THE UNIVERSITY OF MELBOURNE

SWEN30006 Software Modelling and Design

UML INTERACTION DIAGRAMS

Larman Chapter 15

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

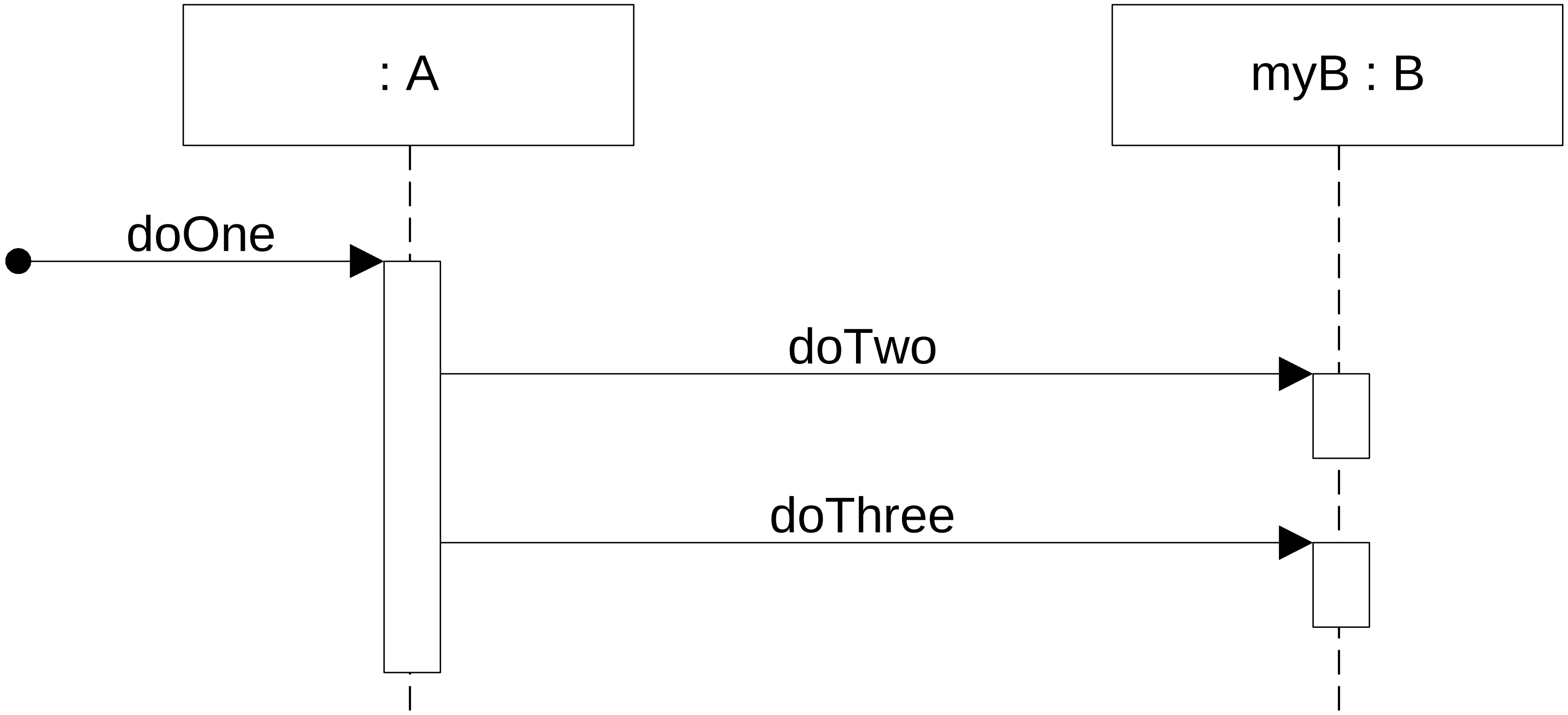
—Jeff Valdez

Objectives

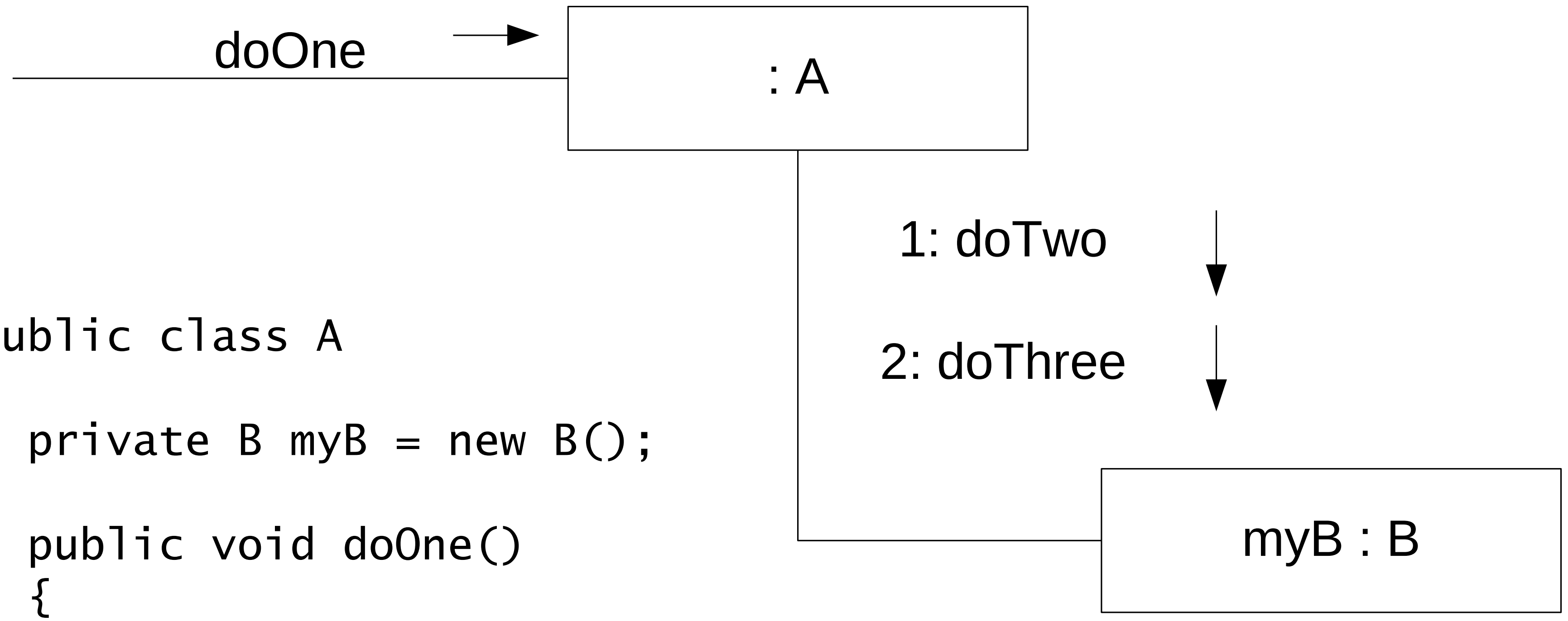
On completion of this topic you should be able to:

- Recognise and apply frequently used UML interaction diagram notation for
  - sequence diagrams
  - communication diagrams

Sequence Diagram

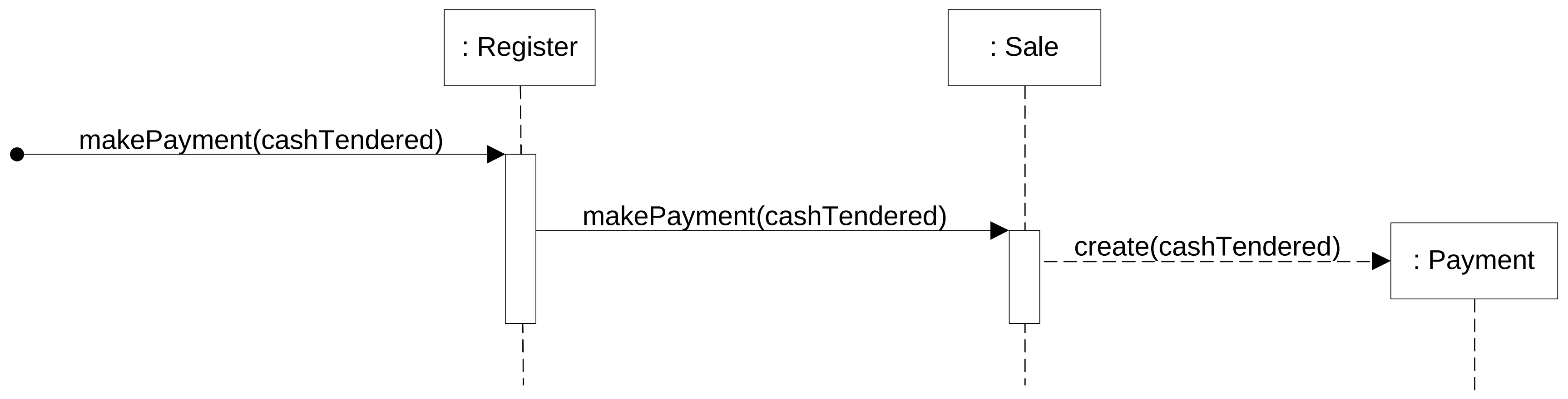


Communication Diagram

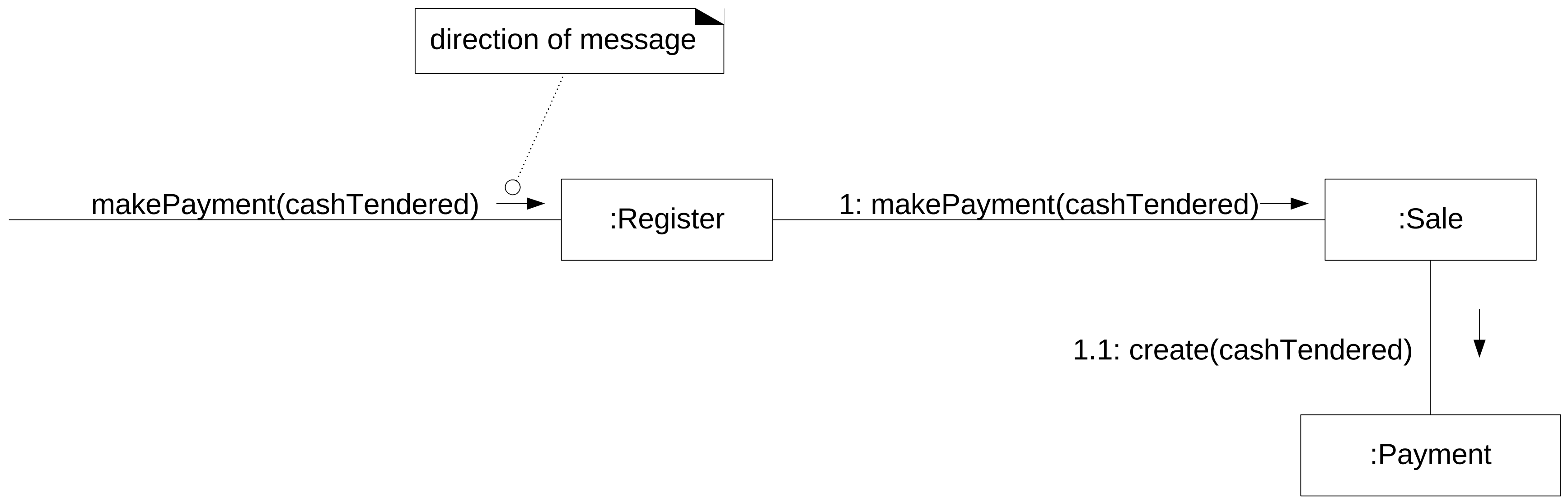


```
public class A
{
   private B myB = new B();

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

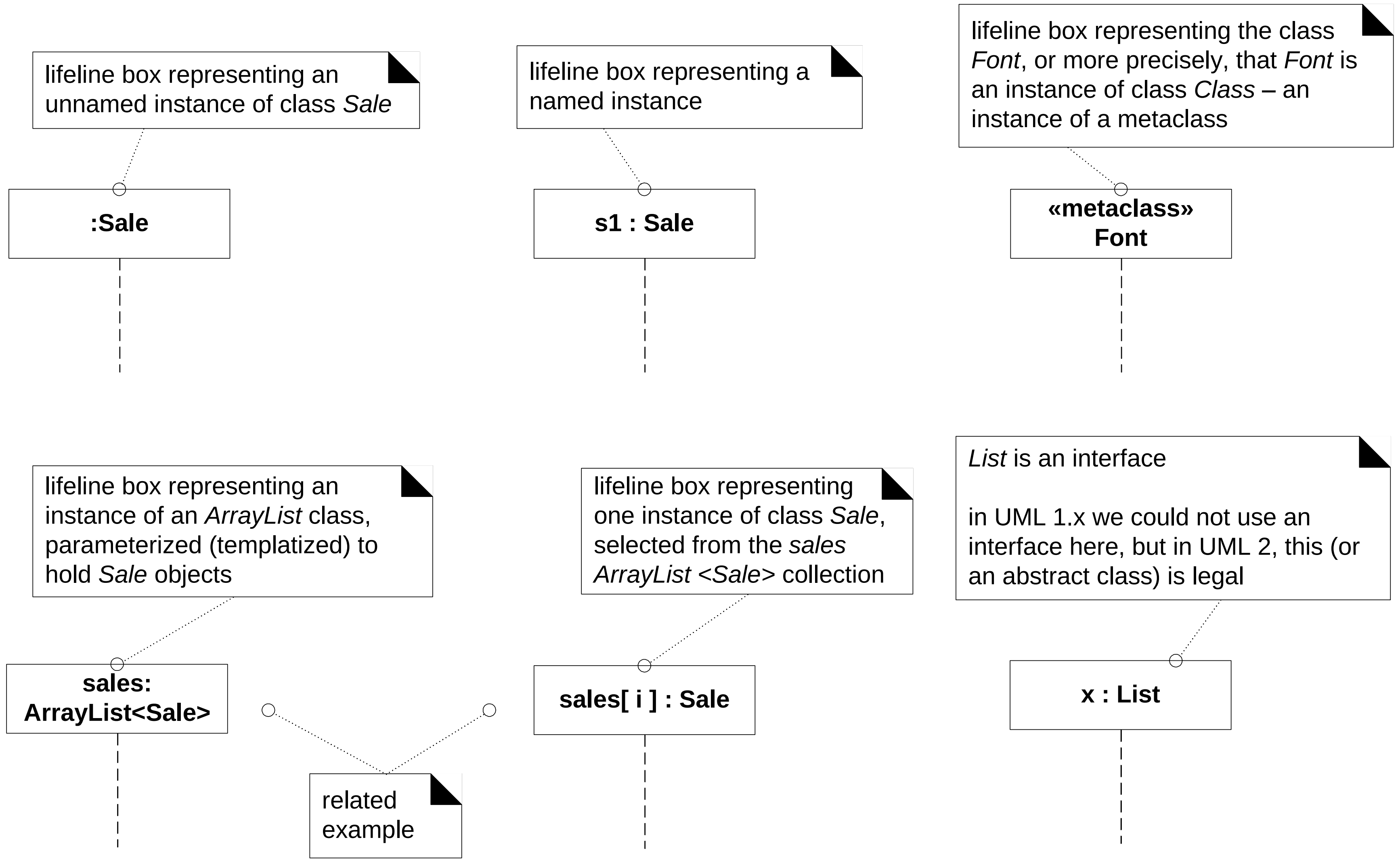
SD: *makePayment*



CD: makePayment


direction of message
makePayment(cashTendered)
:Register
1: makePayment(cashTendered)
:Sale
1.1: create(cashTendered)
:Payment

Lifelines: Different Participants



lifeline box representing an unnamed instance of class *Sale*

lifeline box representing a named instance

lifeline box representing the class *Font*, or more precisely, that *Font* is an instance of class *Class* – an instance of a metaclass

lifeline box representing an instance of an *ArrayList* class, parameterized (templatized) to hold *Sale* objects

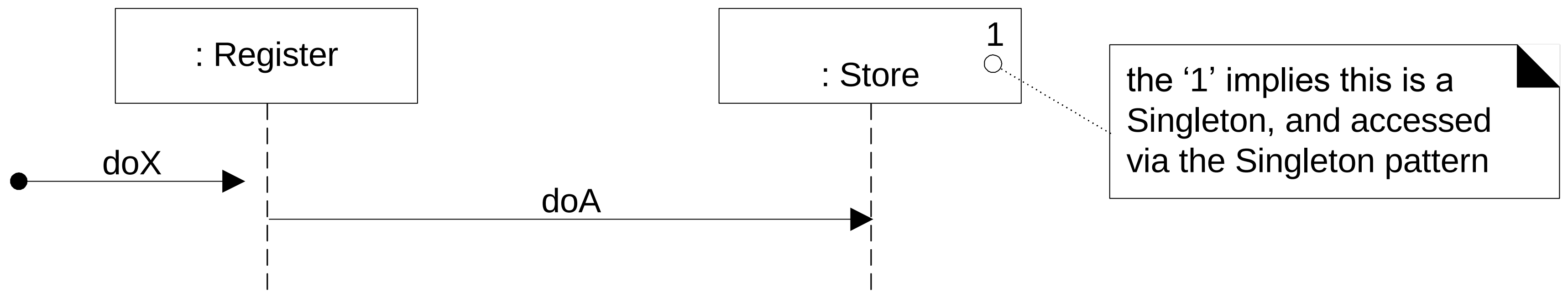
lifeline box representing one instance of class *Sale*, selected from the *sales* *ArrayList <Sale>* collection

List is an interface

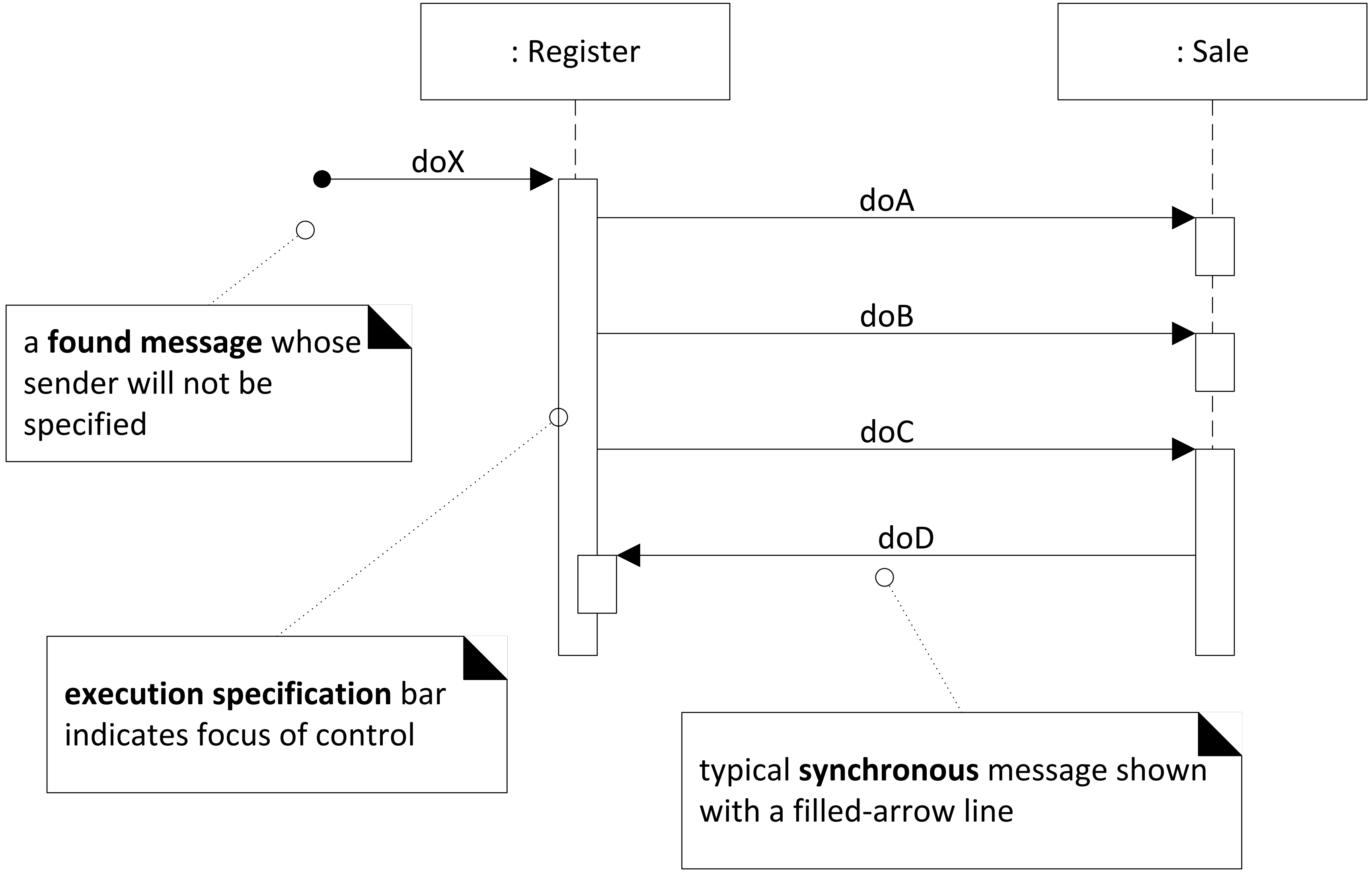
in UML 1.x we could not use an interface here, but in UML 2, this (or an abstract class) is legal

related example

Singletons in Interaction Diagrams


: Register
: Store
1
doX
doA
the '1' implies this is a Singleton, and accessed via the Singleton pattern

Messages and the Exec. Spec. Bar

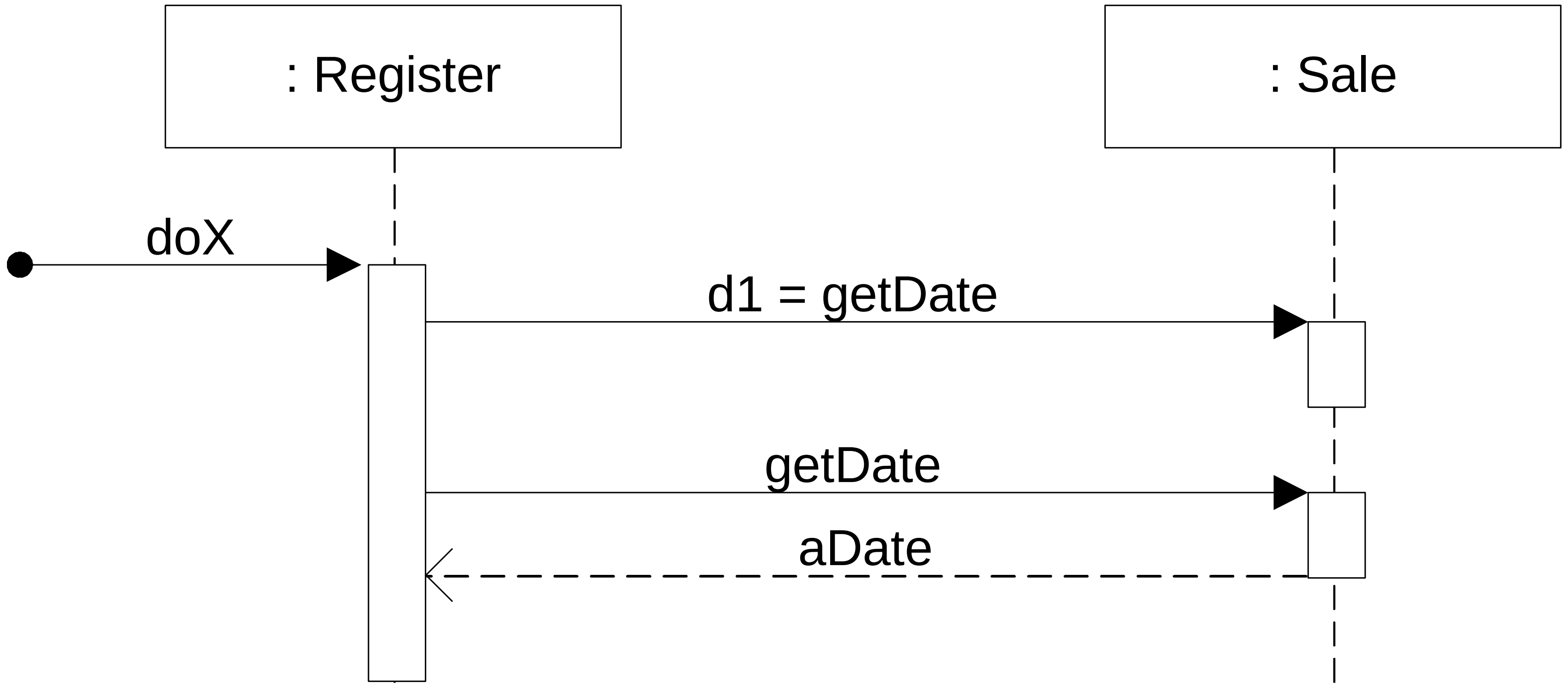


a **found message** whose sender will not be specified

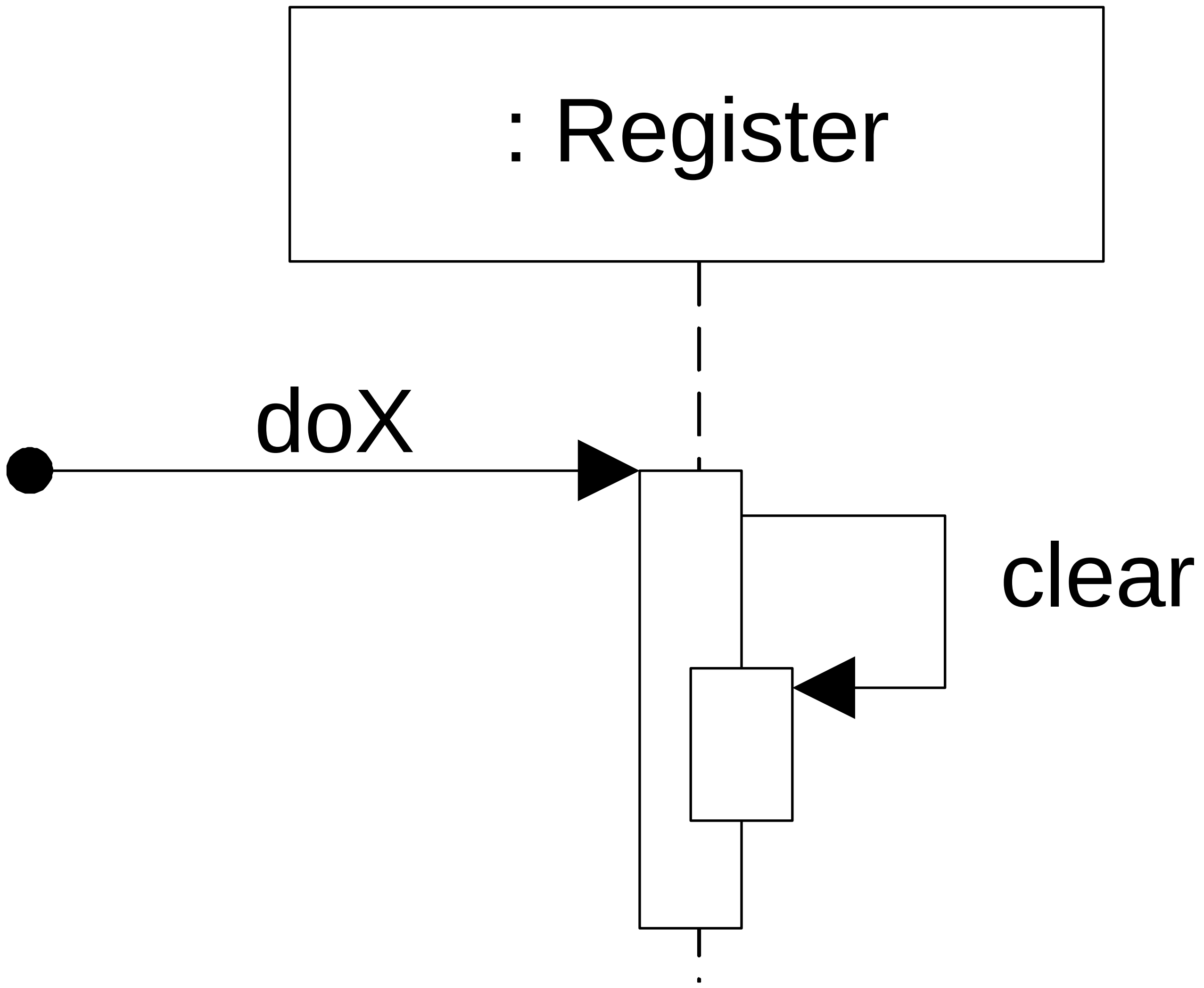
execution specification bar indicates focus of control

typical **synchronous** message shown with a filled-arrow line

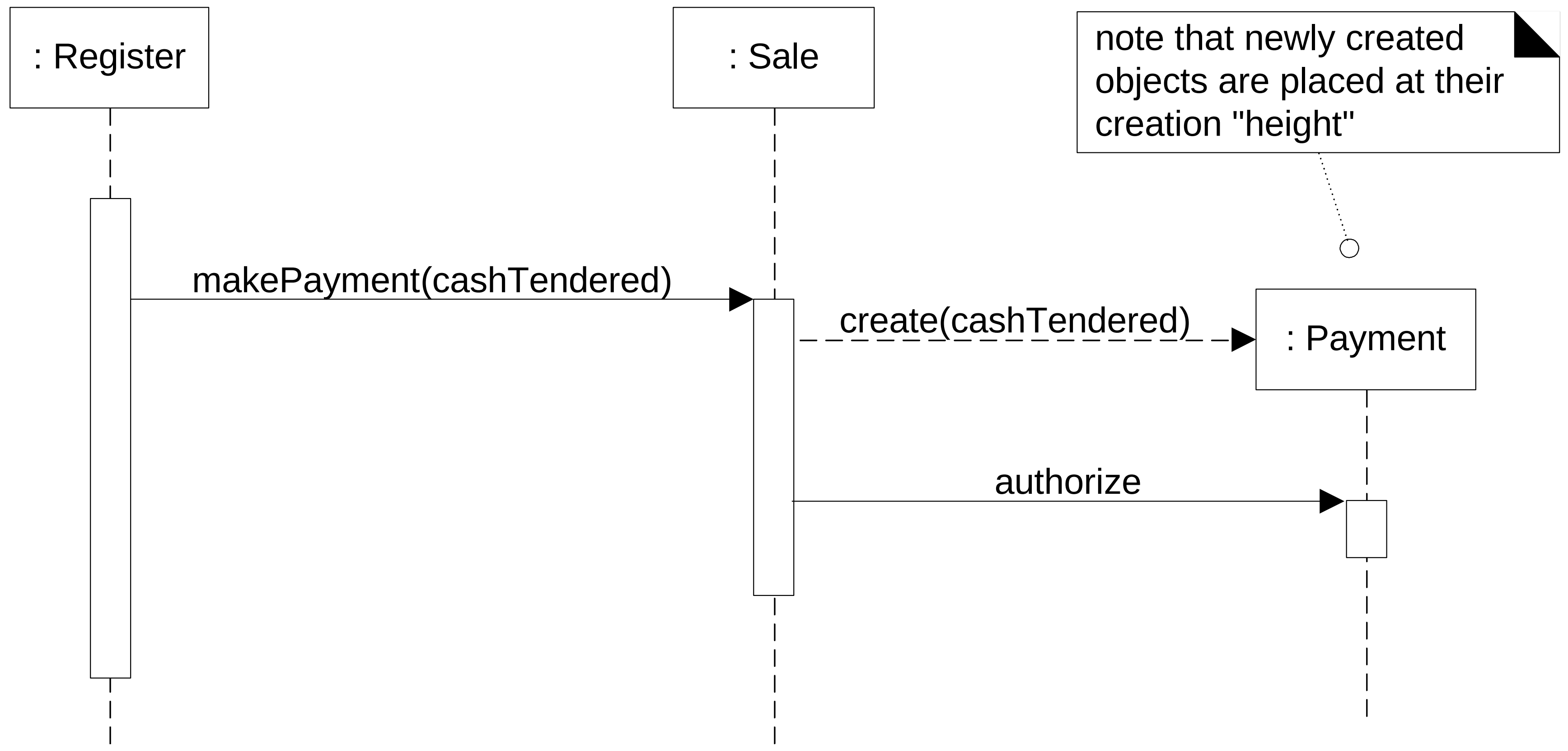
Showing Return Results



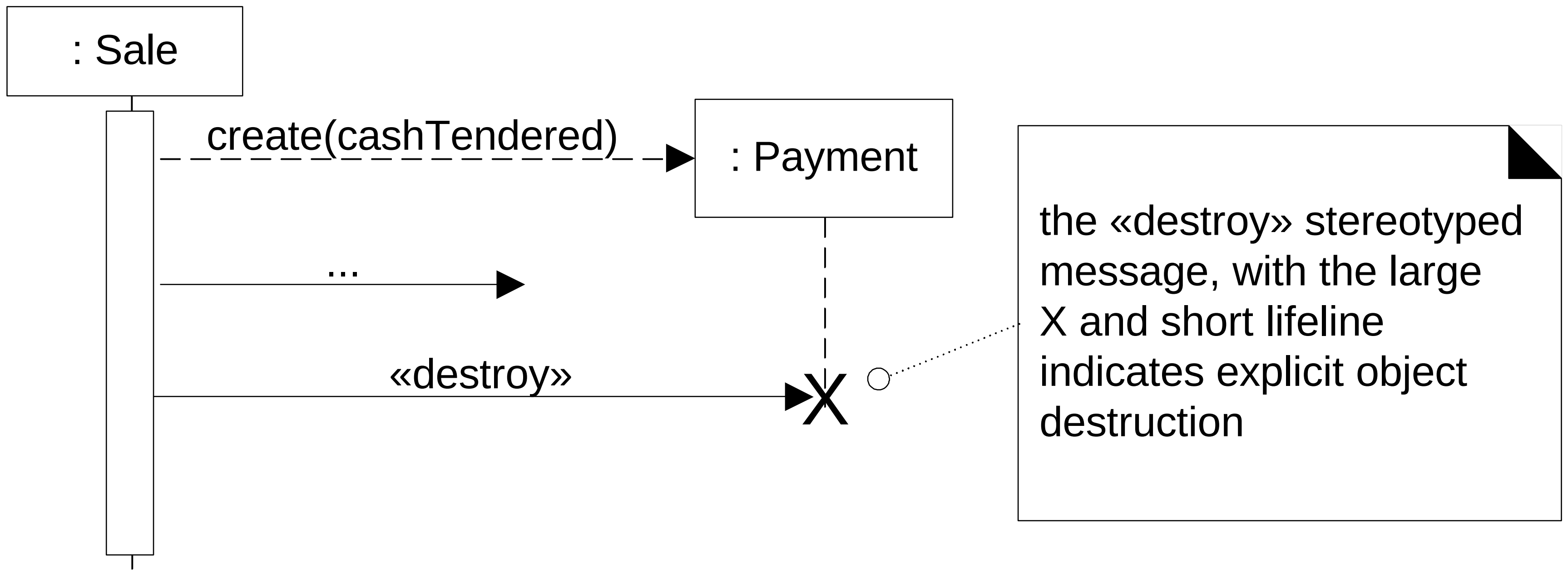
Messages to Self (*this*)



Object Creation and Lifelines

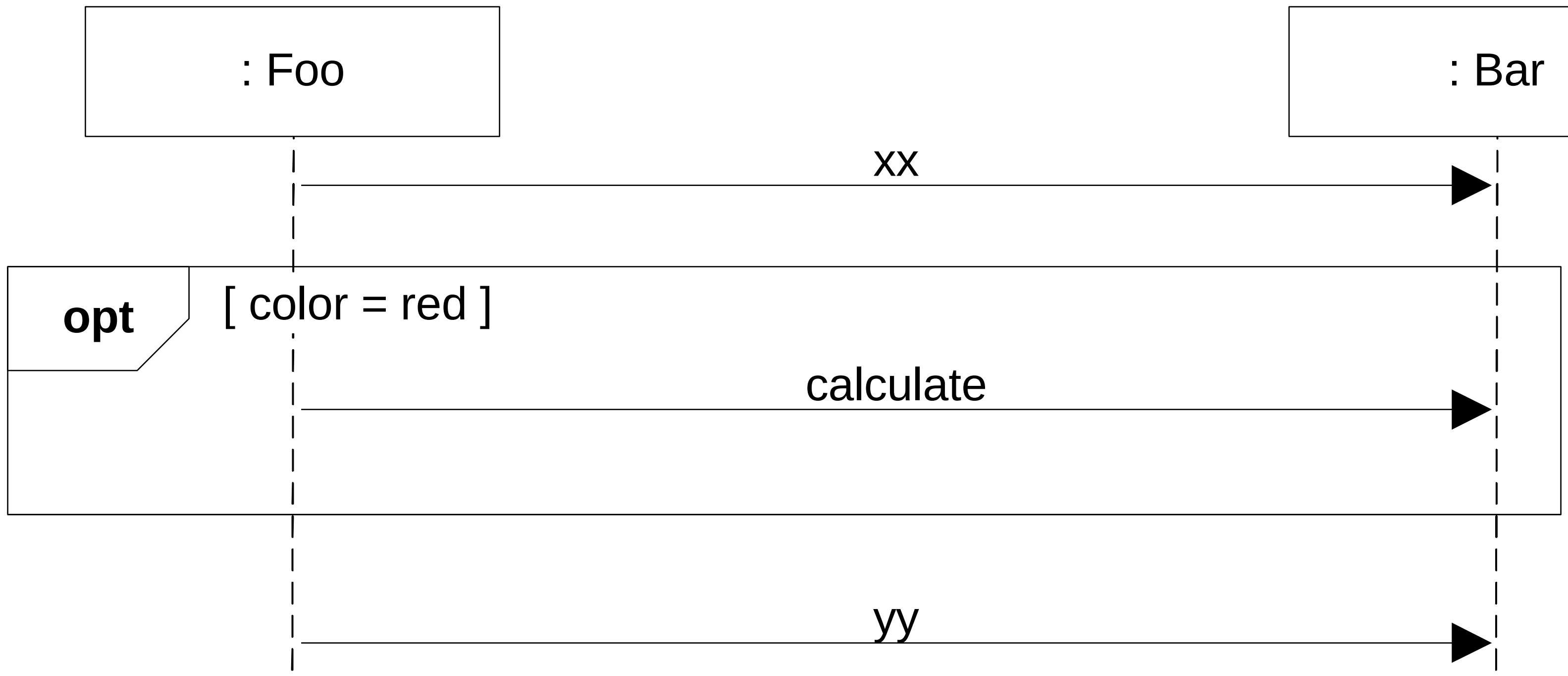

: Register
: Sale
note that newly created objects are placed at their creation "height"
makePayment(cashTendered)
create(cashTendered)
: Payment
authorize

Object Destruction

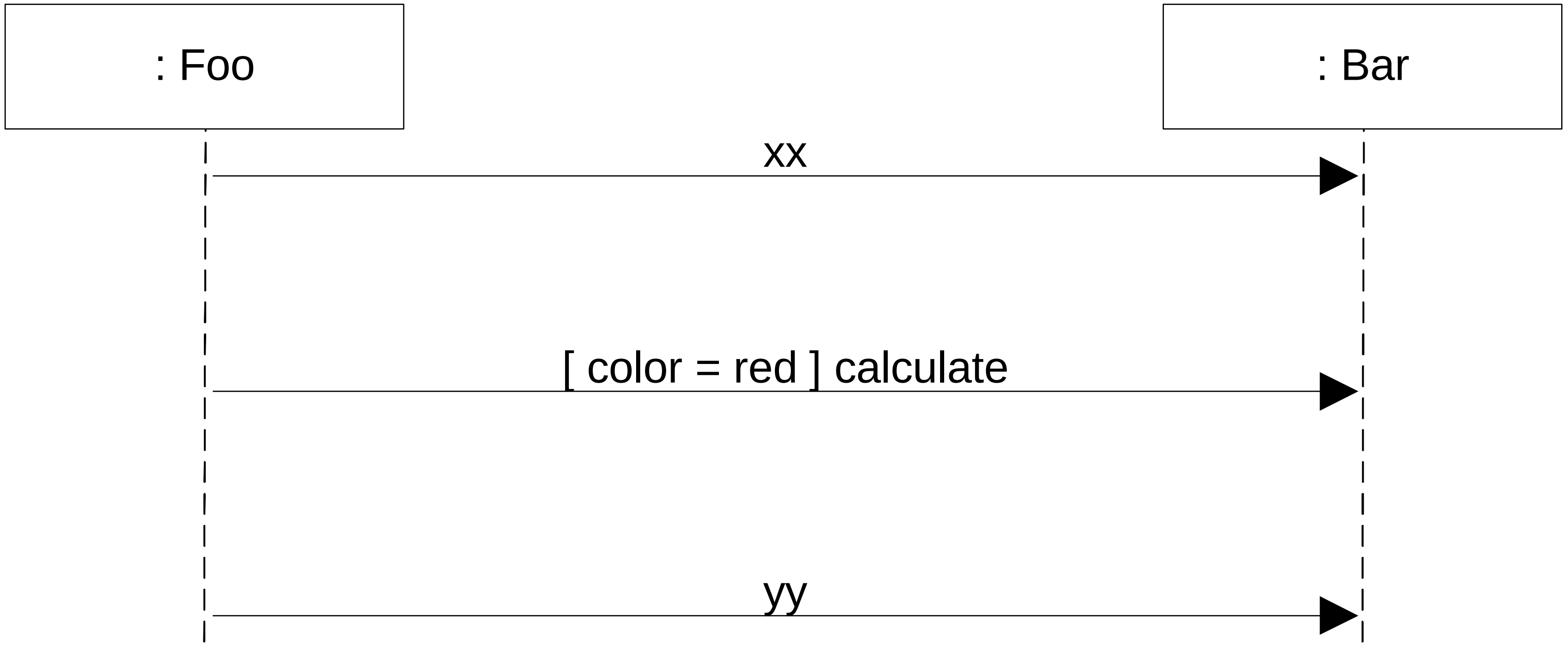


the «destroy» stereotyped message, with the large X and short lifeline indicates explicit object destruction

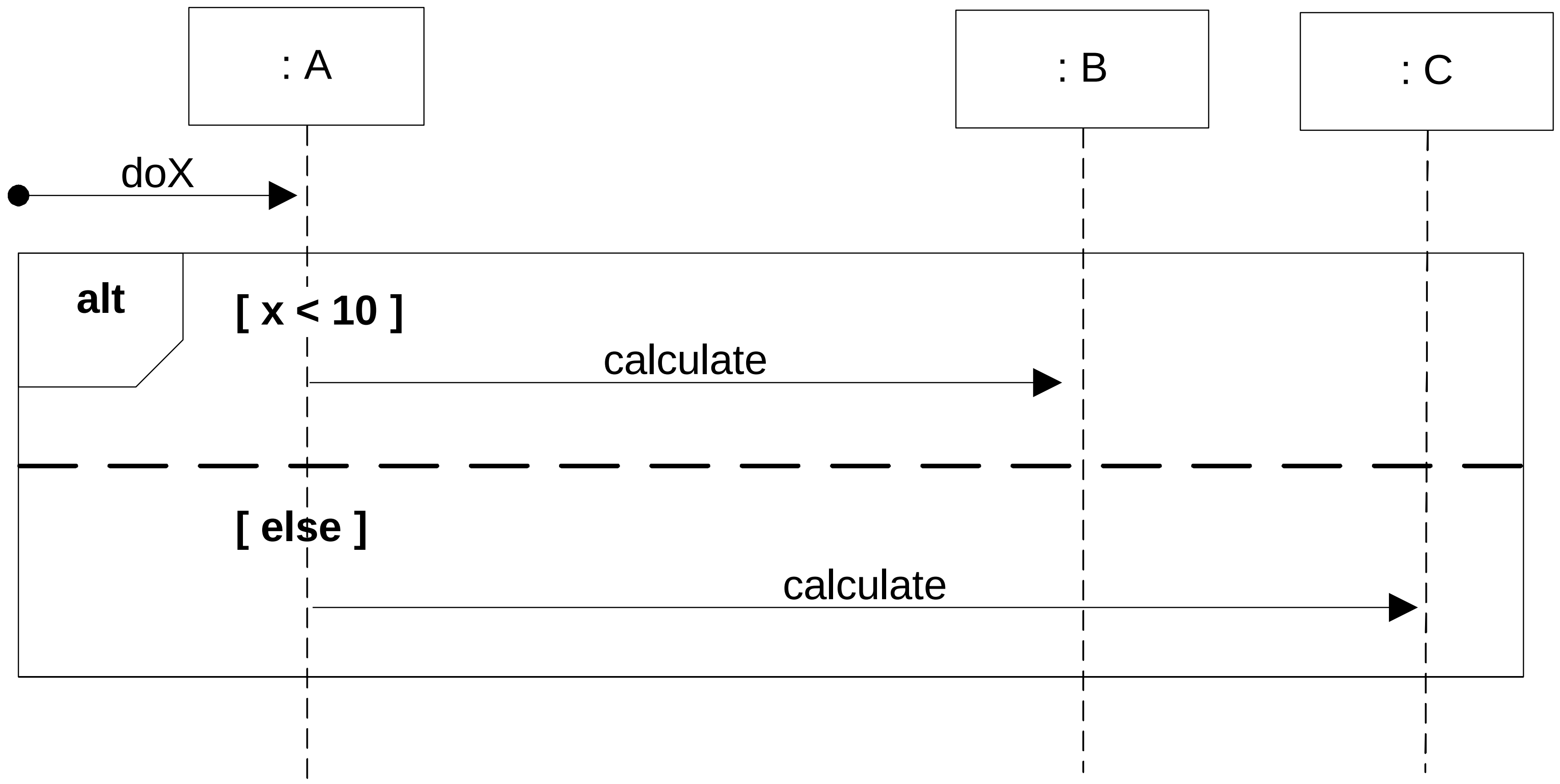
UML Frames: opt (optional)



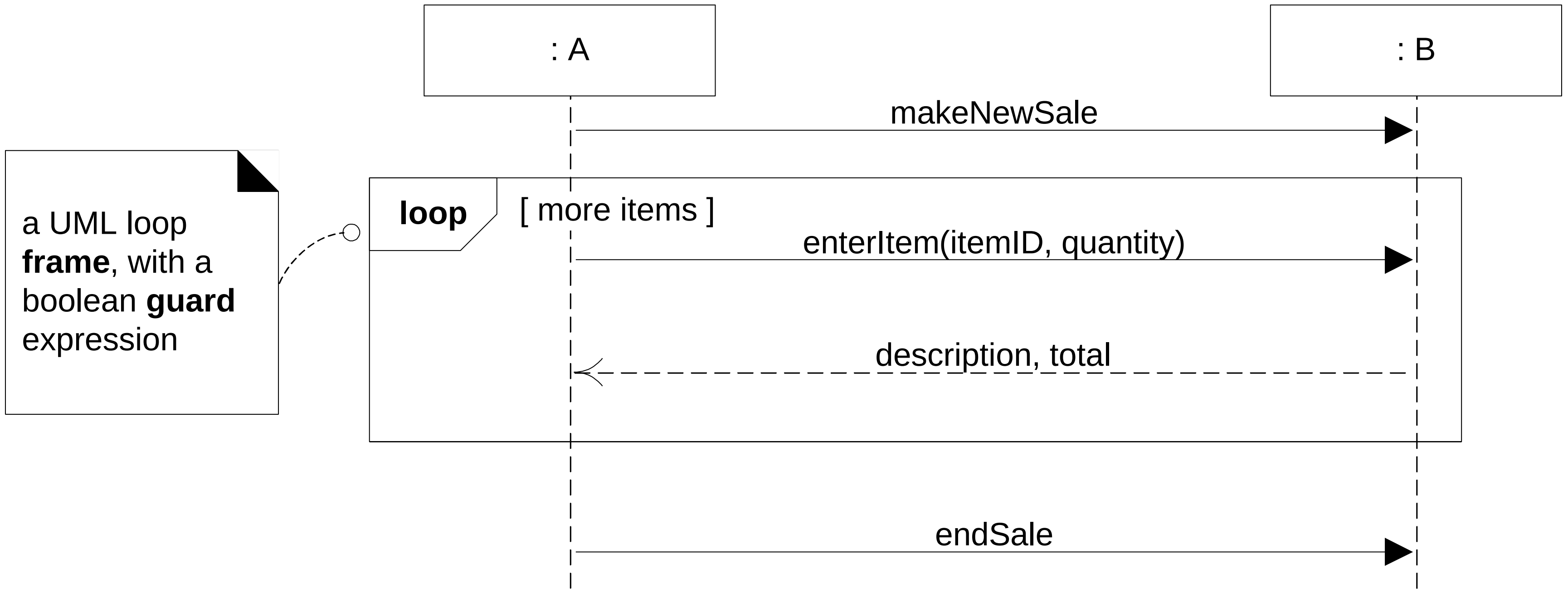
UML 1.x: Conditional Message



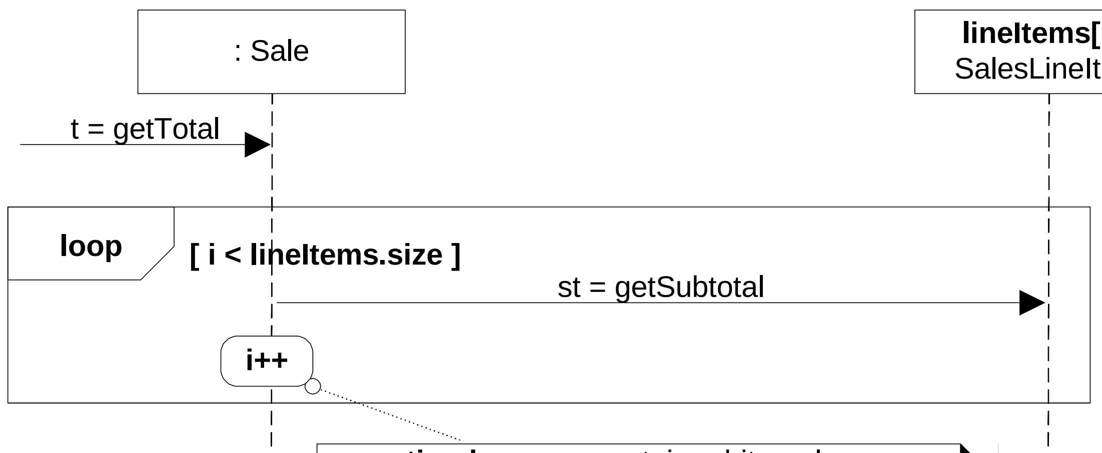
UML Frames: alt (alternatives)



UML Frames: loop


: A
: B
makeNewSale
a UML loop **frame**, with a boolean **guard** expression
loop
[more items]
enterItem(itemID, quantity)
description, total
endSale

UML Frames: loop–Collections (1)



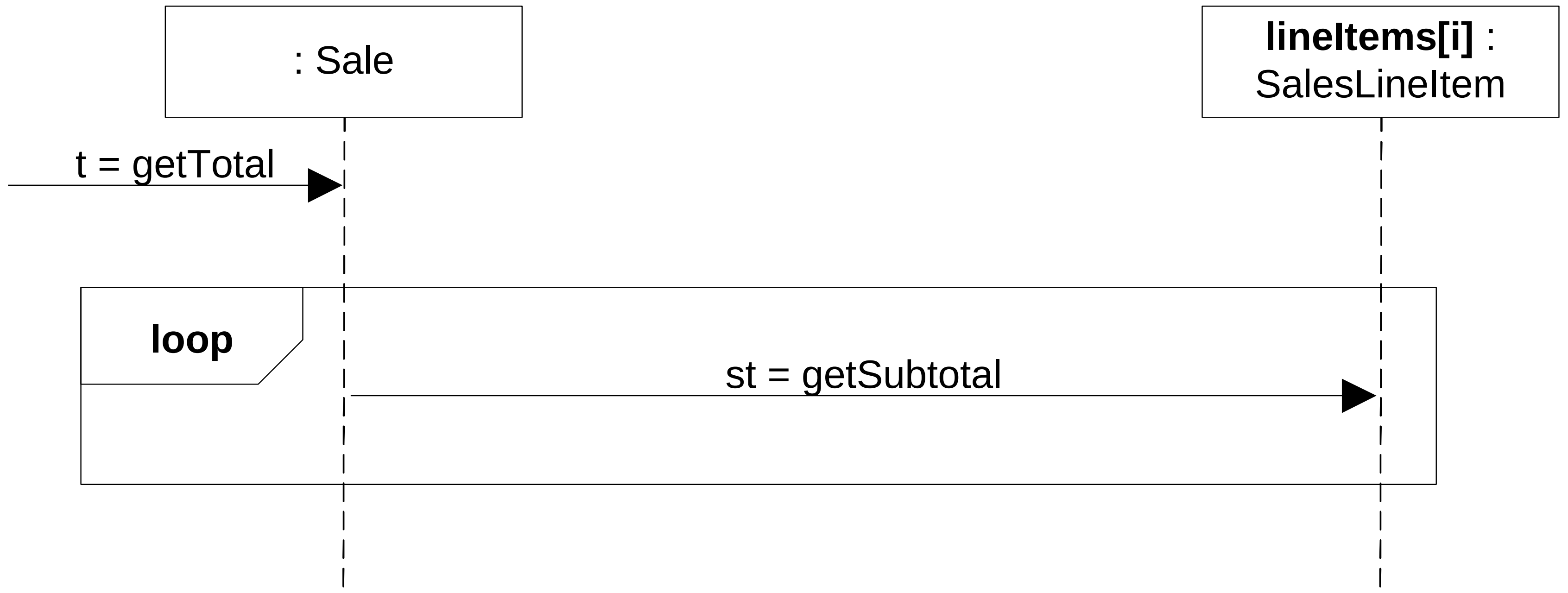
This lifeline box represents one instance from a collection of many *SalesLineItem* objects.

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

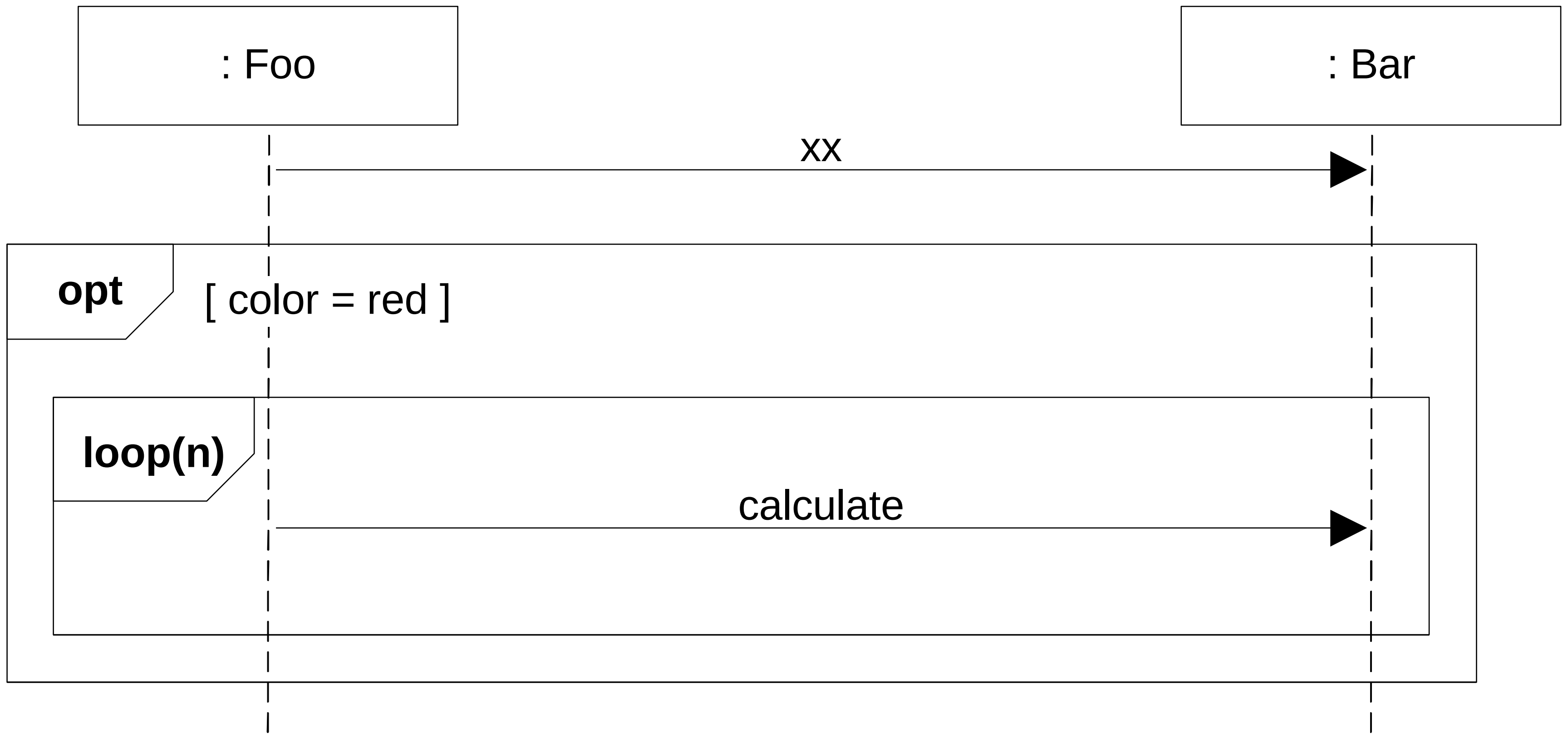
an **action box** may contain arbitrary language statements (in this case, incrementing 'i')

it is placed over the lifeline to which it applies

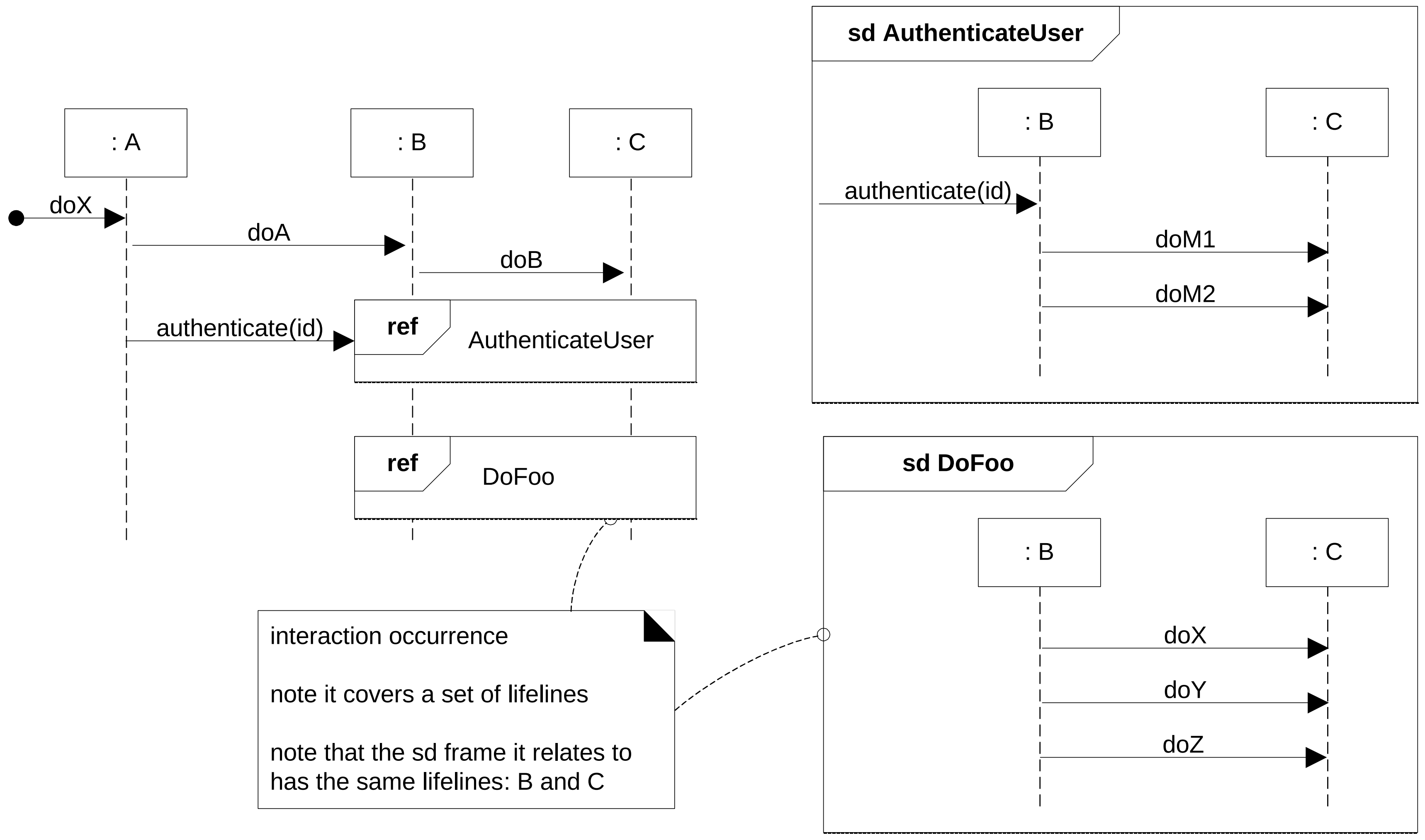
UML Frames: loop–Collections (2)



UML Frames: Nesting



UML Frames: sd/ref (define/refer)



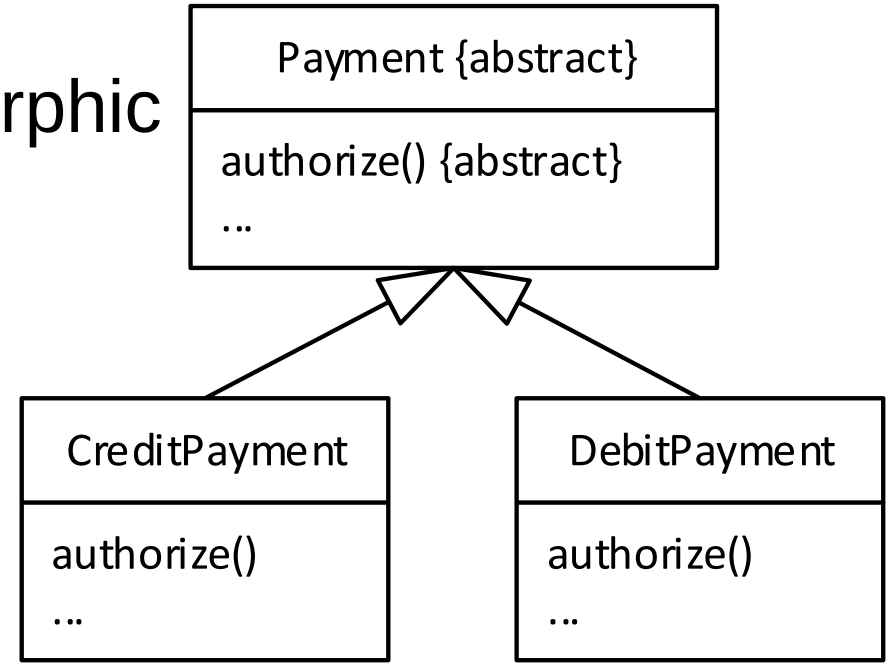
interaction occurrence

note it covers a set of lifelines

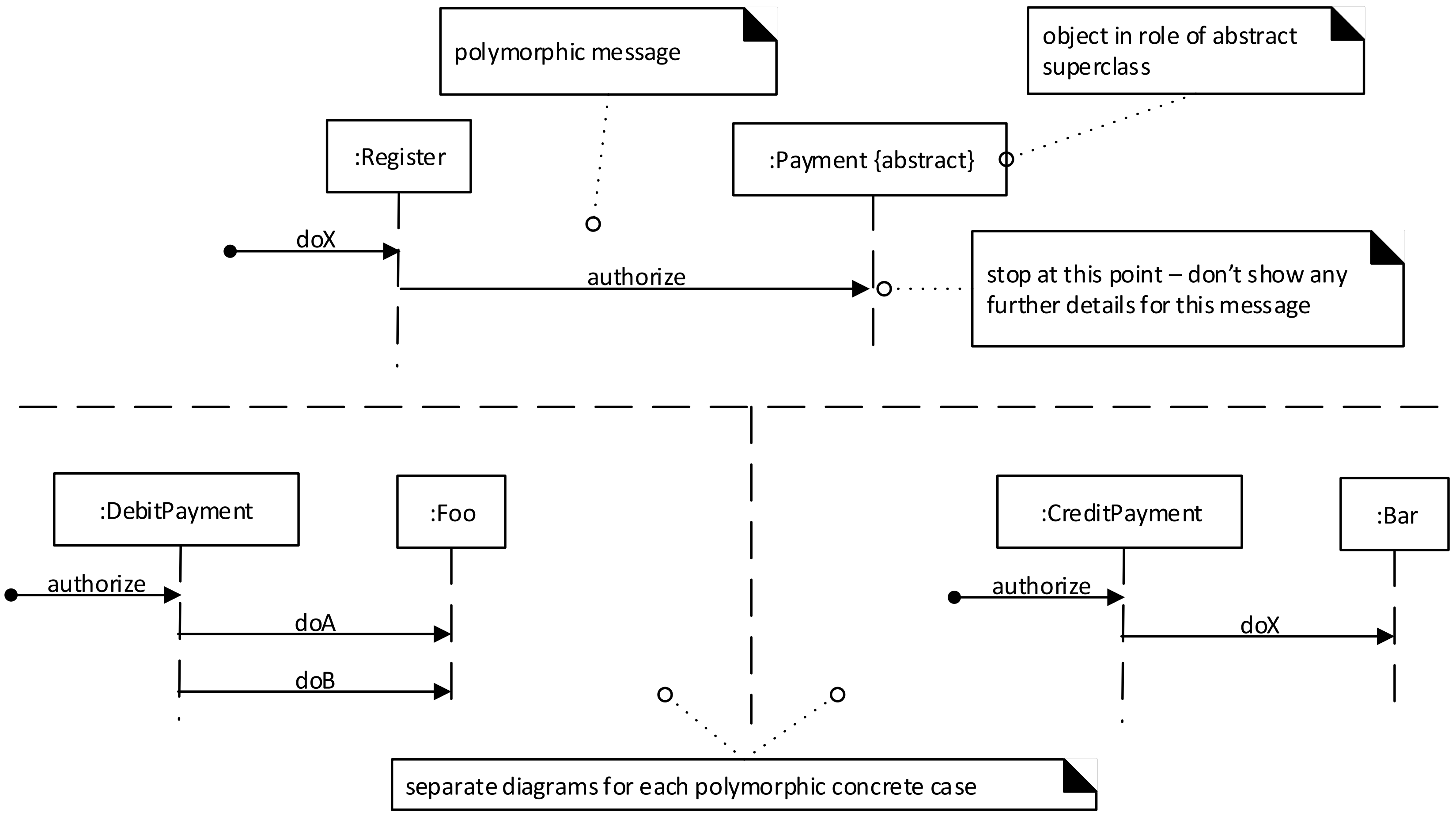
note that the sd frame it relates to has the same lifelines: B and C

Polymorphic Cases (1)

- *Payment* is an abstract superclass
- *CreditPayment, DebitPayment* are concrete subclasses
  - both implement polymorphic operation *authorize*



Polymorphic Cases (2)


polymorphic message
object in role of abstract superclass
:Register
:Payment {abstract}
doX
authorize
stop at this point – don't show any further details for this message
:DebitPayment
:Foo
authorize
doA
doB
:CreditPayment
:Bar
authorize
doX
separate diagrams for each polymorphic concrete case

Asynchronous Calls & Active Objects

a stick arrow in UML implies an asynchronous call

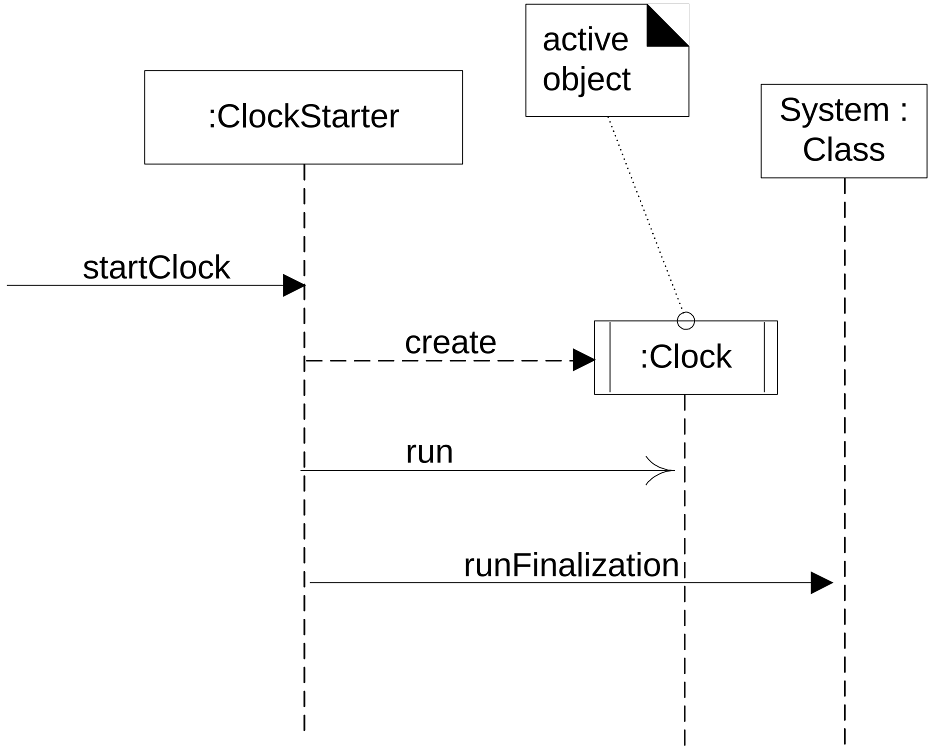
a filled arrow is the more common synchronous call

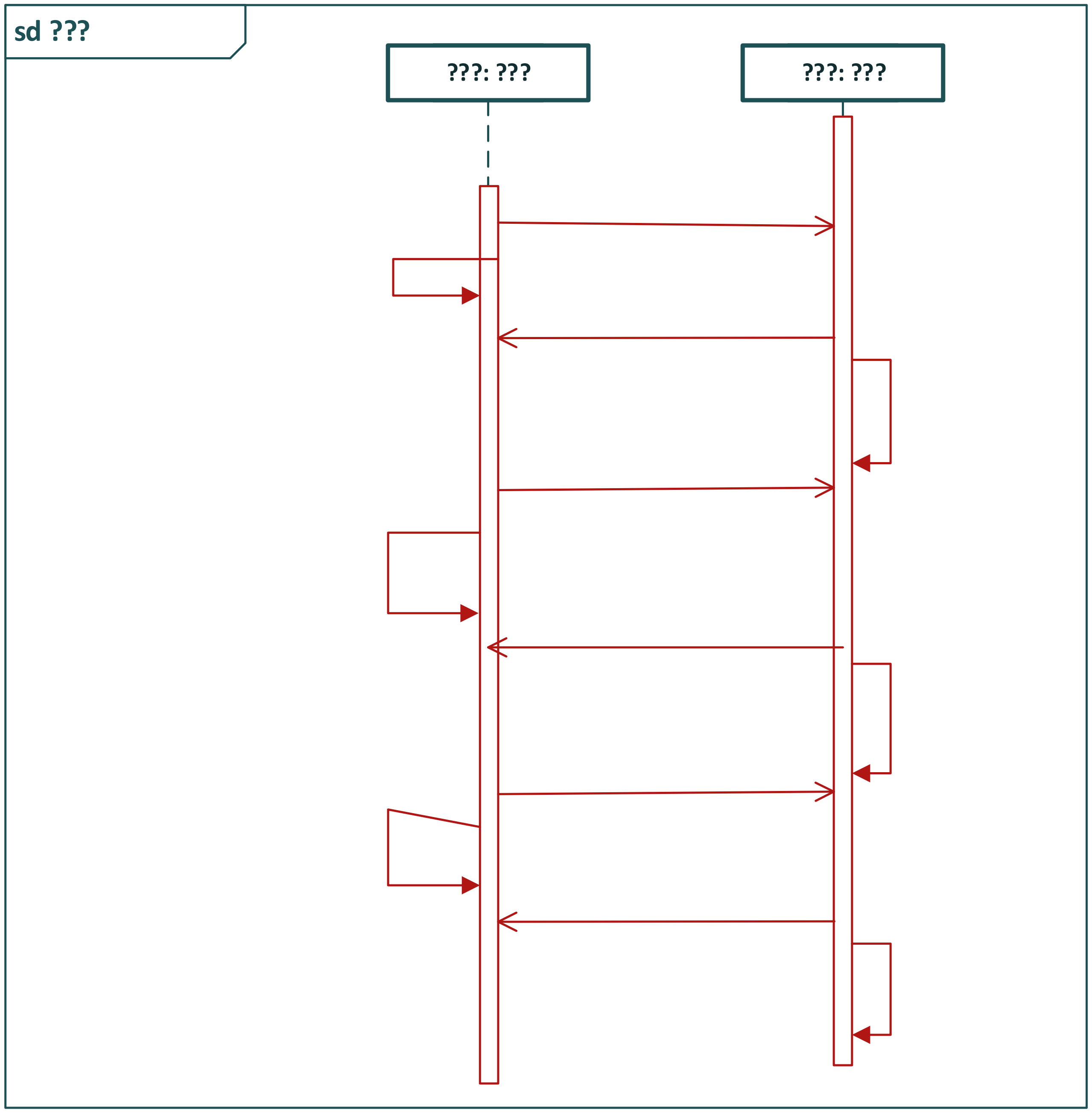
In Java, for example, an asynchronous call may occur as follows:

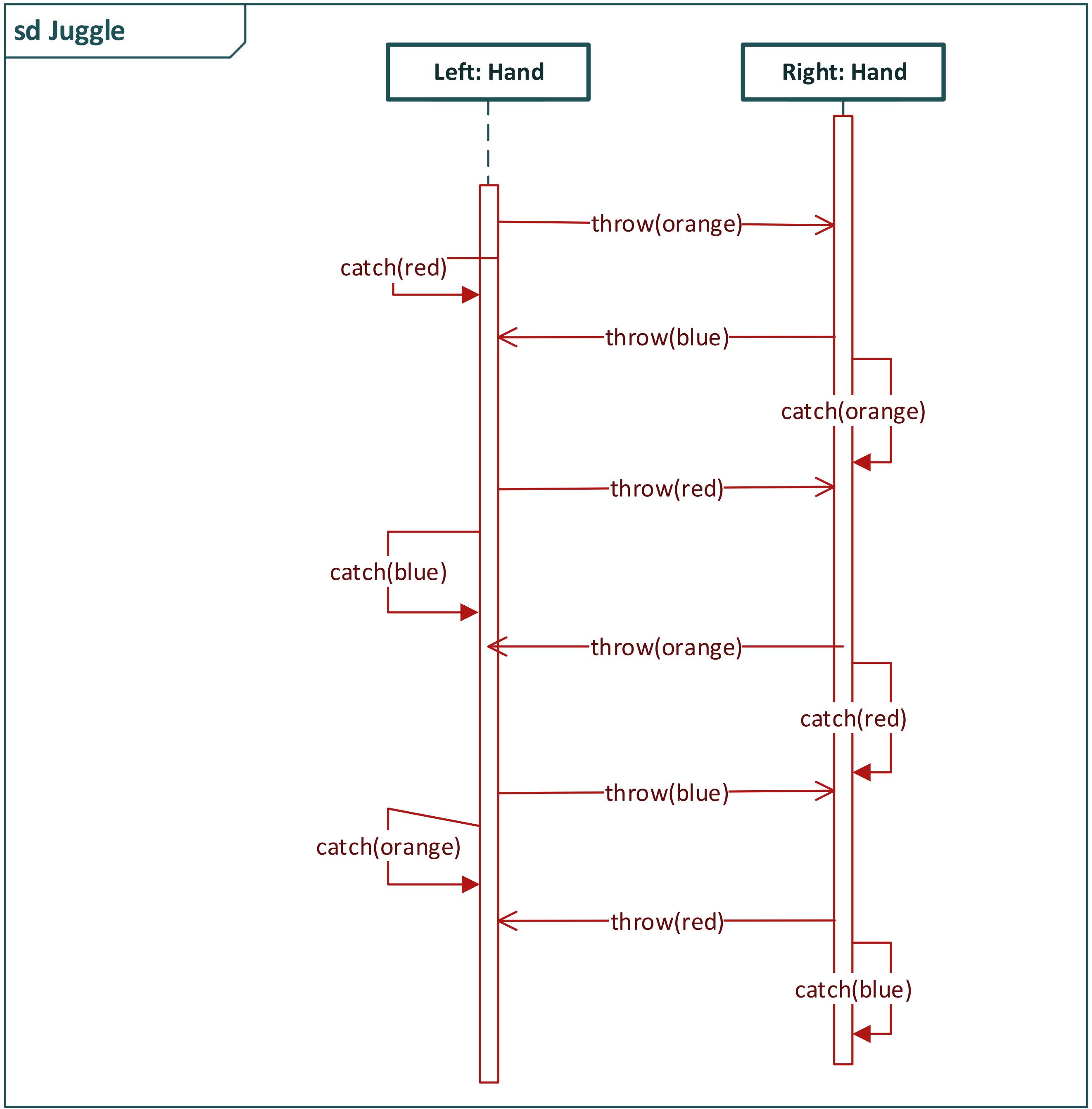
```
// Clock implements the Runnable interface
Thread t = new Thread( new Clock() );
t.start();
```

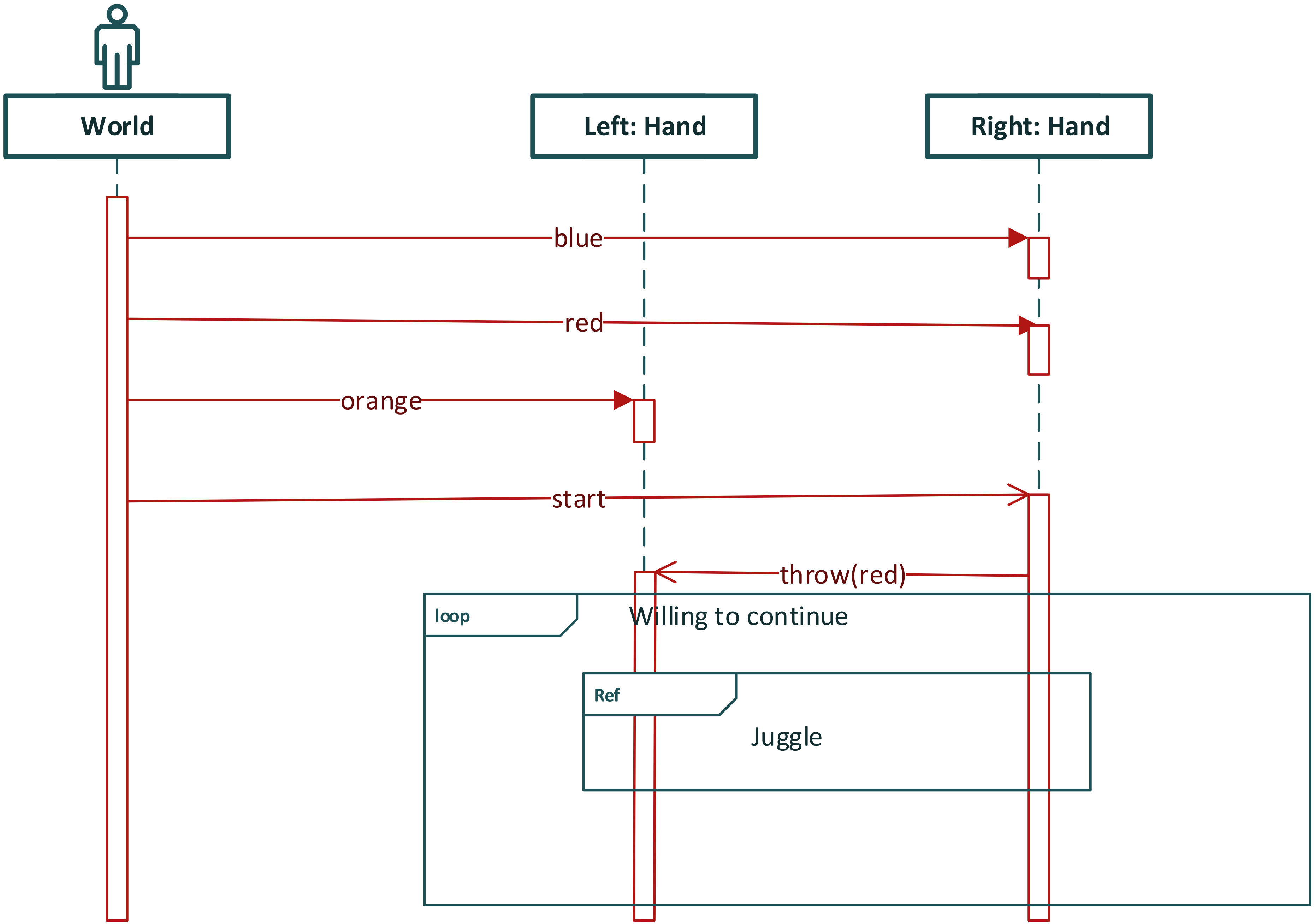
the asynchronous *start* call always invokes the *run* method on the *Runnable* (*Clock*) object

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




sd ???
???: ???
???: ???

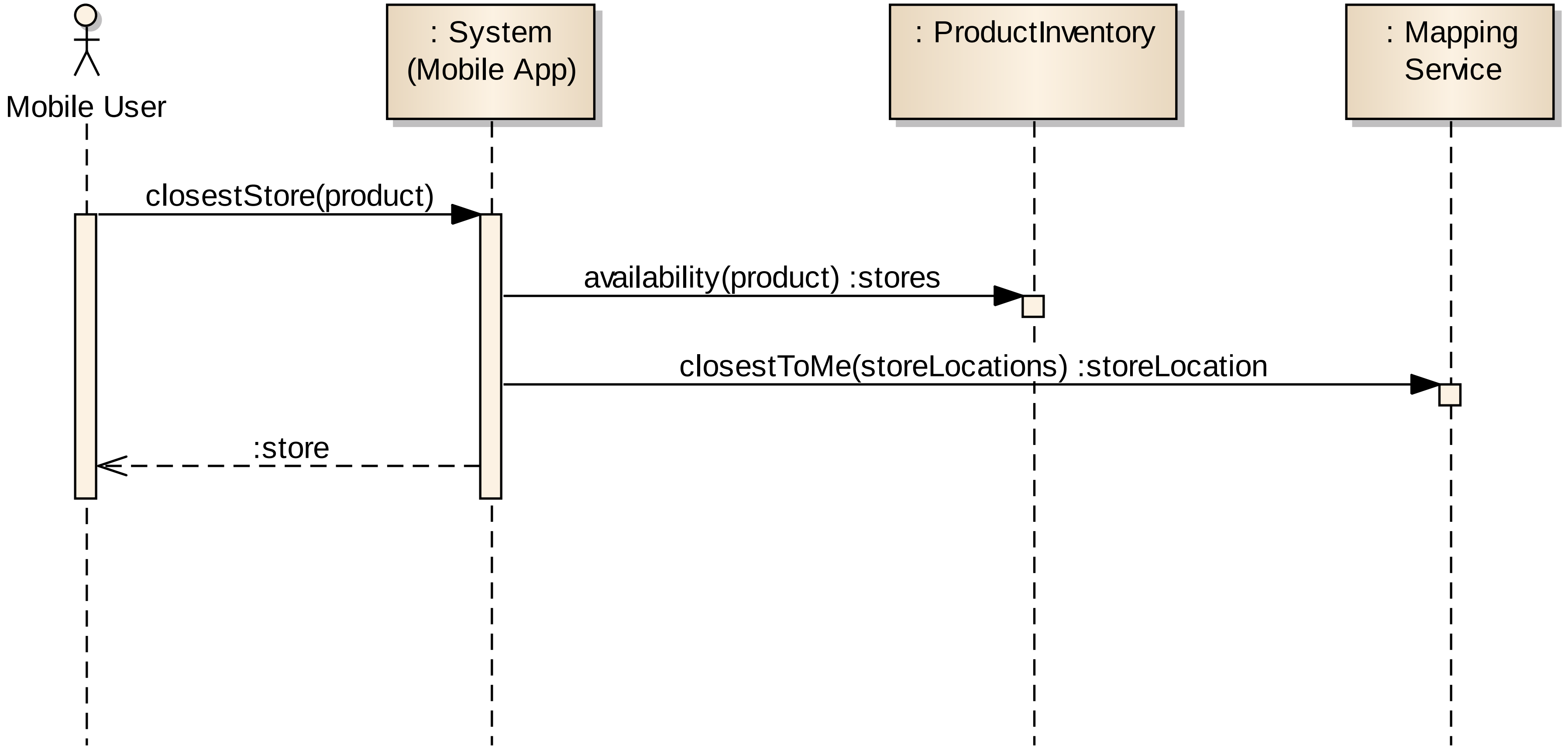

sd Juggle
Left: Hand
Right: Hand
throw(orange)
catch(red)
throw(blue)
catch(orange)
throw(red)
catch(blue)
throw(orange)
catch(red)
throw(blue)
catch(orange)
throw(red)
catch(blue)


World
Left: Hand
Right: Hand
blue
red
orange
start
throw(red)
loop
Willing to continue
Ref
Juggle

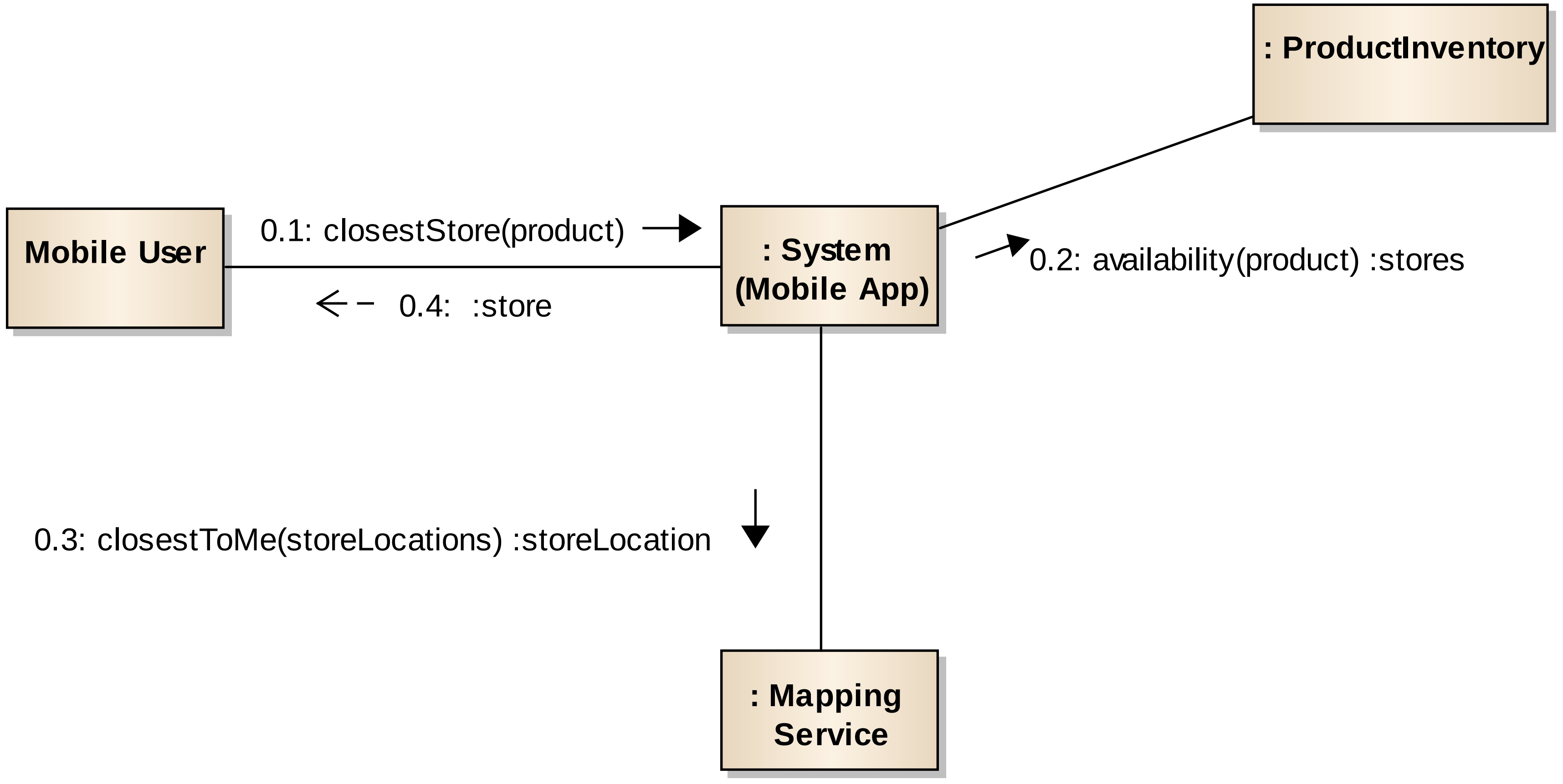
Sequence vs. Communications Diags.

Type	Strengths	Weaknesses
sequence	Clearly shows time ordering of messages Can more easily convey the detail of message protocols between objects	Linear layout of instances can obscure relationships Linear layout consumes horizontal space
communications	More layout options Clearly shows relationships between object instances Can combine scenarios to provide a more complete picture	More difficult to see message sequencing Fewer notation options for expressing message patterns

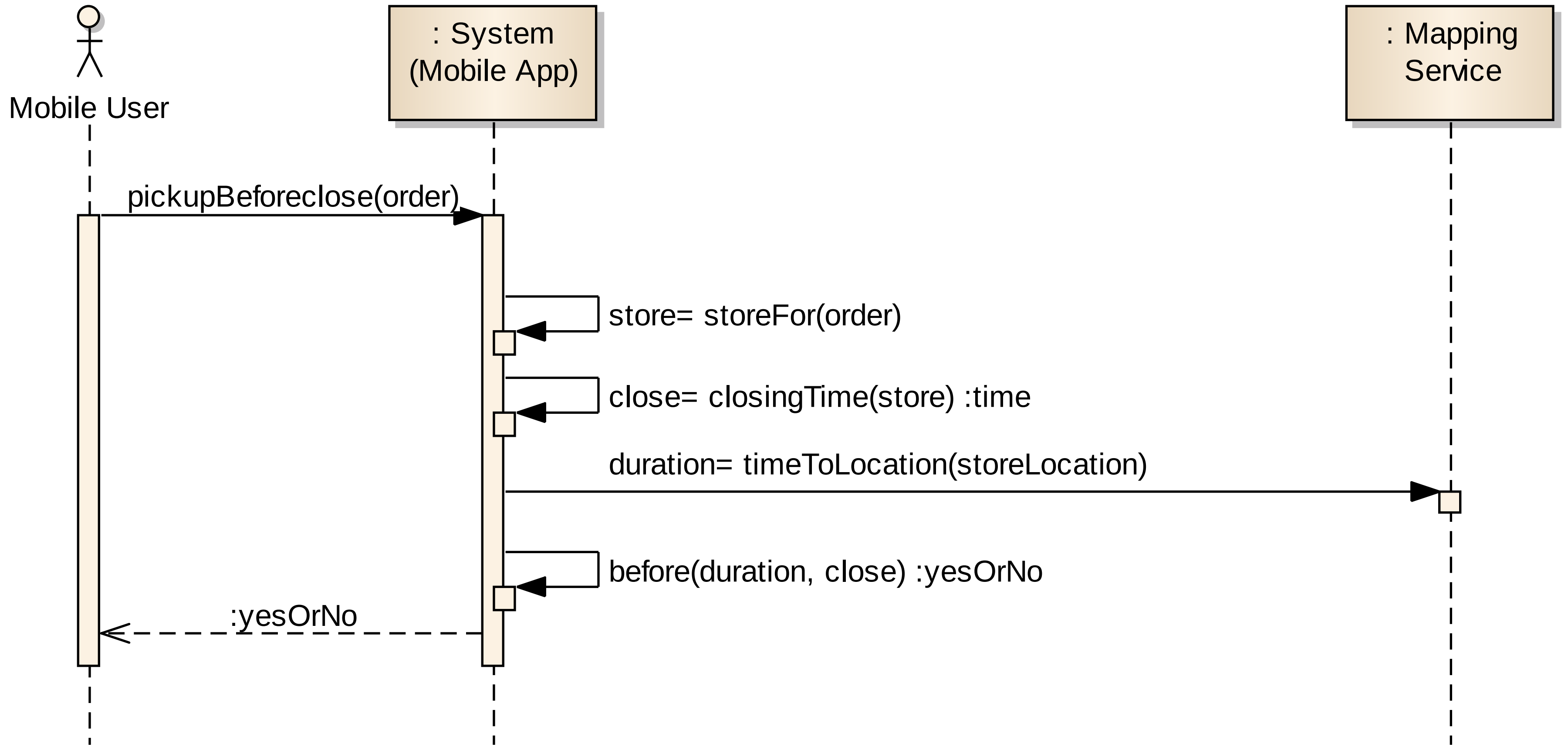
Mobile App: SSD closestStore



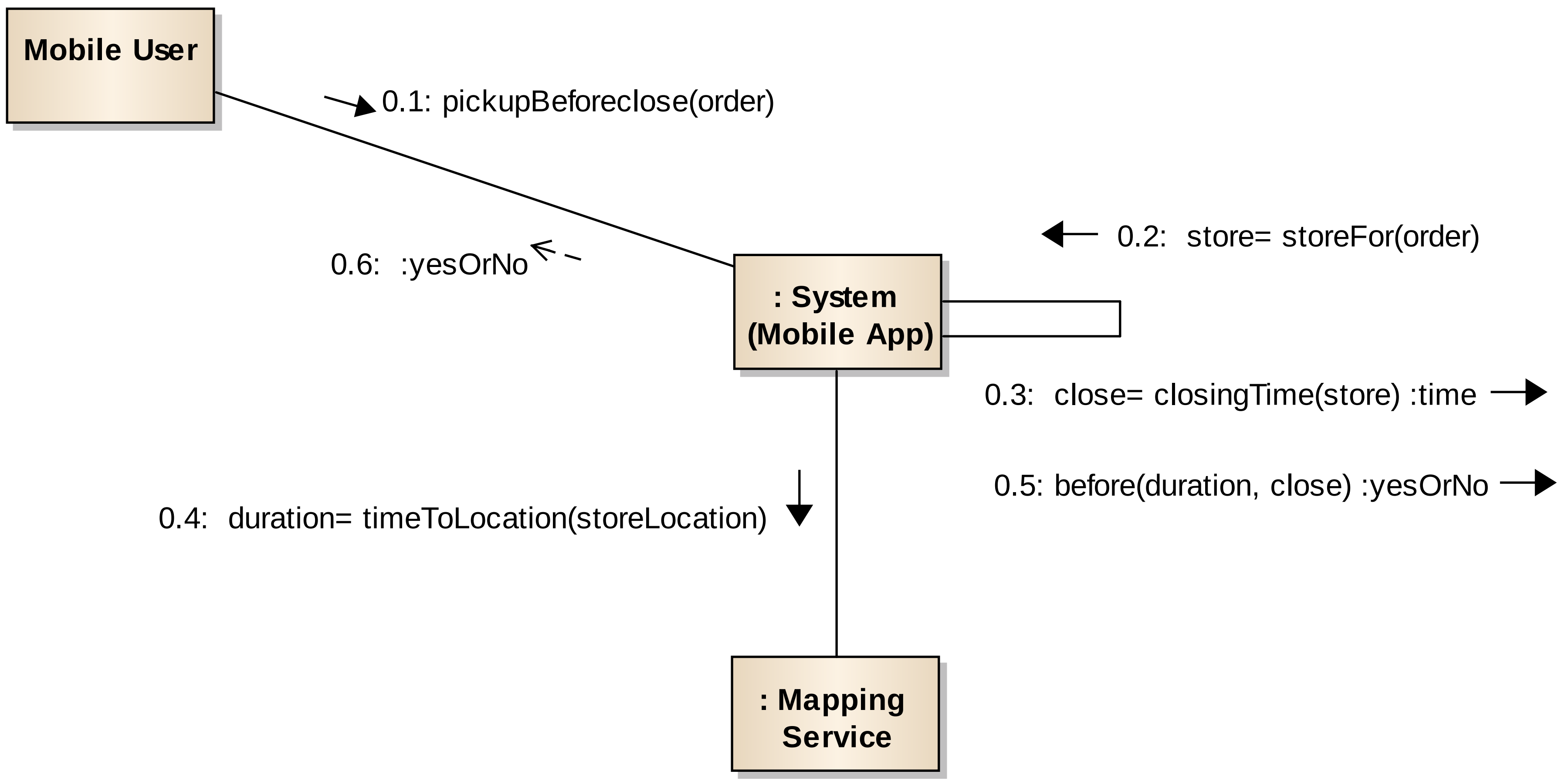
Mobile App: SCD closestStore


: ProductInventory
Mobile User
0.1: closestStore(product)
: System (Mobile App)
0.2: availability(product) :stores
0.4: :store
0.3: closestToMe(storeLocations) :storeLocation
: Mapping Service

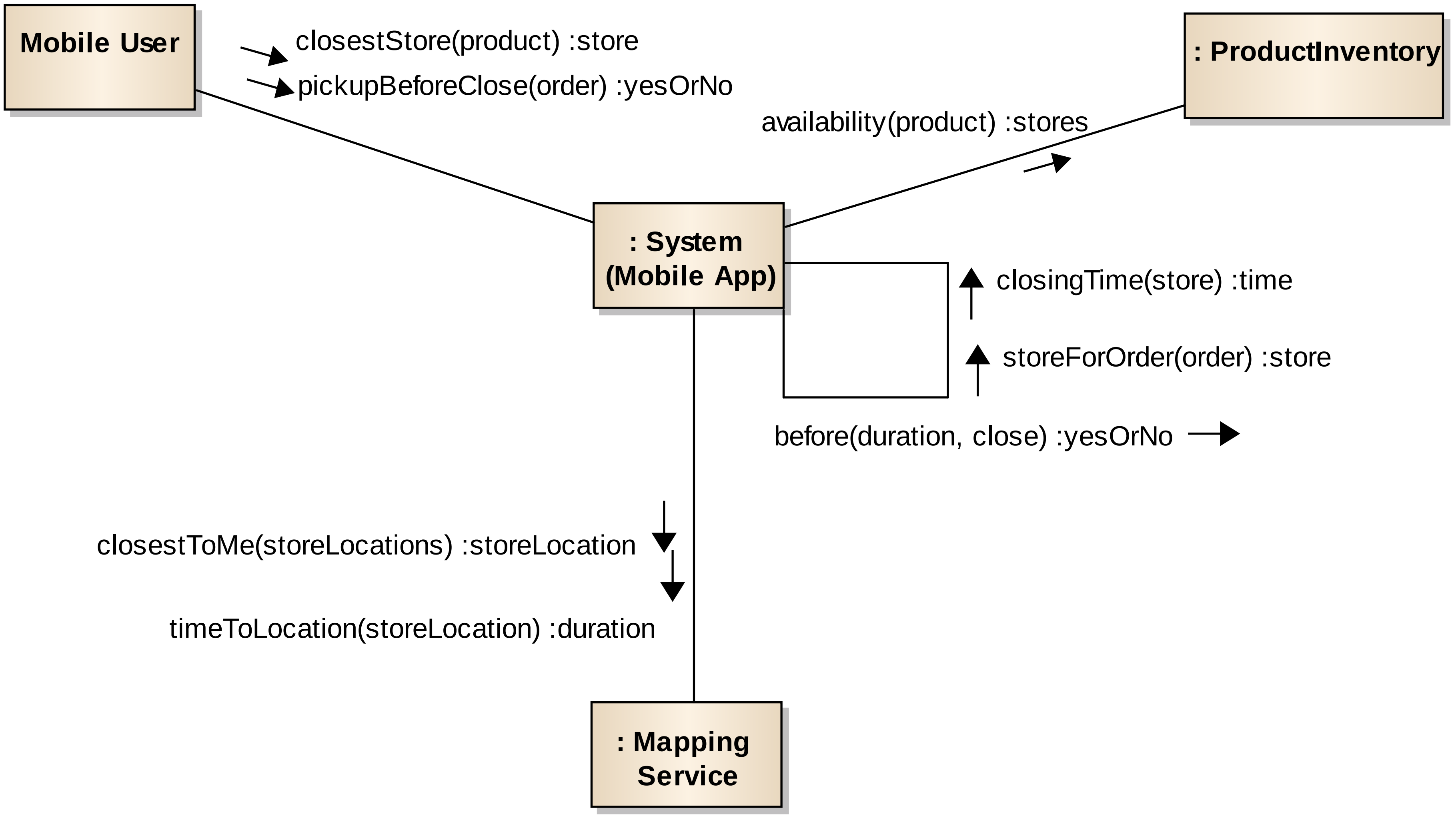
Mobile App: SSD pickupBeforeclose



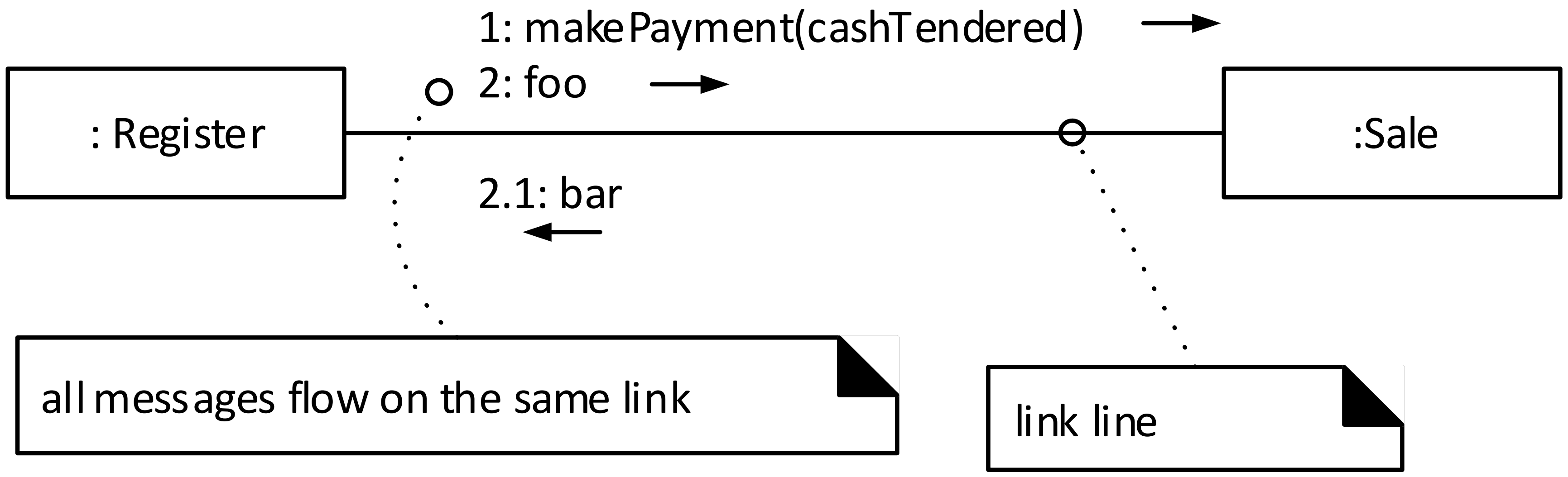
Mobile App: SCD pickupBeforeclose


Mobile User
0.1: pickupBeforeclose(order)
0.6: :yesOrNo
0.2: store= storeFor(order)
: System (Mobile App)
0.3: close= closingTime(store) :time
0.5: before(duration, close) :yesOrNo
0.4: duration= timeToLocation(storeLocation)
: Mapping Service

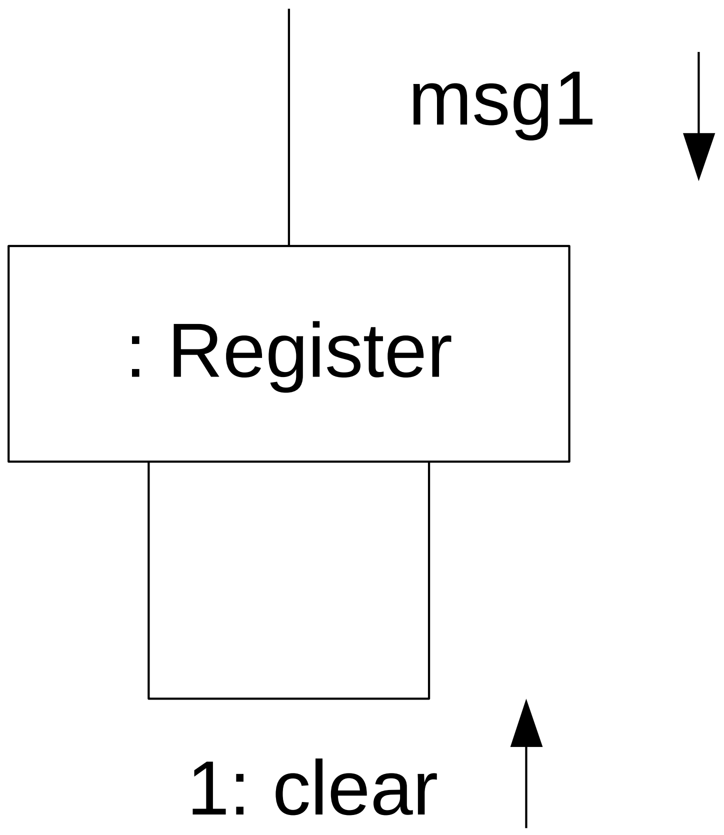
Mobile App: SCD combined


Mobile User
closestStore(product) :store
pickupBeforeClose(order) :yesOrNo
: ProductInventory
availability(product) :stores
: System (Mobile App)
closingTime(store) :time
storeForOrder(order) :store
before(duration, close) :yesOrNo
closestToMe(storeLocations) :storeLocation
timeToLocation(storeLocation) :duration
: Mapping Service

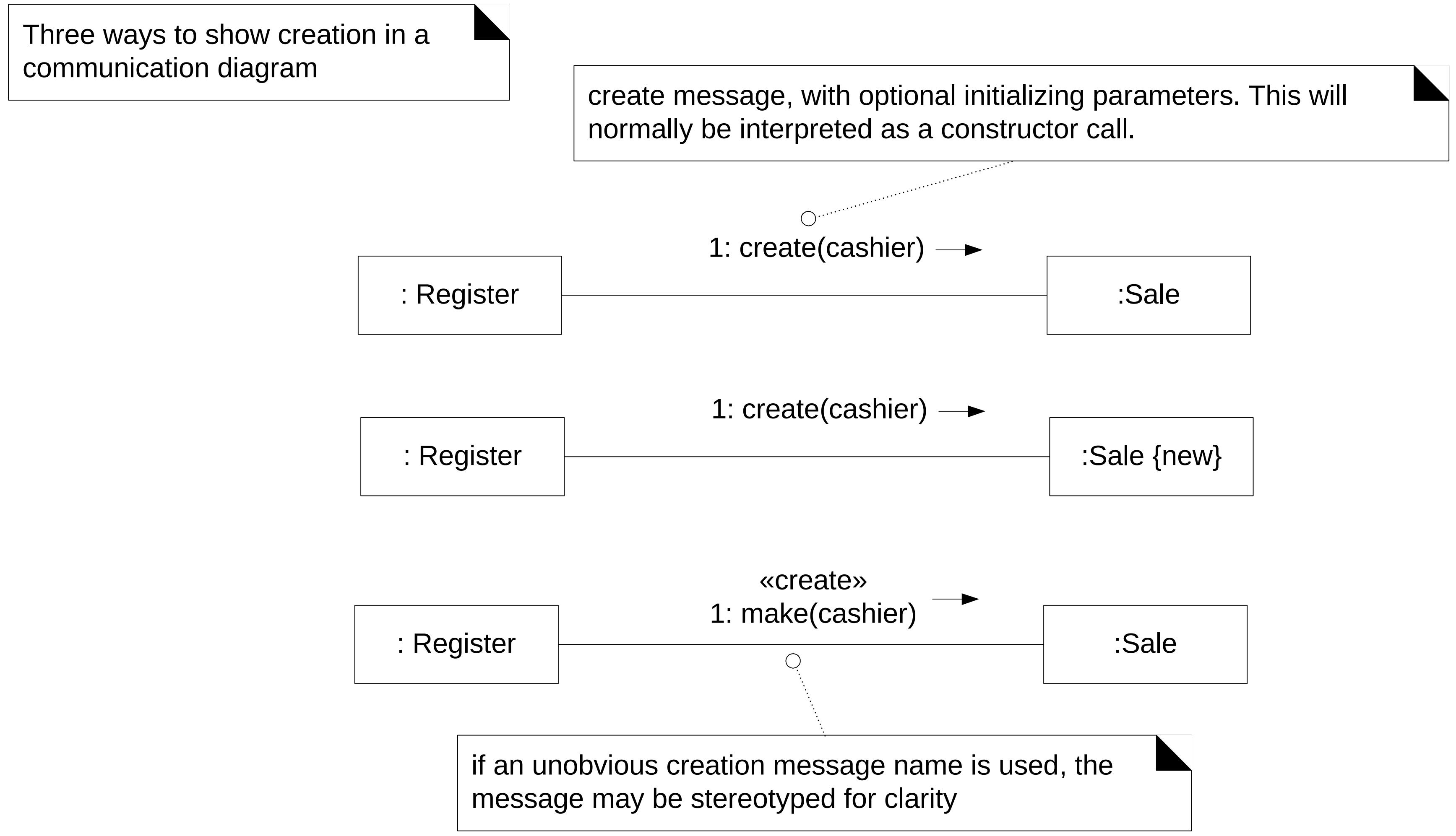
Communication Diagrams


1: makePayment(cashTendered)
2: foo
: Register
:Sale
2.1: bar
all messages flow on the same link
link line

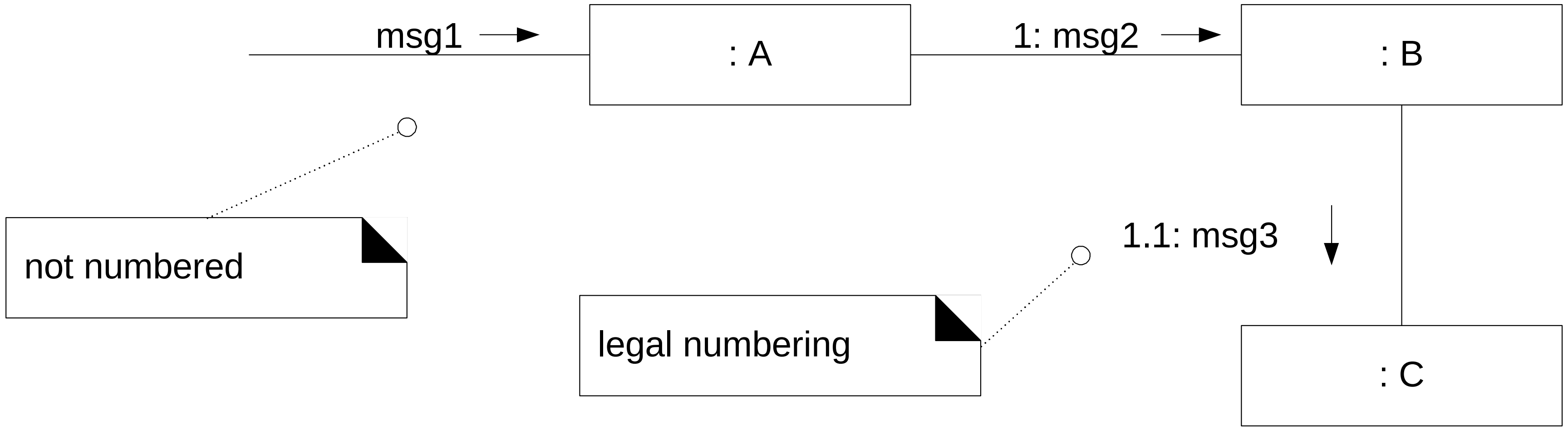
CD: Messages to “this”



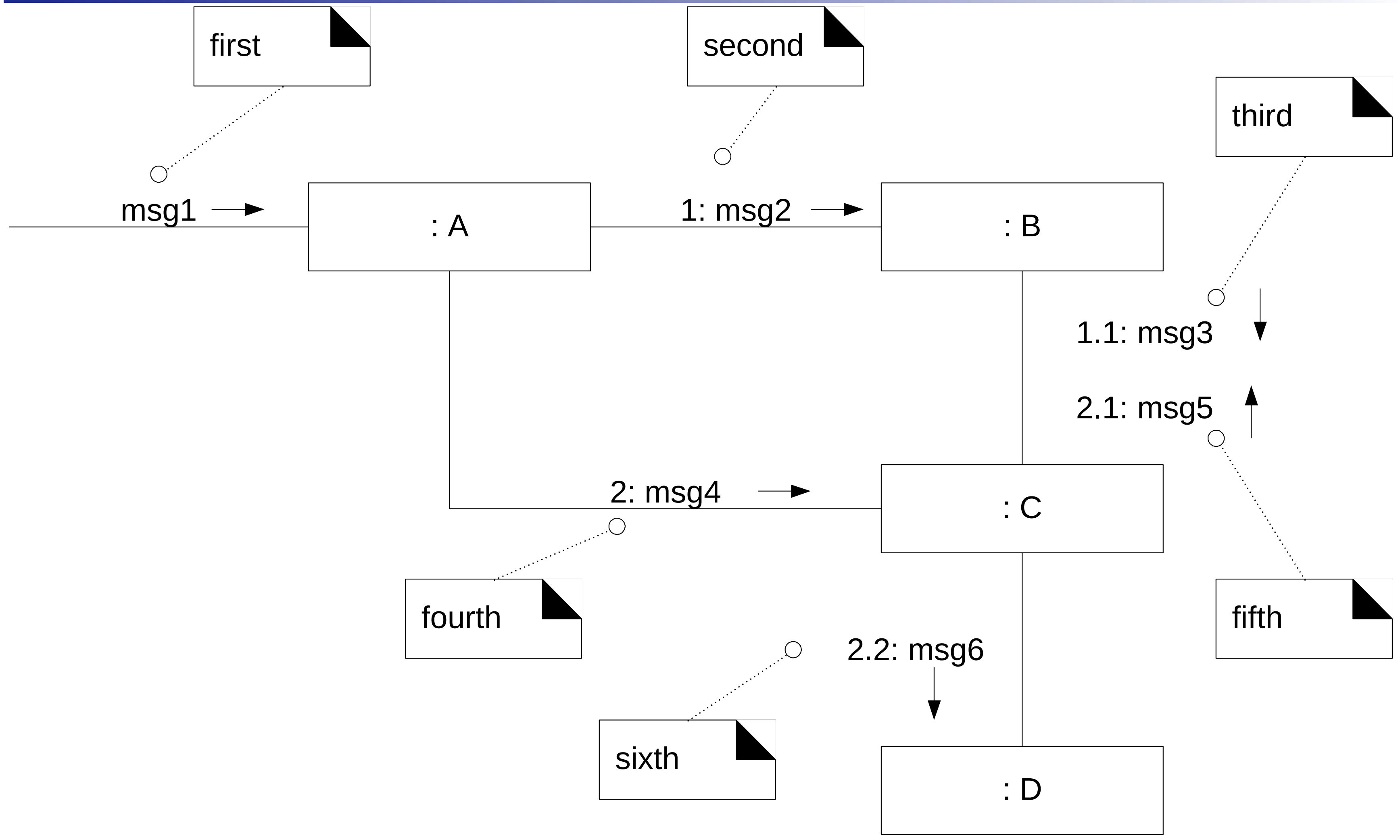
CD: Object Creation


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

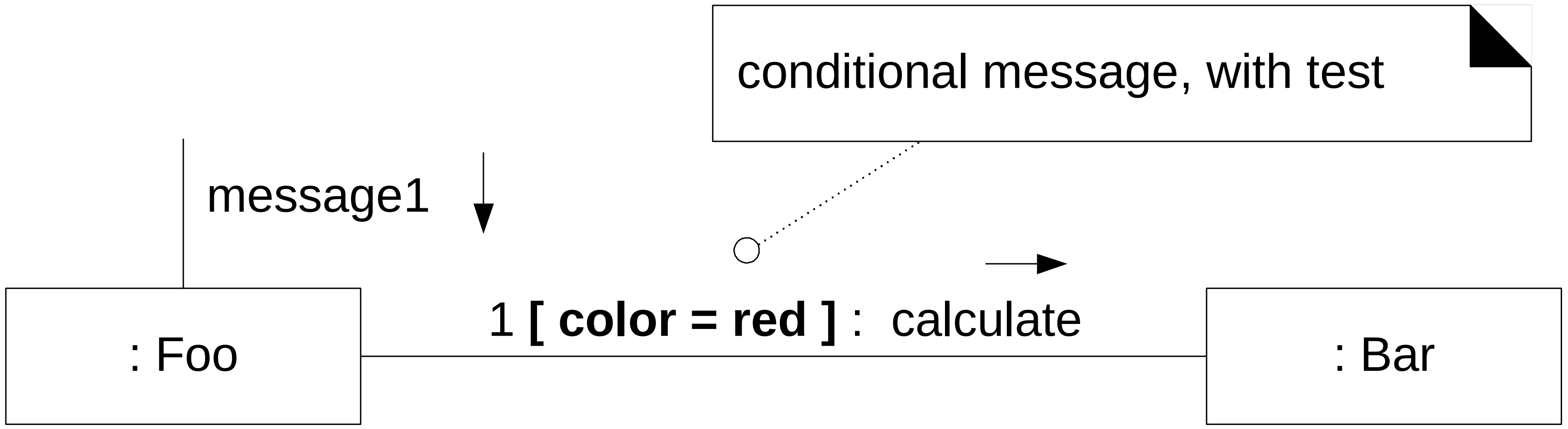
CD: Message Sequence Numbers



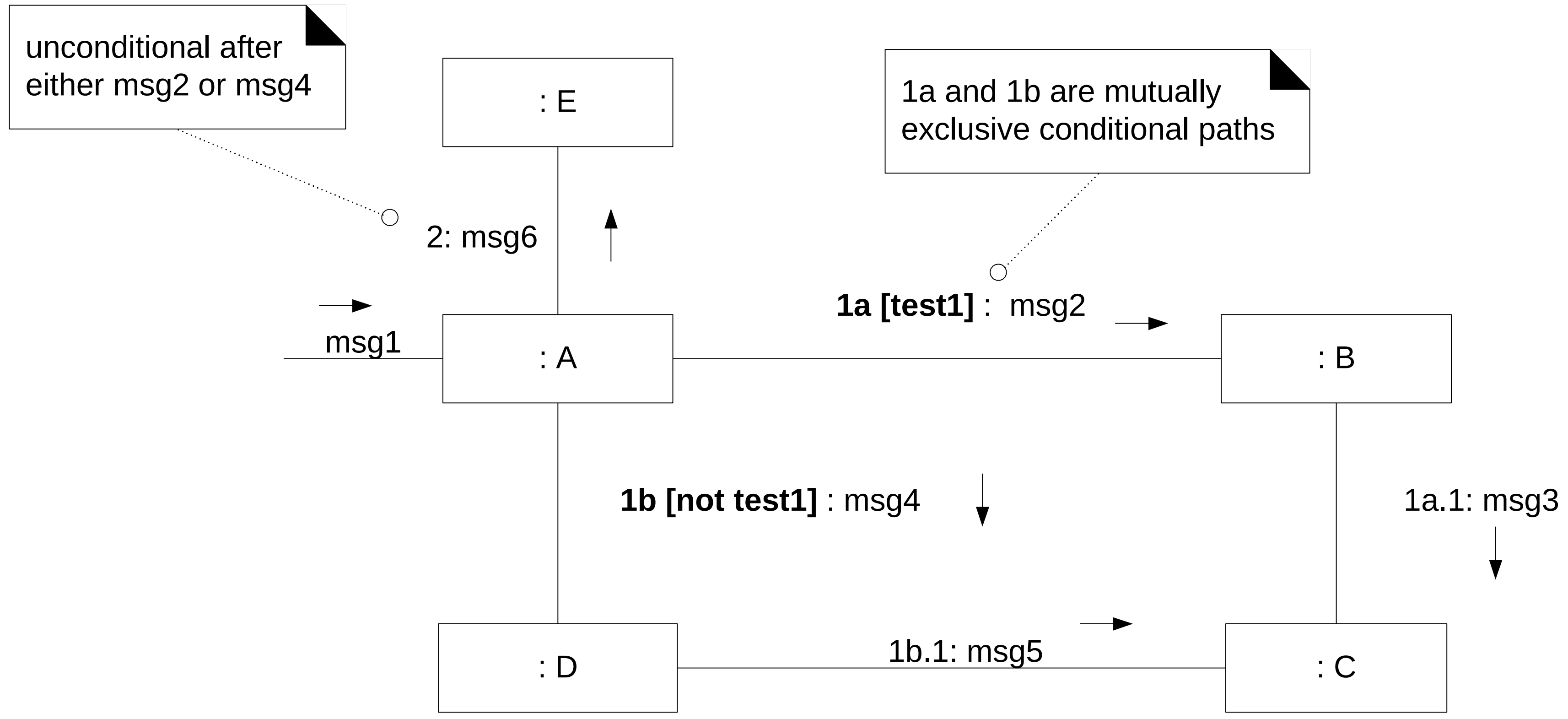
CD: Complex Sequencing



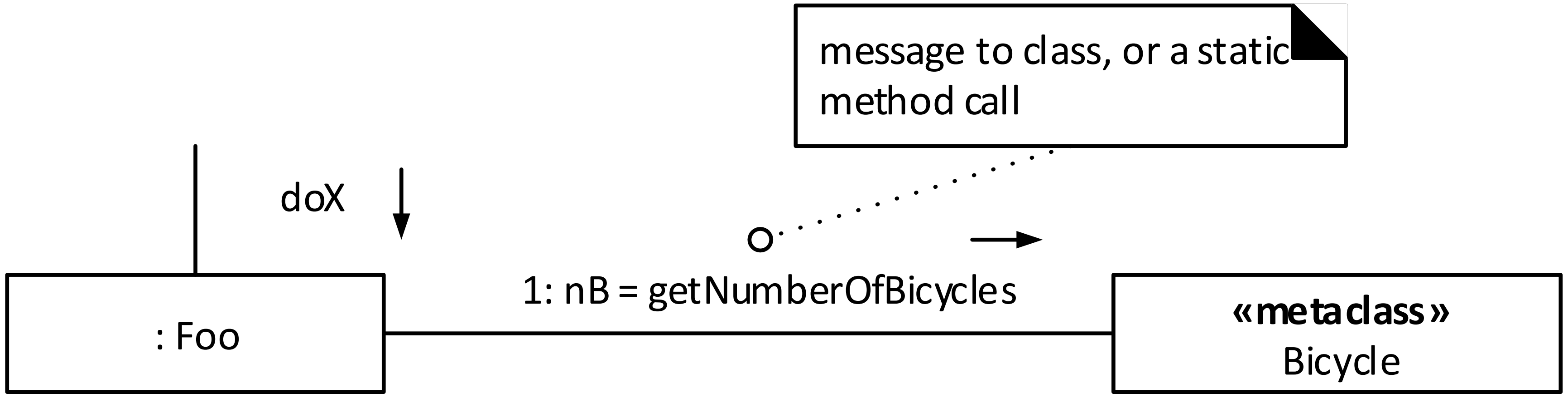
CD: Conditional Messages


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

CD: Mutually Exclusive Messages

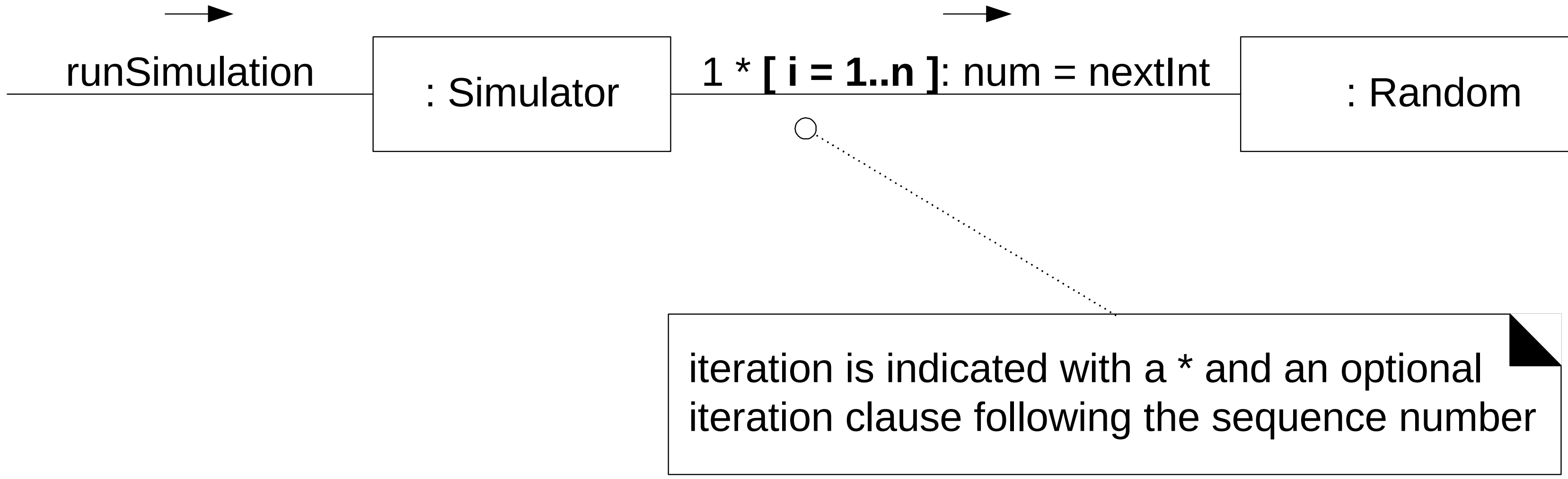

unconditional after either msg2 or msg4
1a and 1b are mutually exclusive conditional paths
: E
: A
: B
: C
: D
msg1
2: msg6
1a [test1] : msg2
1b [not test1] : msg4
1a.1: msg3
1b.1: msg5

CD: Static (or Class) Messages

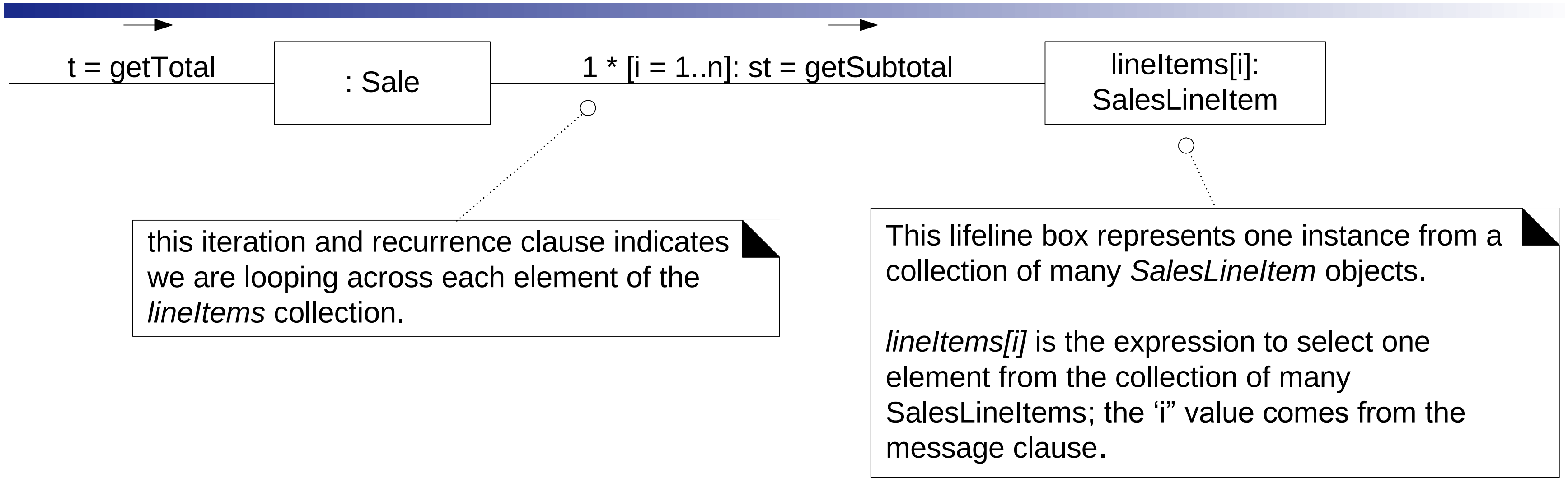

message to class, or a static method call
doX
1: nB = getNumberOfBicycles
: Foo
«metaclass»
Bicycle

```
public class Bicycle {

  private static int numberOfBicycles = 0;

  public static int getNumberOfBicycles() {
    return numberOfBicycles;
// …
}
```

CD: Iteration


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

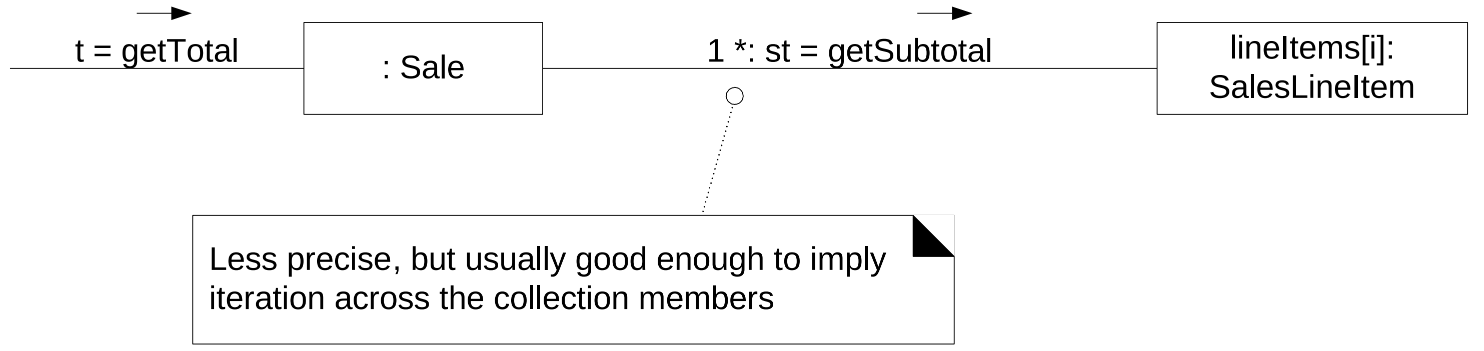
CD: Iteration over a Collection



this iteration and recurrence clause indicates we are looping across each element of the *lineItems* collection.

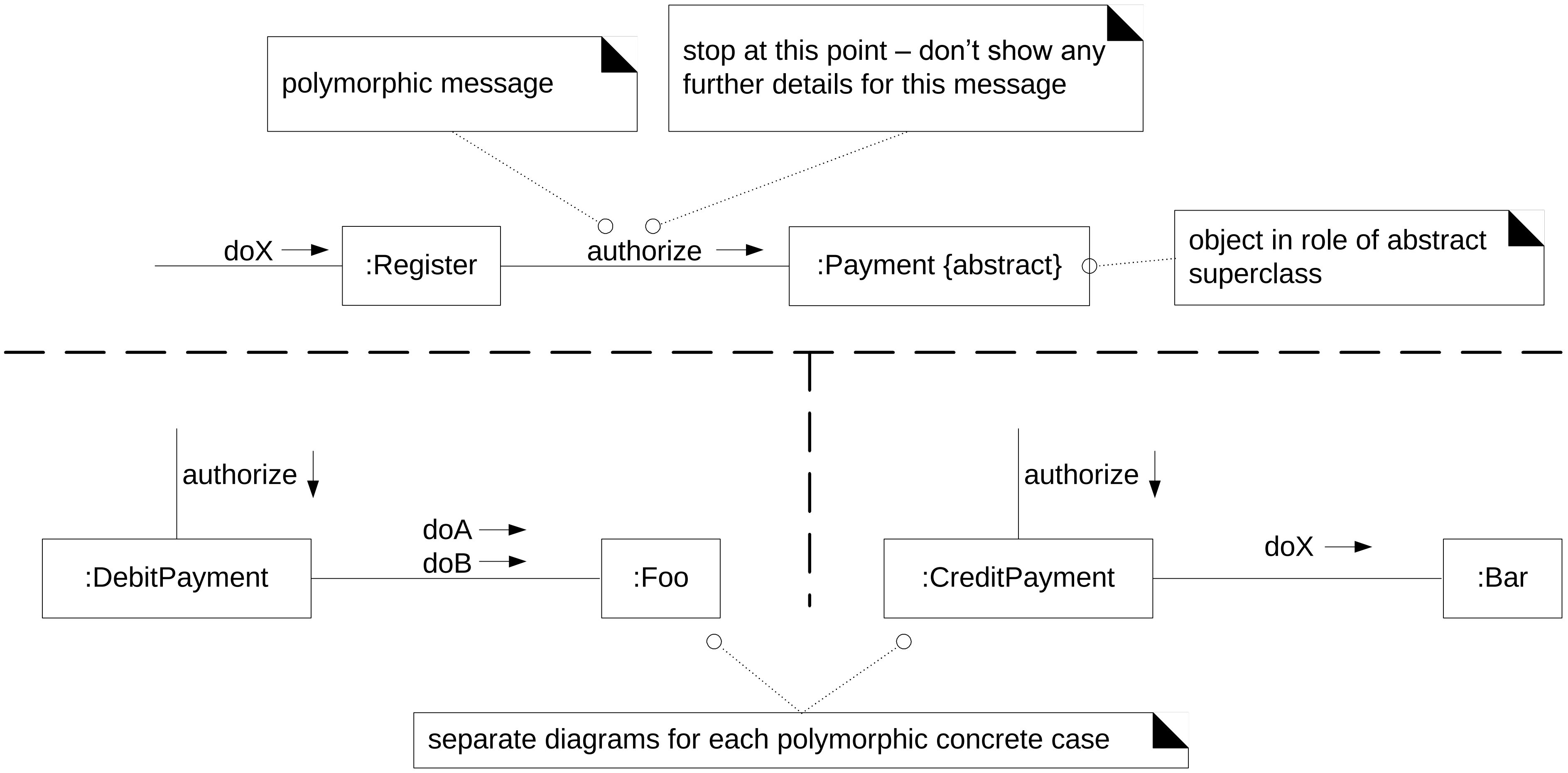
This lifeline box represents one instance from a collection of many *SalesLineItem* objects.

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

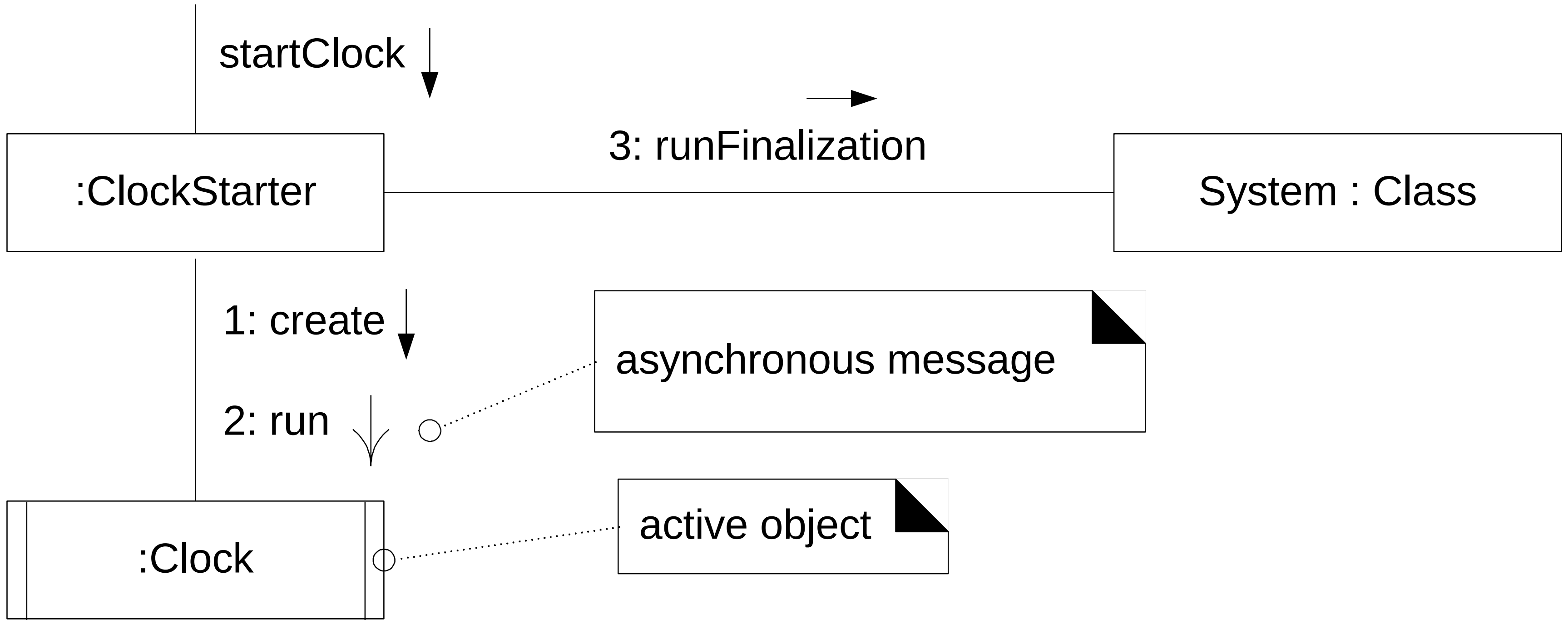


Less precise, but usually good enough to imply iteration across the collection members

CD: Polymorphic Messages/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

CD: Asynchronous/Synchronous Calls


startClock
3: runFinalization
:ClockStarter
System : Class
1: create
2: run
asynchronous message
active object
:Clock