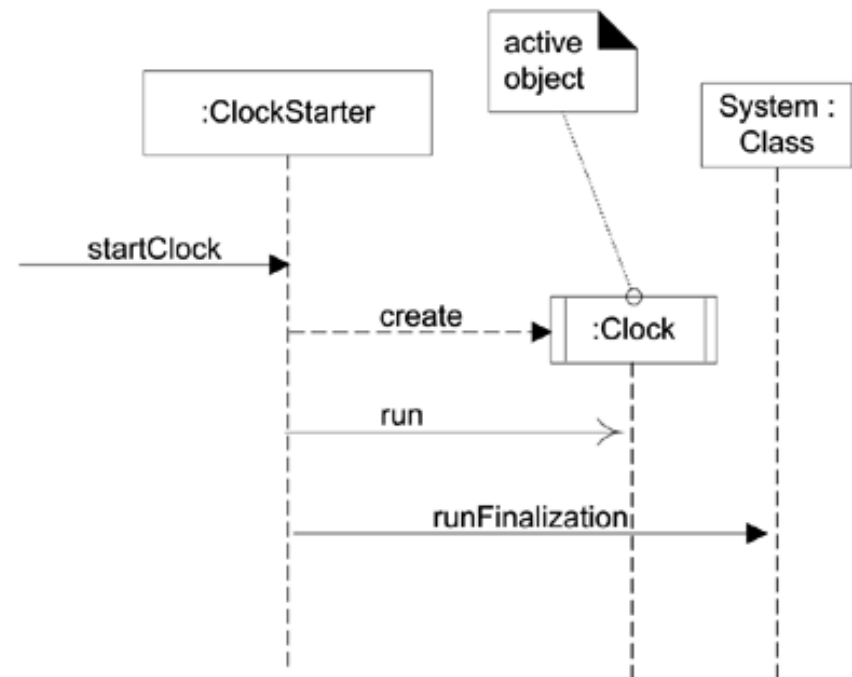
Conti.
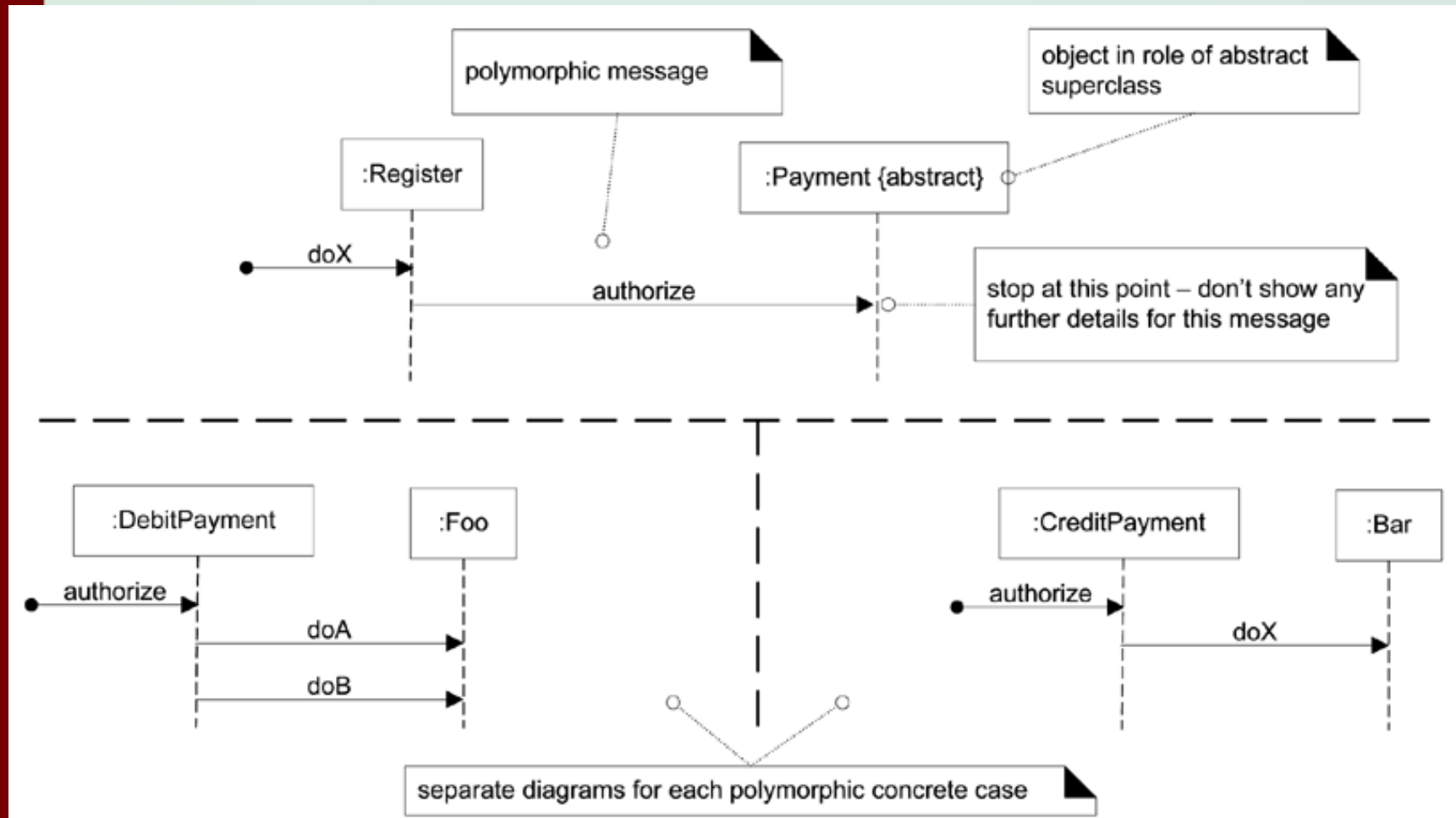
Basic Communication Diagram Notation

❑ A **link** is a connection path between two objects;
  ○ it indicates some form of navigation and visibility between the objects is possible
  ○ More formally, a link is an instance of an association.
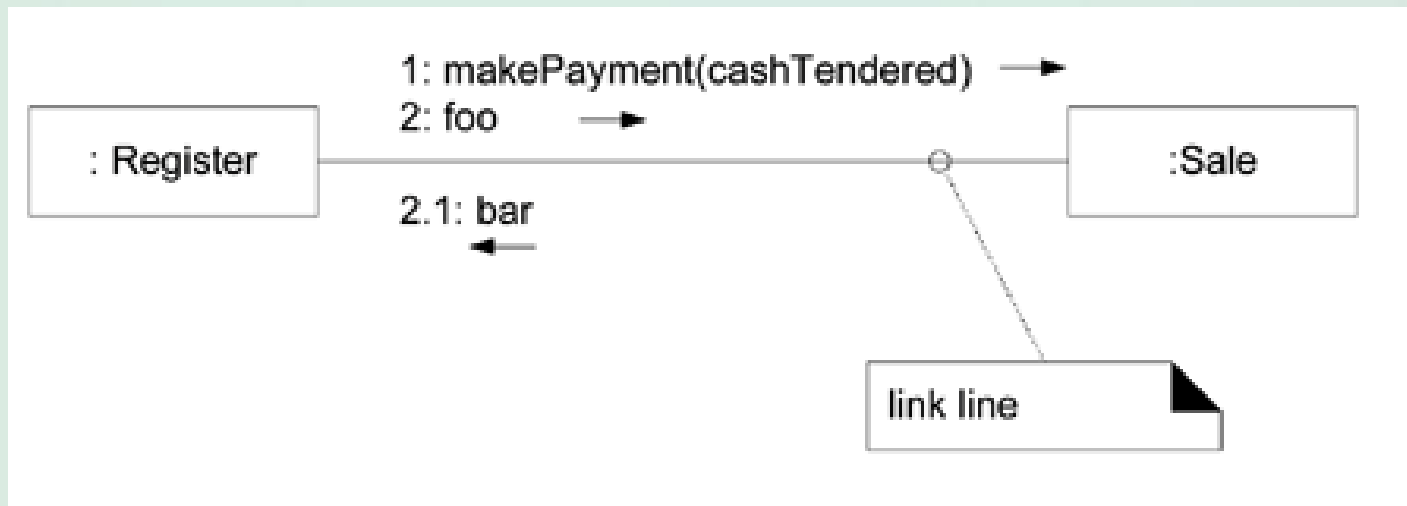
Message

❑ **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.

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

Software Engineering
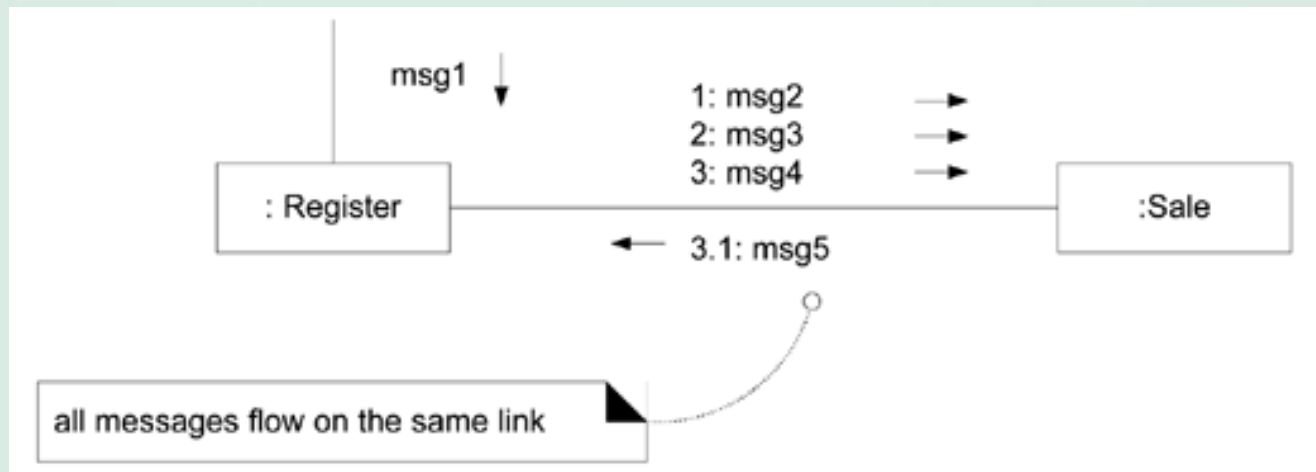
Creation of Instances $_1$

❑ 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).

❑ 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.

Creation of Instances $_2$

Three ways to show creation in a communication diagram

create message, with optional initializing parameters. This will normally be interpreted as a constructor call

```
1: create(cashier)
: Register  ─────────────────────  :Sale
```

```
1: create(cashier)
: Register  ─────────────────────  :Sale {new}
```

```
        «create»
        1: make(cashier)
: Register  ─────────────────────  :Sale
```

if an unobvious creation message name is used, the message may be stereotyped for clarity
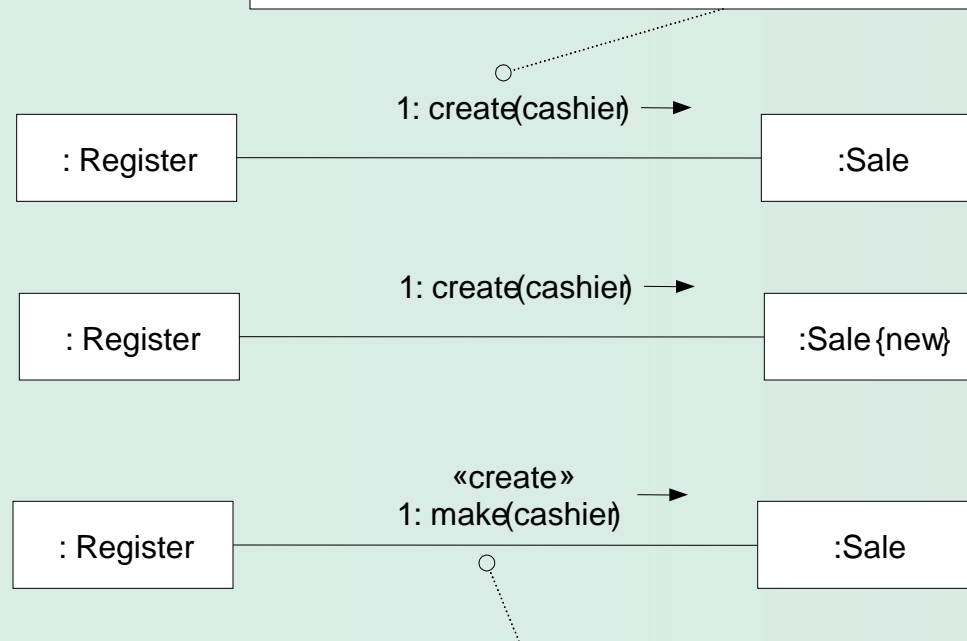
Message Number Sequencing $_1$

❑ The order of messages is illustrated with sequence numbers, The numbering scheme is:

  ○ The first message is not numbered. Thus, msg1 is unnumbered

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

  ○ 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 pre-pending the incoming message number to the outgoing message number

Message Number Sequencing ₂

msg1 → | : A | 1: msg2 → | : B |

not numbered

1.1: msg3 ↓

legal numbering

| : C |

Message Number Sequencing ₃

first

second

third

msg1 →

| : A | 1: msg2 → | : B |

1.1: msg3

2.1: msg5

2: msg4 →

| : C |

fourth

fifth

2.2: msg6

sixth

| : D |
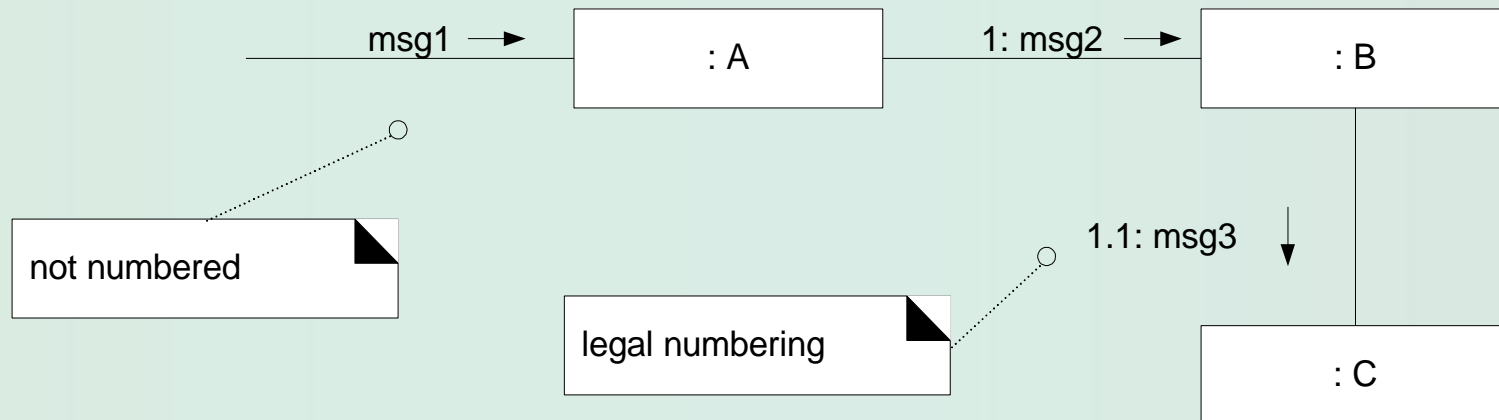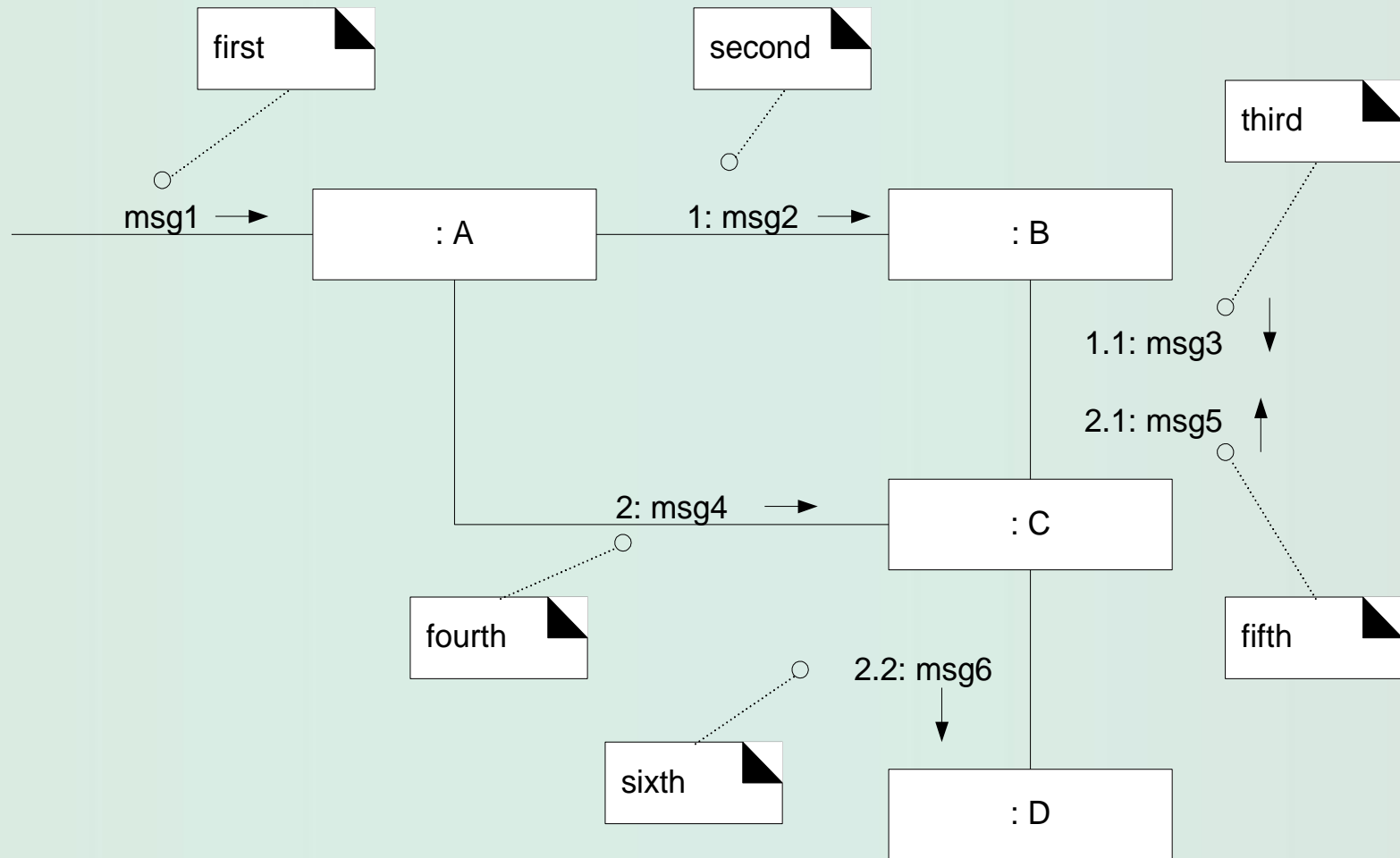
Conditional Messages

❑ A conditional message 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

1 **[color = red]** : calculate

: Foo

: Bar

Mutually Exclusive Conditional Paths ₁

unconditional after
either msg2 or msg4

: E

1a and 1b are mutually
exclusive conditional paths

2: msg6

msg1
: A

1a [test1] : msg2

: B

1b [not test1] : msg4

1a.1: msg3

: D

1b.1: msg5

: C
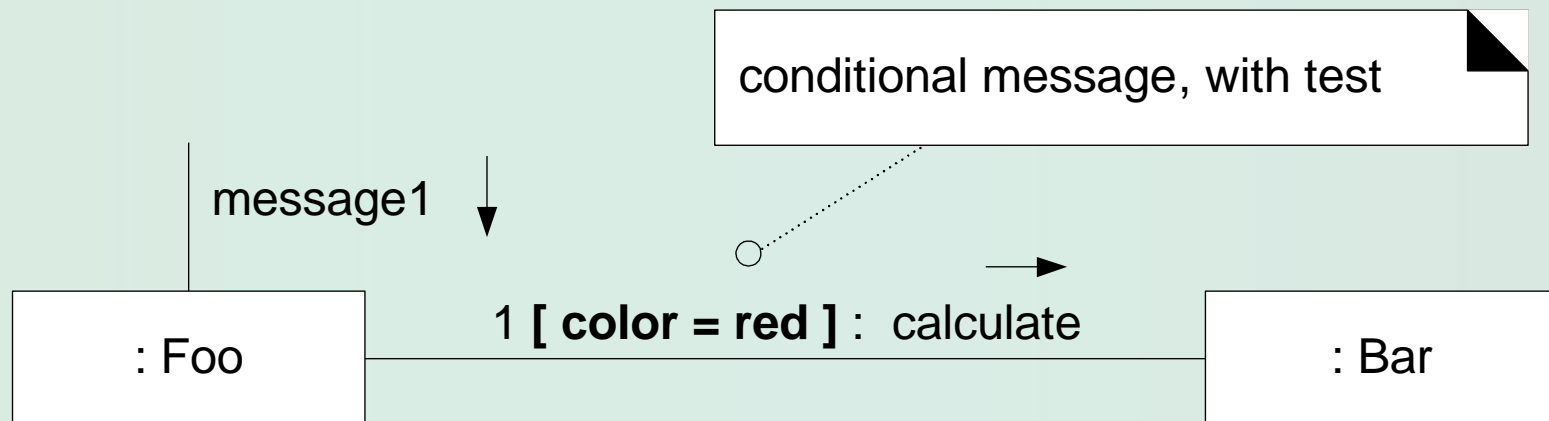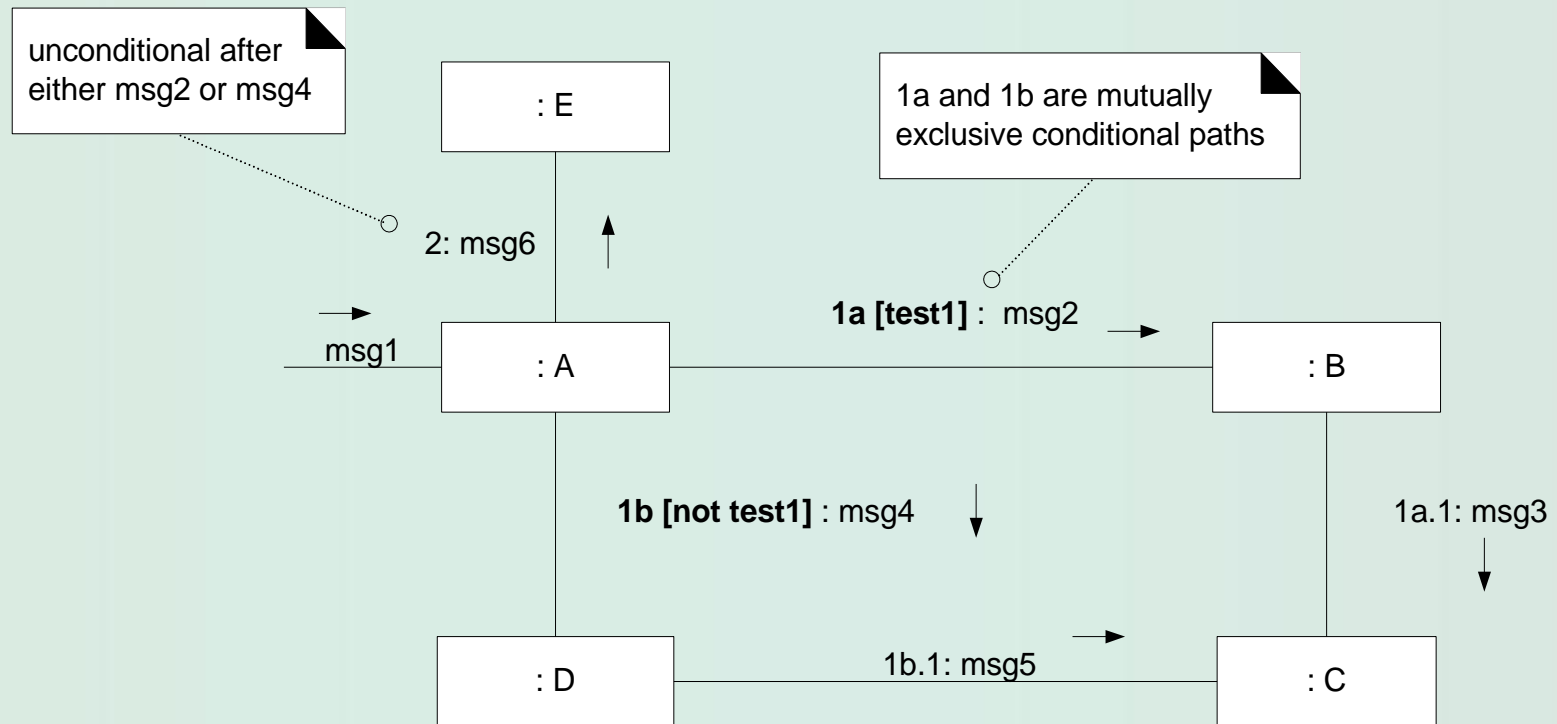
Mutually Exclusive Conditional Paths $_2$

❑ The example illustrates the sequence numbers with mutually exclusive conditional paths

❑ In this case we must modify the sequence expressions with a conditional path letter. The first letter used is a by convention. 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

❑ 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
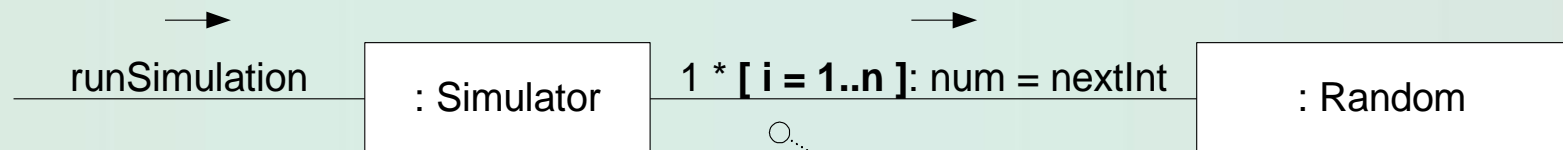
iteration is indicated with a * and an optional iteration clause following the sequence number

Iteration Over a Collection

t = getTotal → | : Sale | 1 * [i = 1..n]: st = getSubtotal → | lineItems[i]: SalesLineItem |
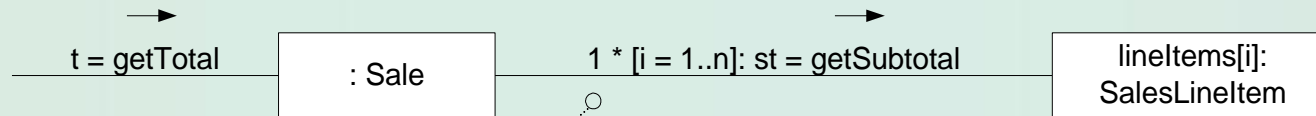
this iteration and recurrence clause indicates we are looping across each element of the *lineItems* collection.

This lifeline box represents one instance from a collection of many *SalesLineItem* objects.

lineItems[i] is the expression to select one element from the collection of many SalesLineItems; the value comes from the message clause.

t = getTotal → | : Sale | 1 *: st = getSubtotal → | lineItems[i]: SalesLineItem |
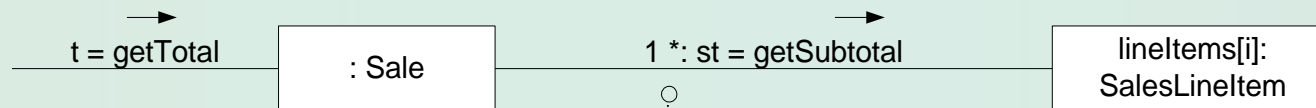
Less precise, but usually good enough to imply iteration across the collection members

Software Engineering

Messages to a Classes to Invoke Static Methods

message to class, or a static method call

doX

1: locs = getAvailableLocales

: Foo

«metaclass»
Calendar

Polymorphic Messages and Cases

polymorphic message

stop at this point– don't show any further details for this message

doX → :Register — authorize → :Payment {abstract}

object in role of abstract superclass

authorize ↓

:DebitPayment — doA →
doB → :Foo

authorize ↓

:CreditPayment — doX → :Bar

separate diagrams for each polymorphic concrete case

Software Engineering

Asynchronous and Synchronous Calls

startClock

3: runFinalization

:ClockStarter

System : Class

1: create

asynchronous message

2: run

:Clock

active object

OAuth 2.0 与交互图（1）

```
+--------+                              +---------------+
|        |--(A)- Authorization Request ->|   Resource    |
|        |                              |     Owner     |
|        |<-(B)-- Authorization Grant ---|               |
|        |                              +---------------+
|        |
|        |                              +---------------+
|        |--(C)-- Authorization Grant -->| Authorization |
| Client |                              |     Server    |
|        |<-(D)----- Access Token -------|               |
|        |                              +---------------+
|        |
|        |                              +---------------+
|        |--(E)----- Access Token ------>|   Resource    |
|        |                              |     Server    |
|        |<-(F)--- Protected Resource ---|               |
+--------+                              +---------------+
```

http://www.ruanyifeng.com/blog/2014/05/oauth_2_0.html

OAuth 2.0 与交互图（2）

```
+----------+
| Resource |
|  Owner   |
|          |
+----------+
     ^
     |
    (B)
+----|-----+          Client Identifier      +---------------+
|    -+----(A)-- & Redirection URI ---->|               |
|  User-   |                                |  Authorization |
|  Agent  -+----(B)-- User authenticates --->|    Server    |
|          |                                |               |
|         -+----(C)-- Authorization Code ---<|              |
+-|----|---+                                +---------------+
  |    |                                        ^      v
 (A)  (C)                                       |      |
  |    |                                        |      |
  ^    v                                        |      |
+----------+                                    |      |
|          |>---(D)-- Authorization Code --------'      |
| Client   |          & Redirection URI                |
|          |                                           |
|          |<---(E)----- Access Token -----------------'
+----------+          (w/ Optional Refresh Token)
```

Software Engineering

OAuth 2.0 与交互图（3）