The University of Iowa

CS:2820 (22C:22)

Object-Oriented Software Development

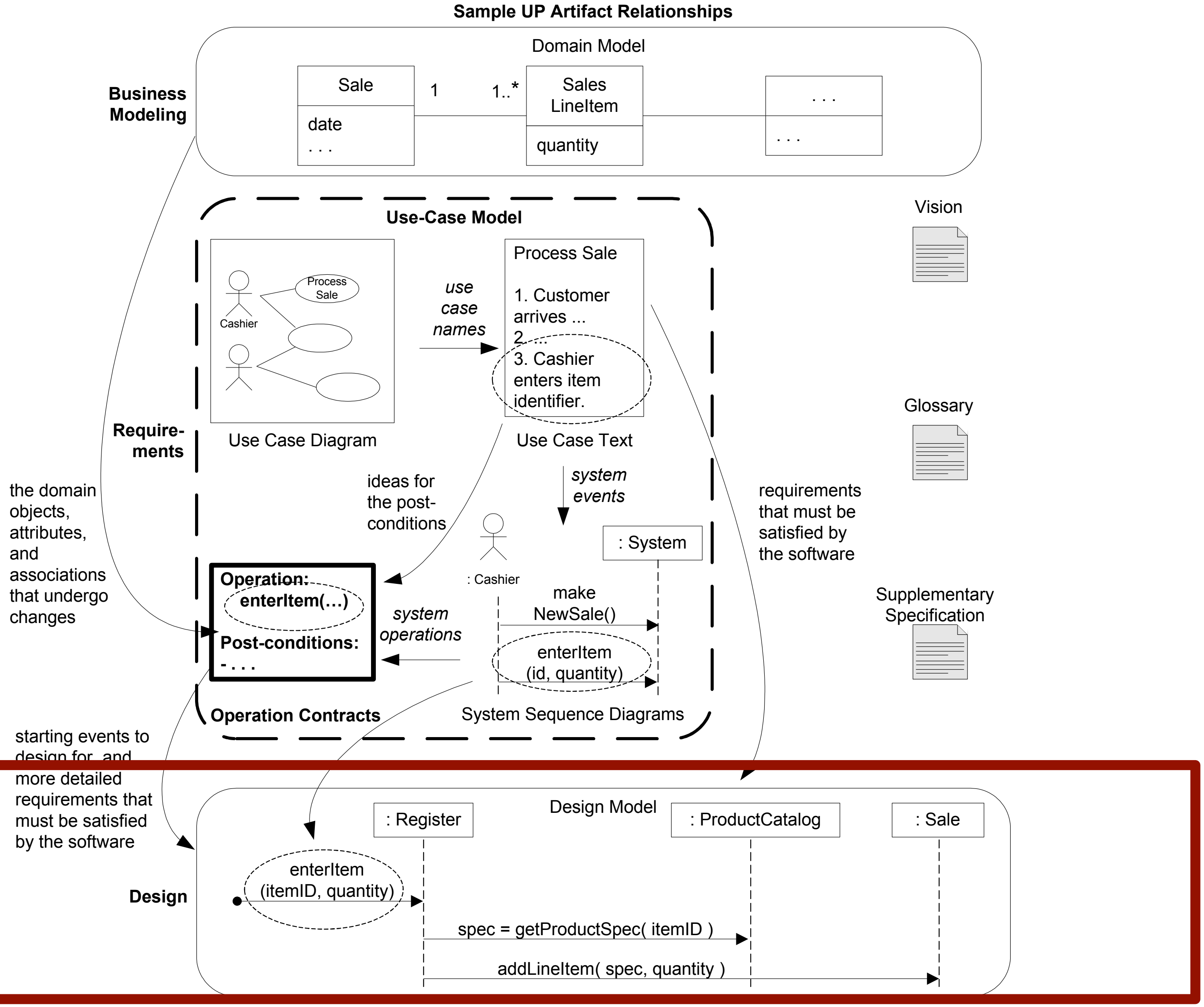
Spring 2015

Interaction Diagrams

(Chapter 15)

by
Mauricio Monsalve

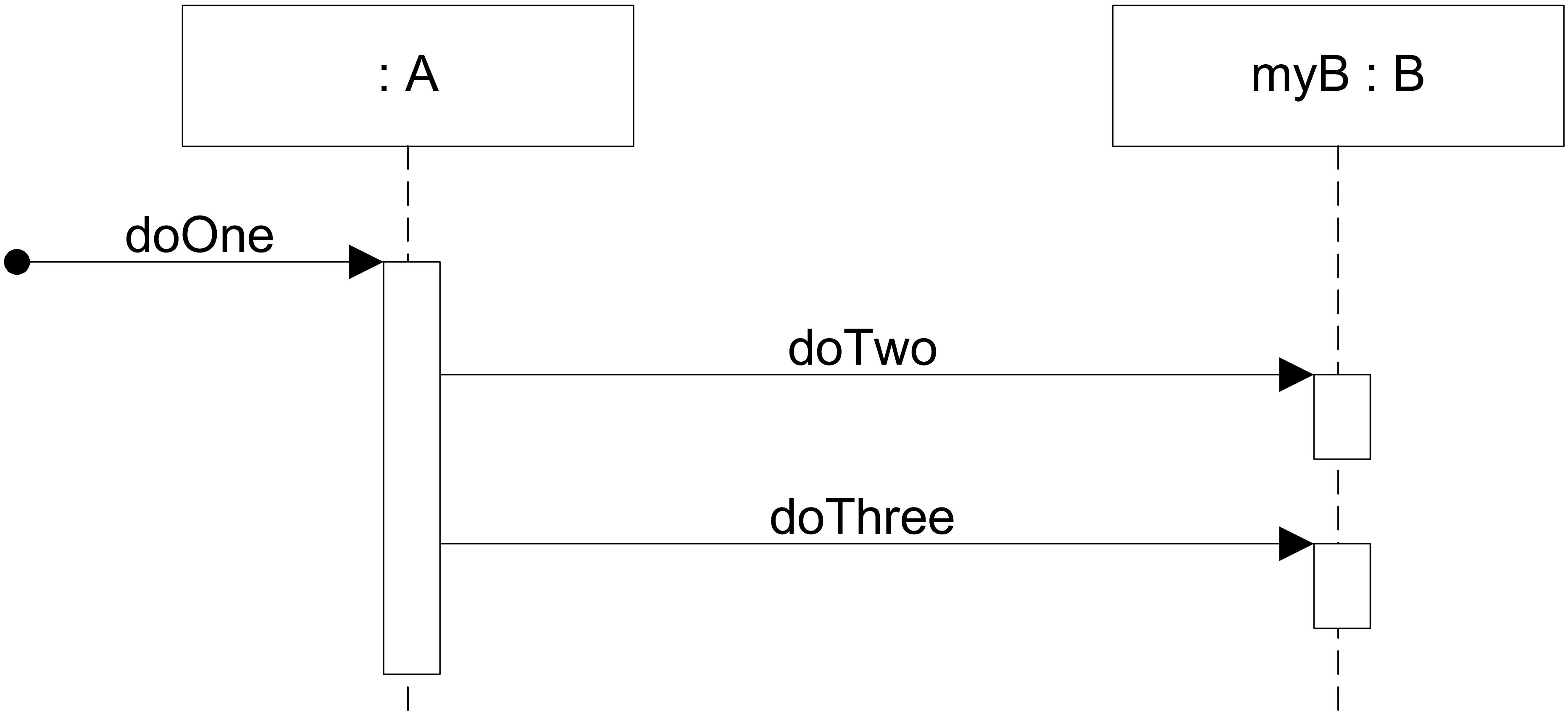
Design Road


Sample UP Artifact Relationships
Domain Model
Business Modeling
Sale
date
. . .
1
1..*
Sales LineItem
quantity
. . .
. . .
Use-Case Model
Vision
Process Sale
Cashier
use case names
Process Sale
1. Customer arrives ...
2. ...
3. Cashier enters item identifier.
Require-ments
Use Case Diagram
Use Case Text
Glossary
the domain objects, attributes, and associations that undergo changes
ideas for the post-conditions
system events
requirements that must be satisfied by the software
: System
: Cashier
Operation: enterItem(…)
Post-conditions: - . . .
system operations
make NewSale()
enterItem (id, quantity)
Supplementary Specification
Operation Contracts
System Sequence Diagrams
starting events to design for, and more detailed requirements that must be satisfied by the software
Design Model
: Register
: ProductCatalog
: Sale
Design
enterItem (itemID, quantity)
spec = getProductSpec(itemID)
addLineItem(spec, quantity)

Interaction Diagrams

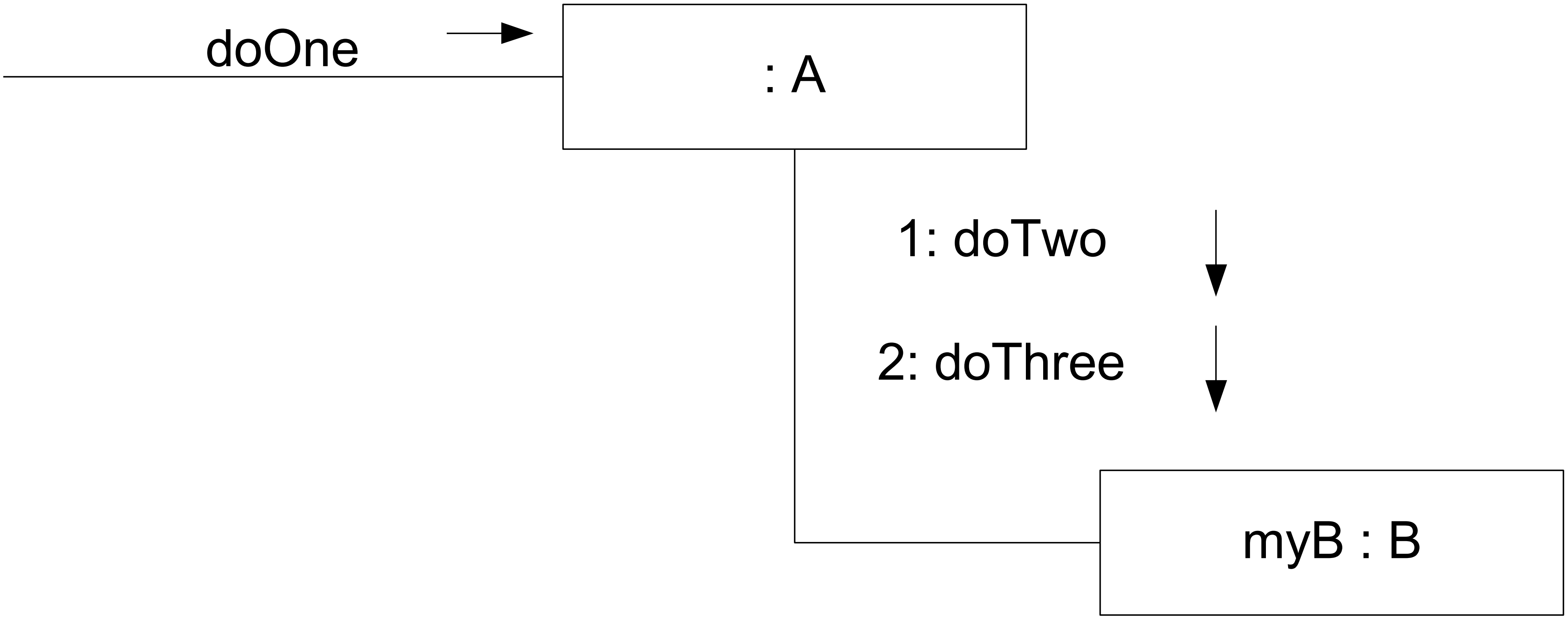
- UML interaction diagrams represent interaction (communication, collaboration) between objects/classes
- For dynamic object modeling
- UML interaction diagrams consist of
  - Sequence diagrams
  - Communication diagrams

Sequence Diagram



We have used a simplified version of these for System Sequence Diagrams

Communication Diagram



Steps are enumerated and placed in lines with arrows

The diagrams compared

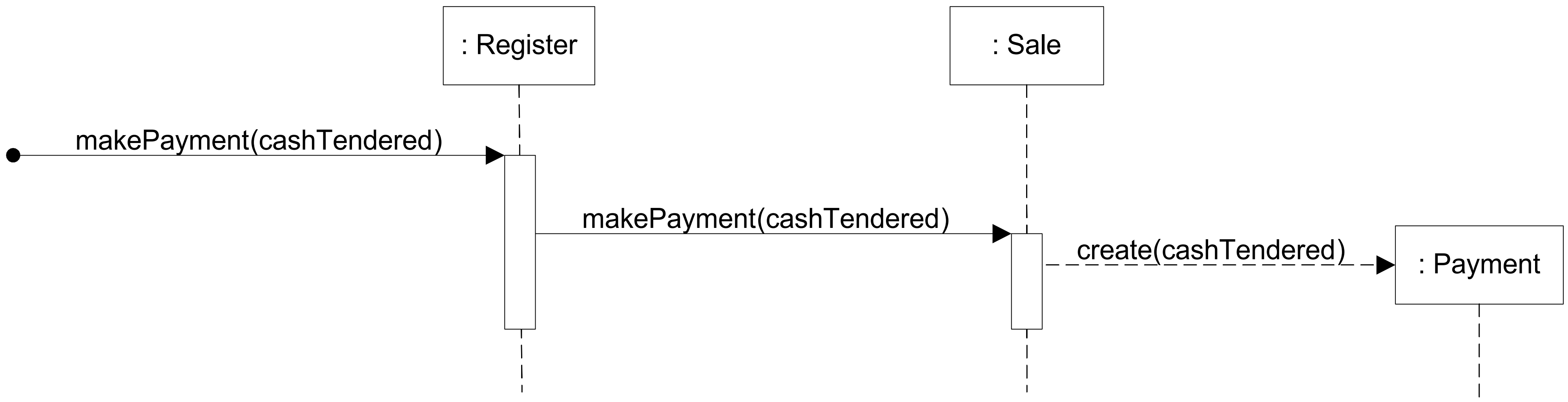
Sequence diagram

- 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

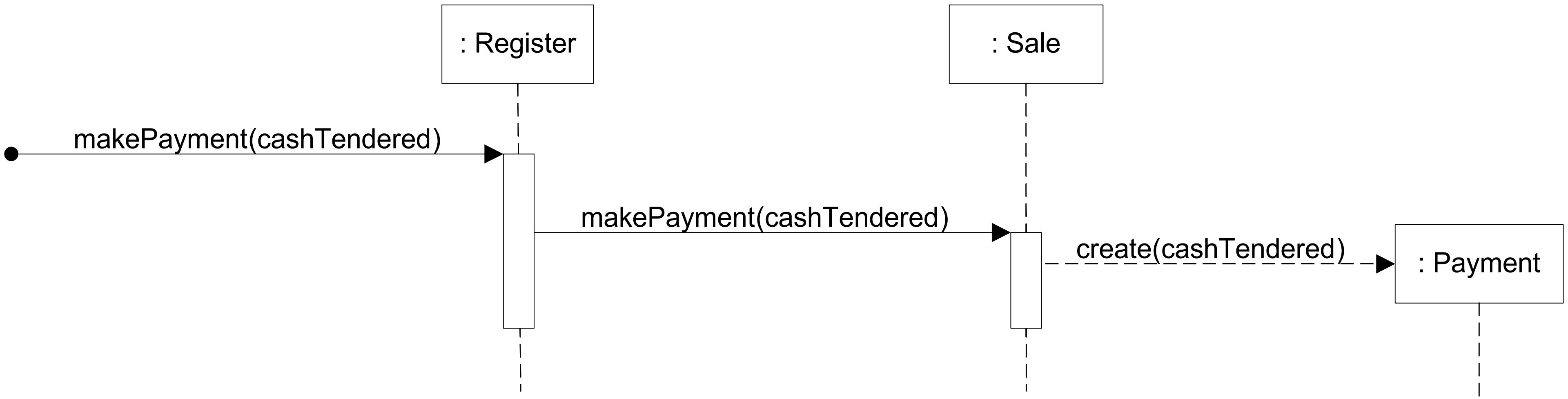
Communication diagram

- space economical; flexibility to add new objects in two dimensions
- more difficult to see sequence of messages
- fewer notation options

Exercise


: Register
: Sale
makePayment(cashTendered)
makePayment(cashTendered)
create(cashTendered)
: Payment

Exercise



: Register

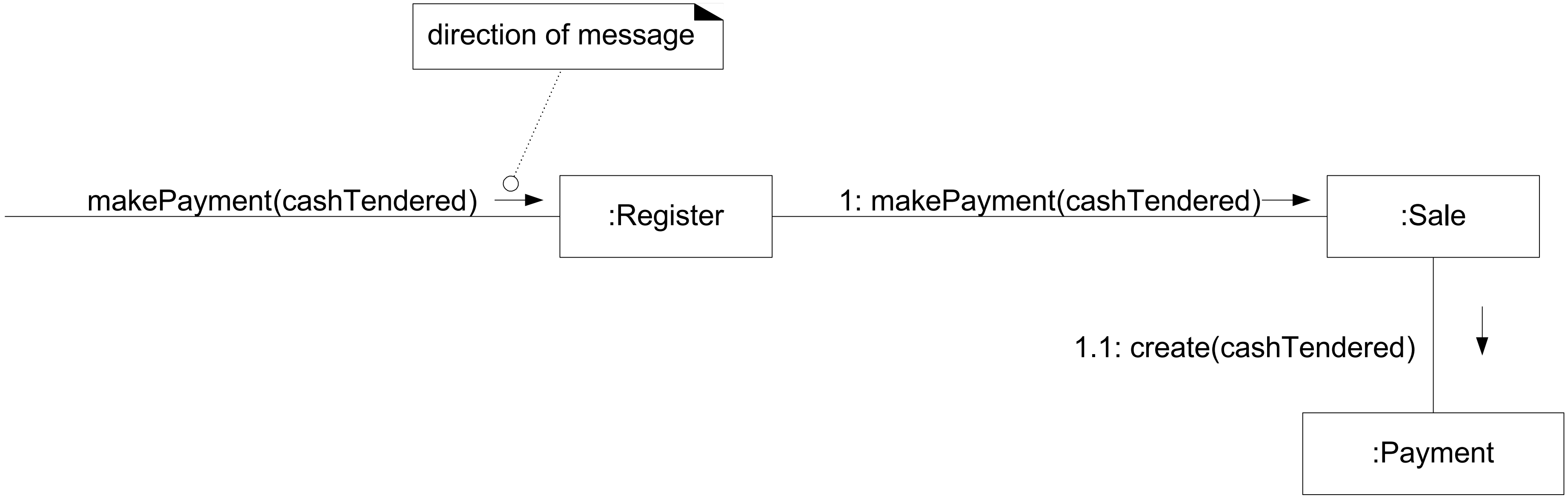
: Sale

makePayment(cashTendered)

makePayment(cashTendered)

create(cashTendered)

: Payment



direction of message

makePayment(cashTendered)

:Register

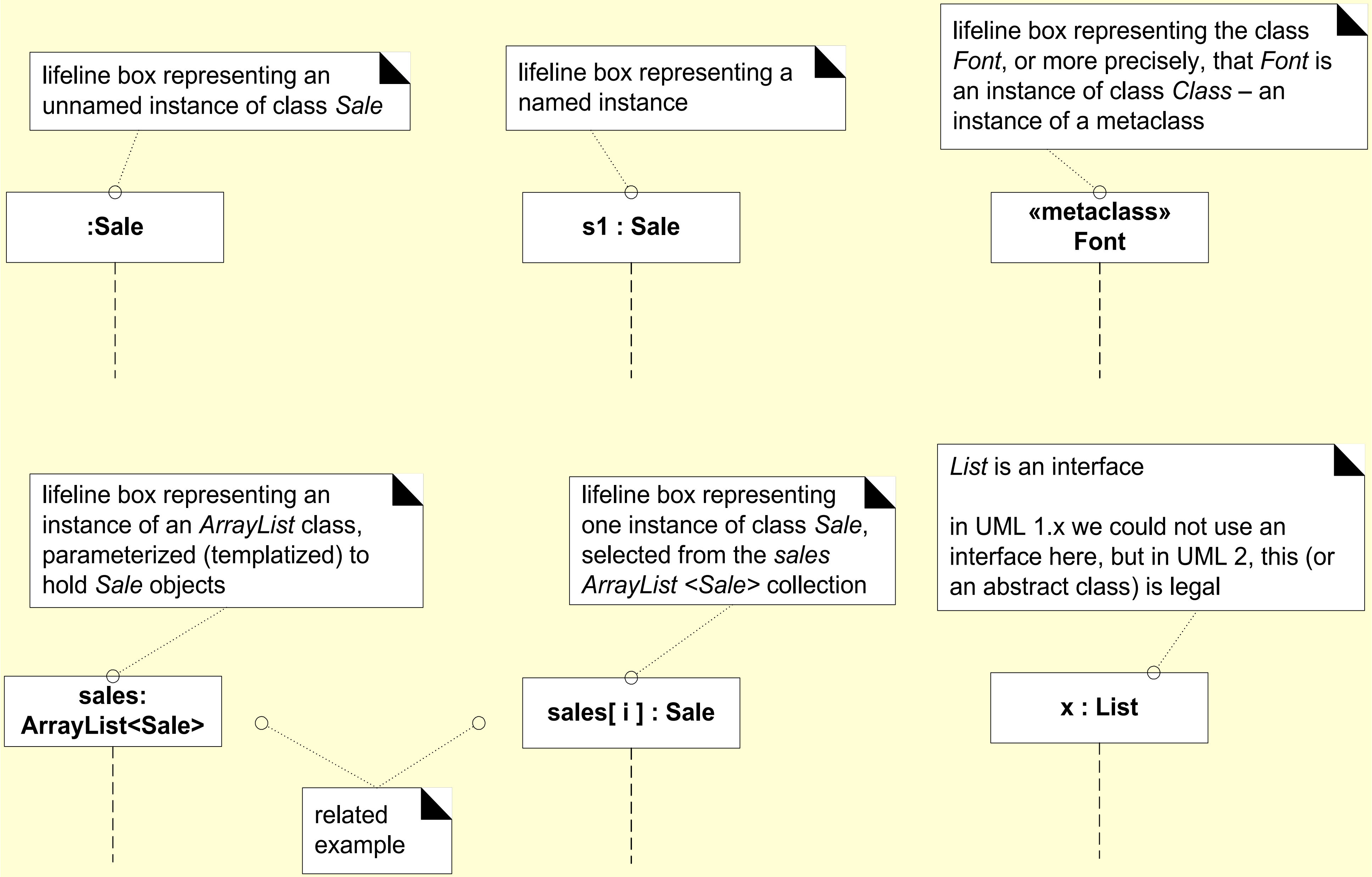
1: makePayment(cashTendered)

:Sale

1.1: create(cashTendered)

:Payment

Drawing Sequence Diagrams



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

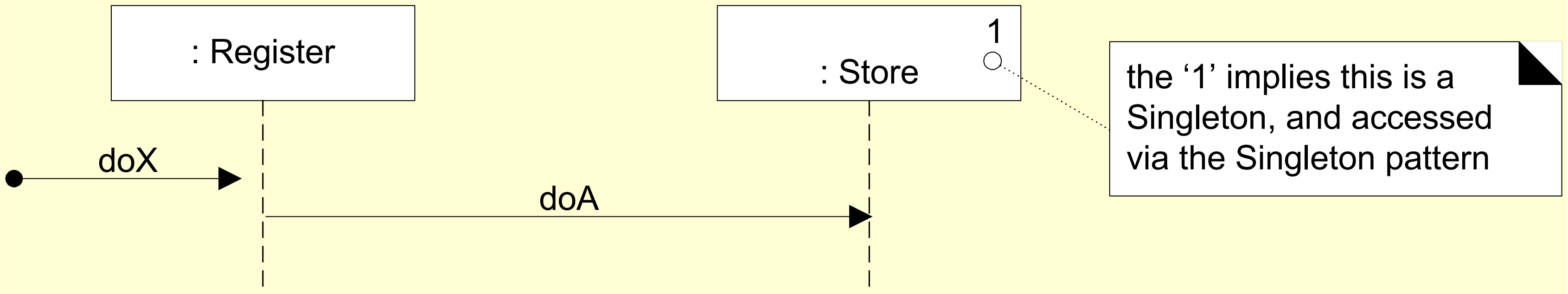
lifeline box representing one instance of class *Sale*, selected from the *sales* *ArrayList <Sale>* collection

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

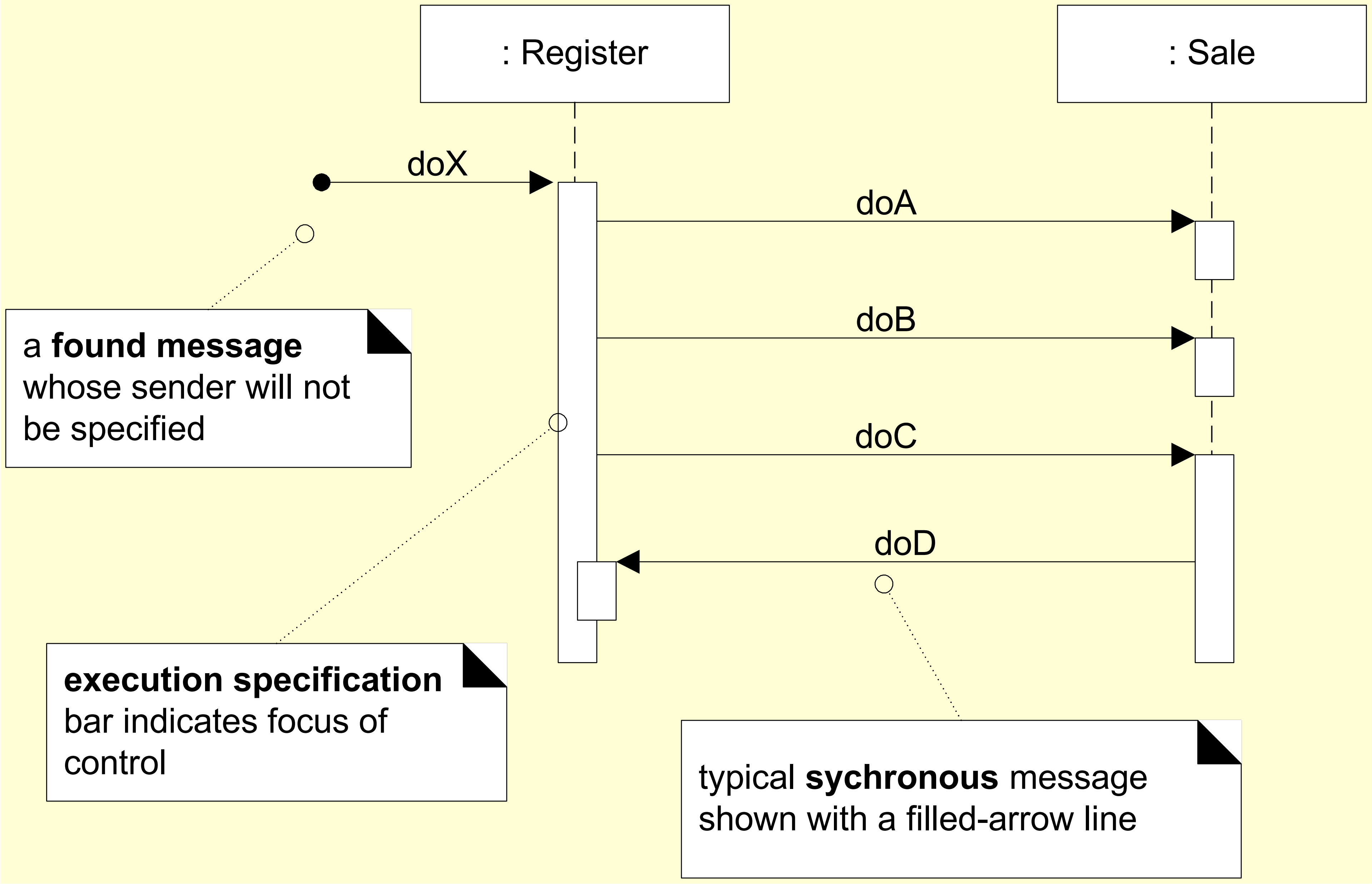
Drawing Sequence Diagrams


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

- In the case of singleton objects/classes, we put a “1” on their boxes

- Singleton classes are the ones that only have one instance
  - Cf. Scala: singleton defined with “object”, not “class”

Drawing Sequence Diagrams

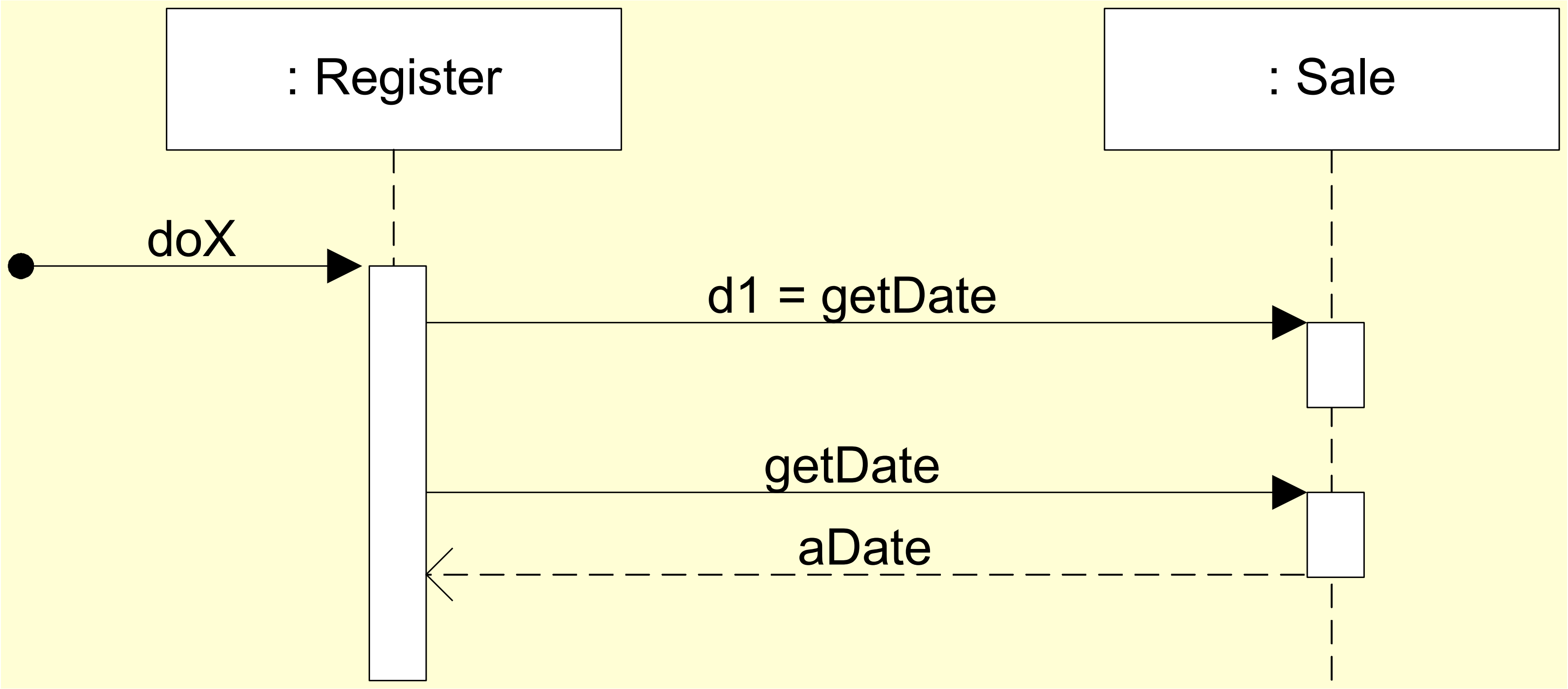


a **found message** whose sender will not be specified

execution specification bar indicates focus of control

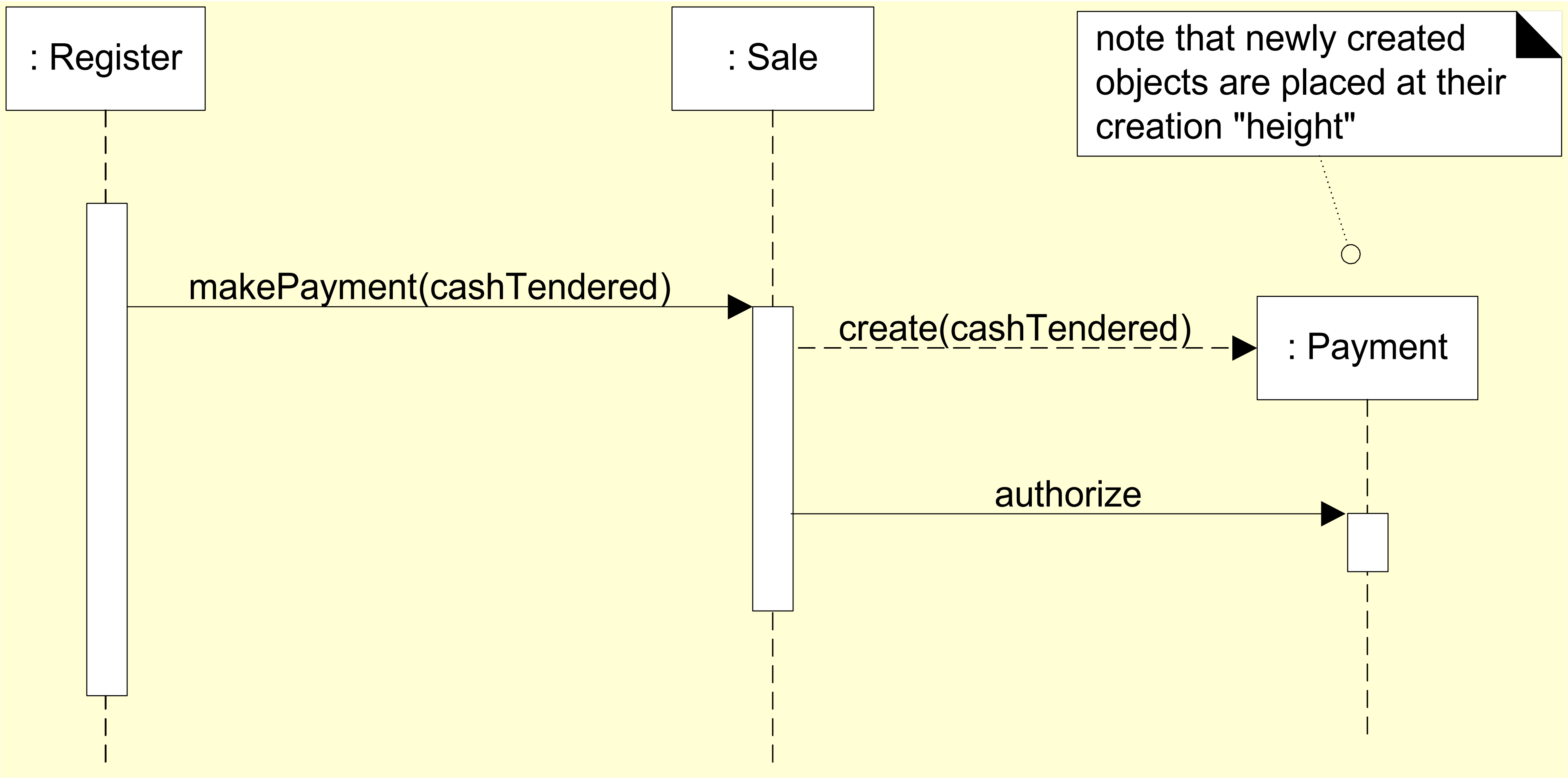
typical **sychronous** message shown with a filled-arrow line

Drawing Sequence Diagrams

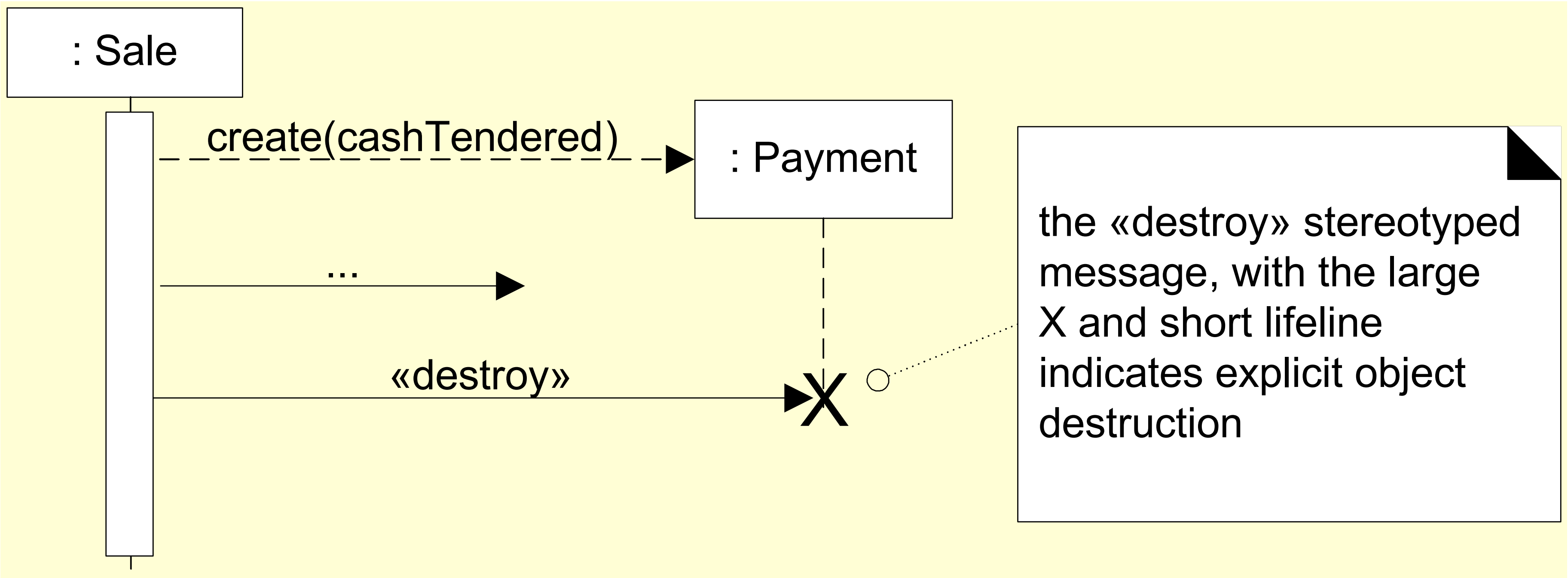


Two ways to specify a return value.
The first one is brief.
The second one allows one to describe the information contained in the returned value.

Drawing Sequence Diagrams


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

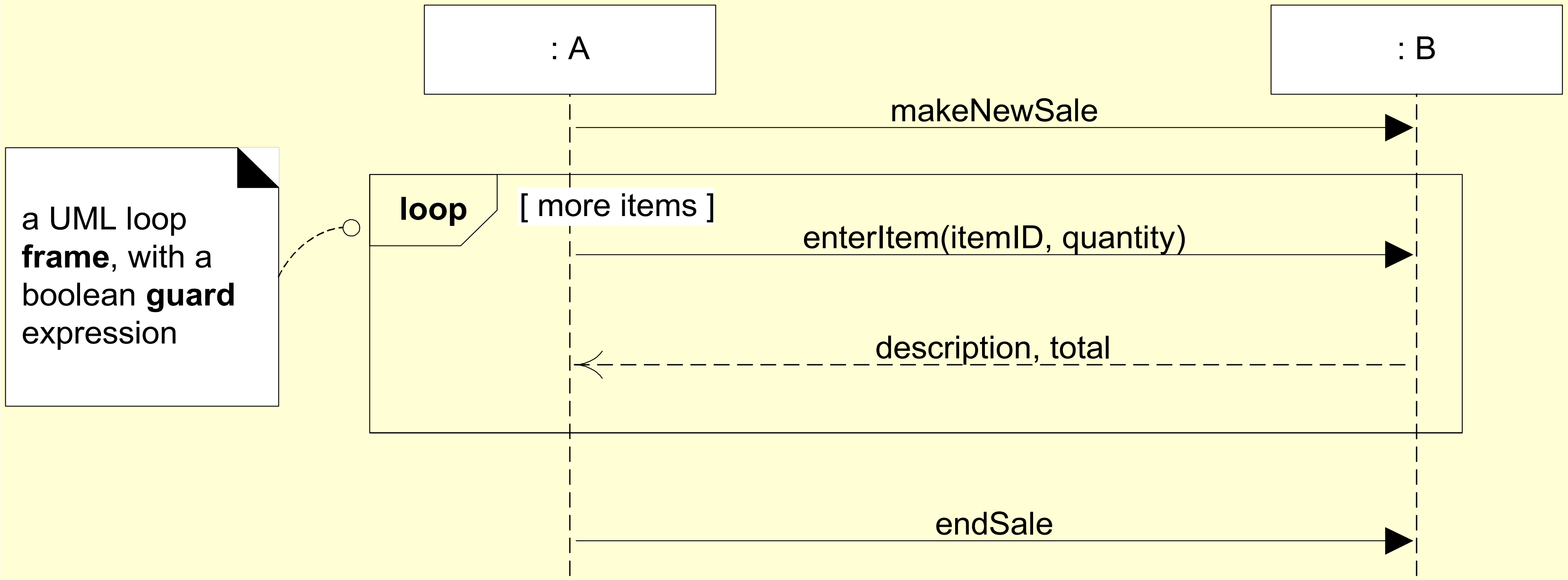
Drawing Sequence Diagrams



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

Vertical "presence" or coverage demonstrates the life-cycle of an object

Drawing Sequence Diagrams


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

Types of frames:

loop—for repeated statements,

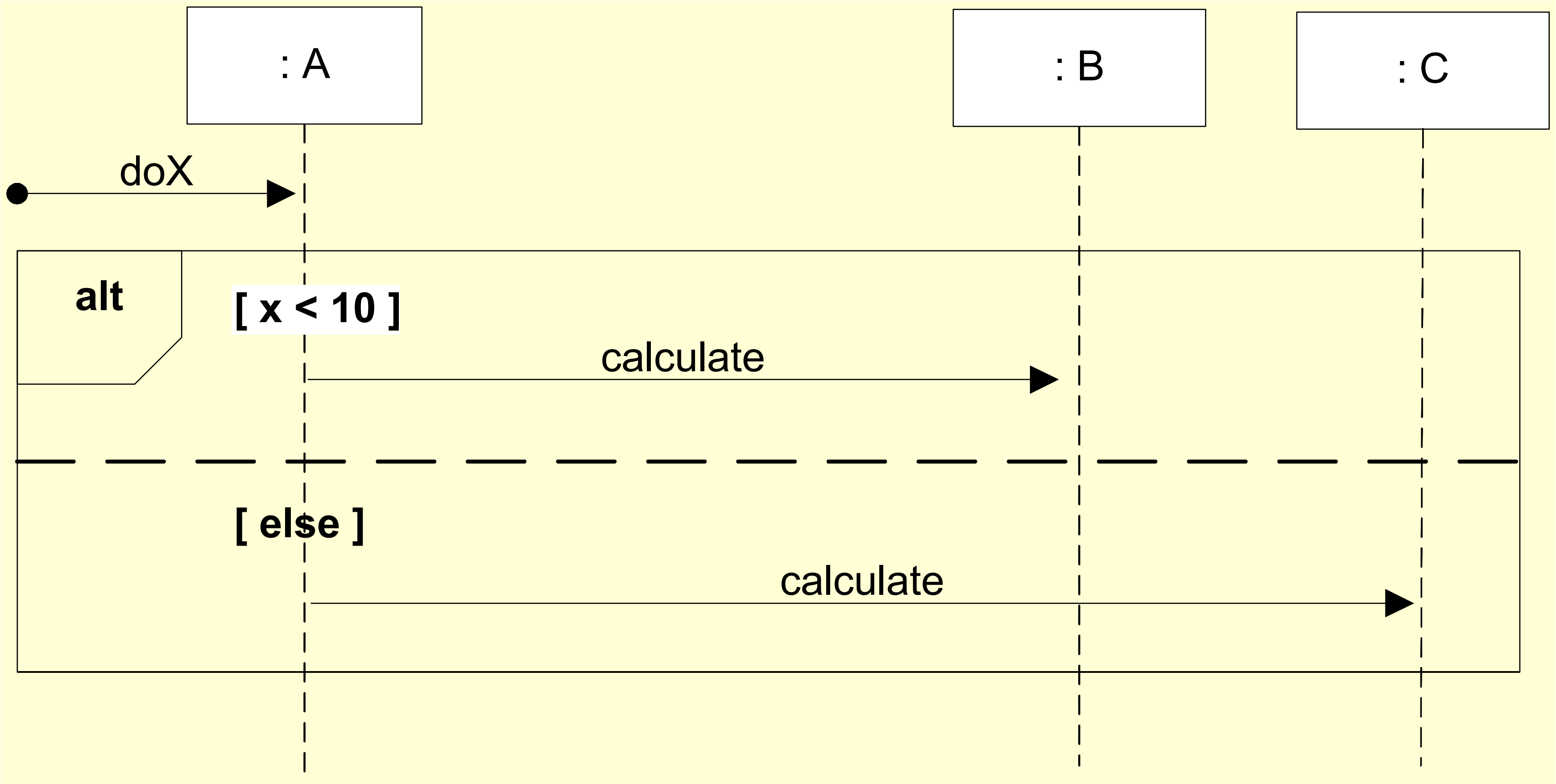
opt—for if-statements without else,

alt—for if-statements with else or else-if,

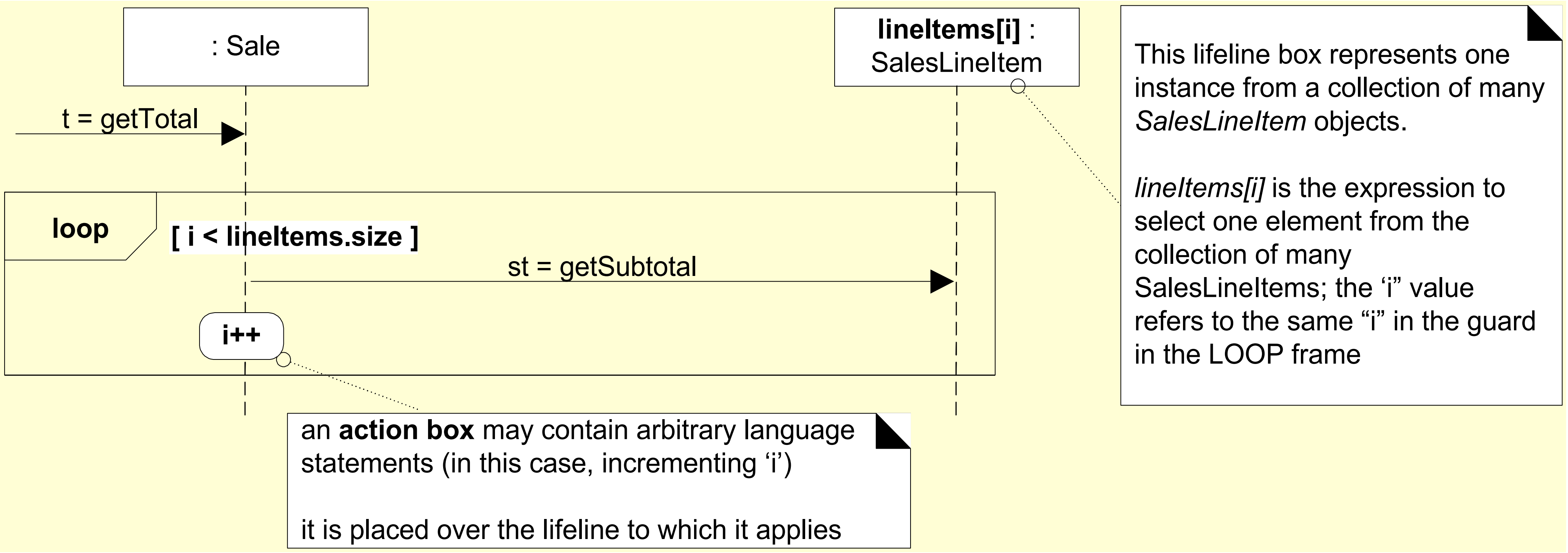
par—for parallel execution,

region—for critical region (concurrency).

Drawing Sequence Diagrams



Drawing Sequence Diagrams



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

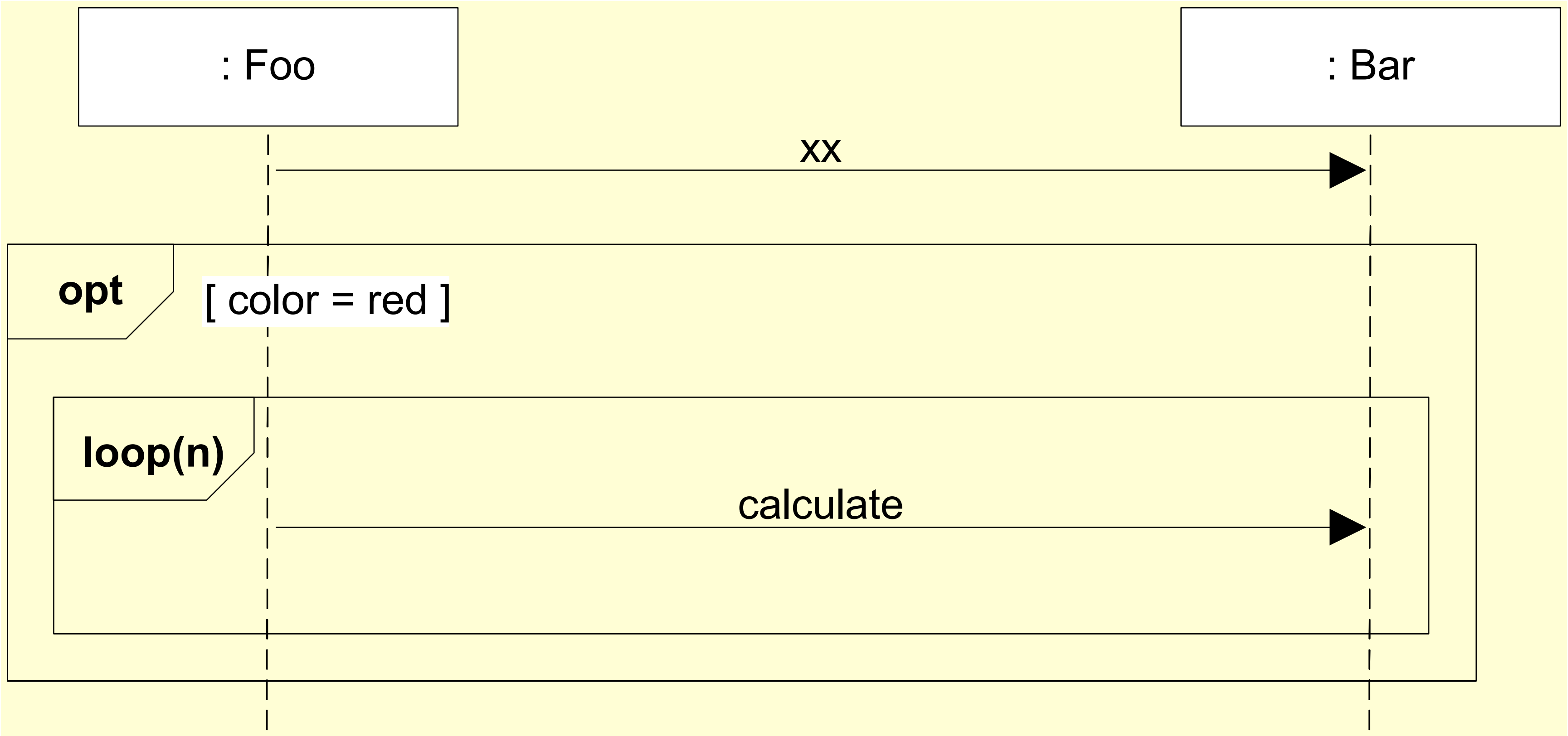
lineItems[i] is the expression to select one element from the collection of many SalesLineItems; the 'i" value refers to the same "i" in the guard in the LOOP frame

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

it is placed over the lifeline to which it applies

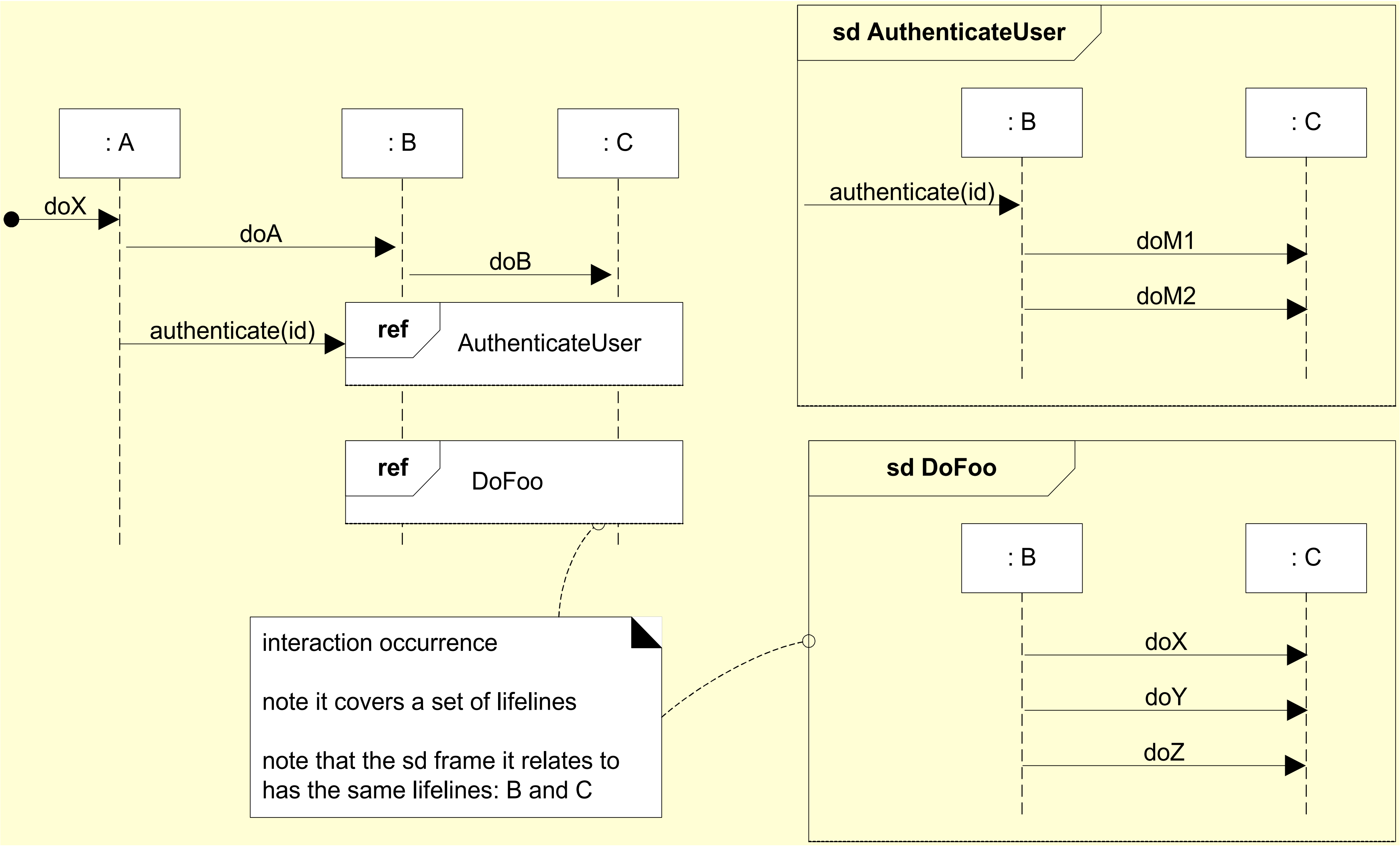
Vertical "presence" or coverage demonstrates the life-cycle of an object

Drawing Sequence Diagrams



Nesting of frames

Drawing Sequence Diagrams

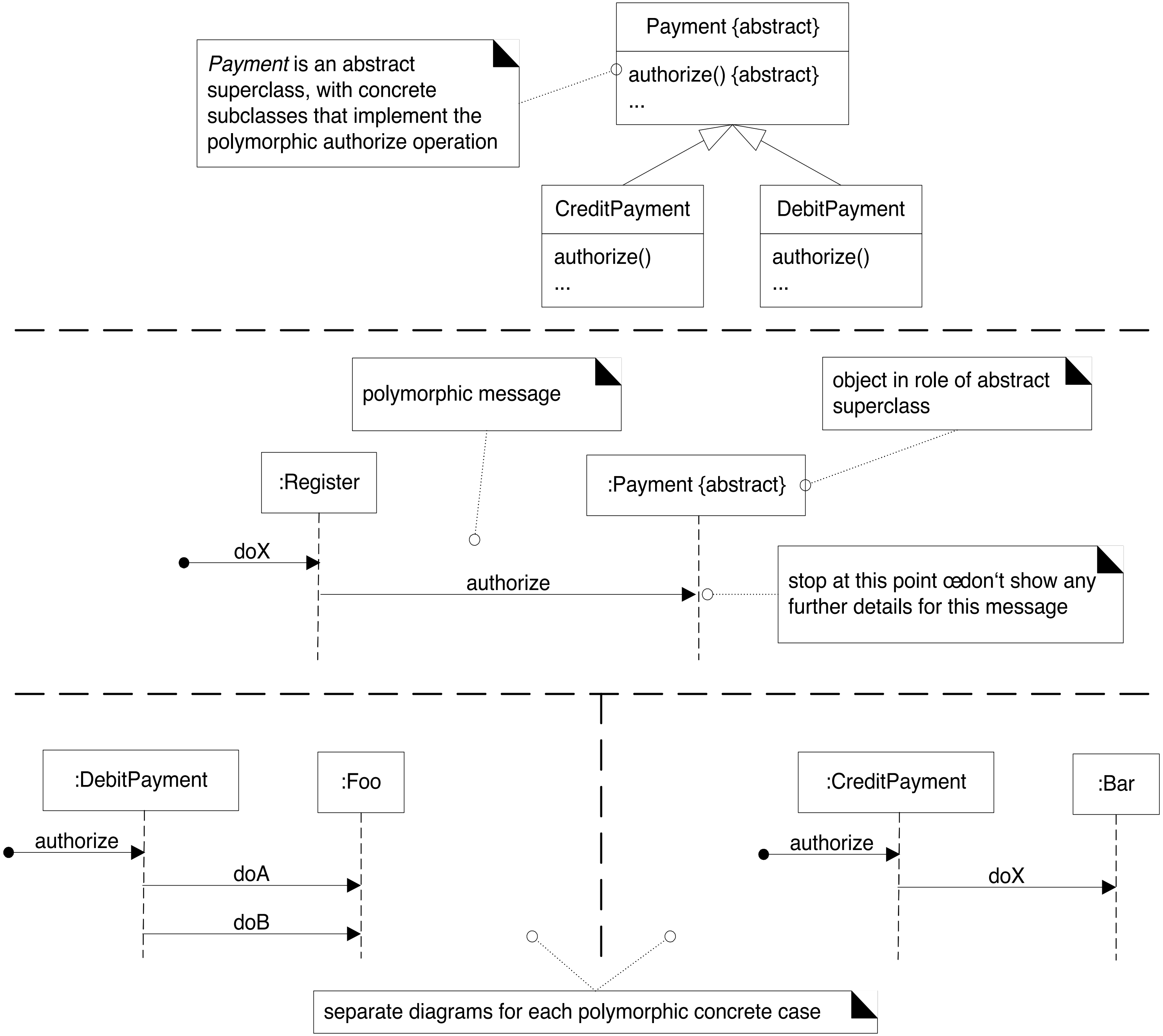


interaction occurrence

note it covers a set of lifelines

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

Polymorphism



Payment is an abstract superclass, with concrete subclasses that implement the polymorphic authorize operation

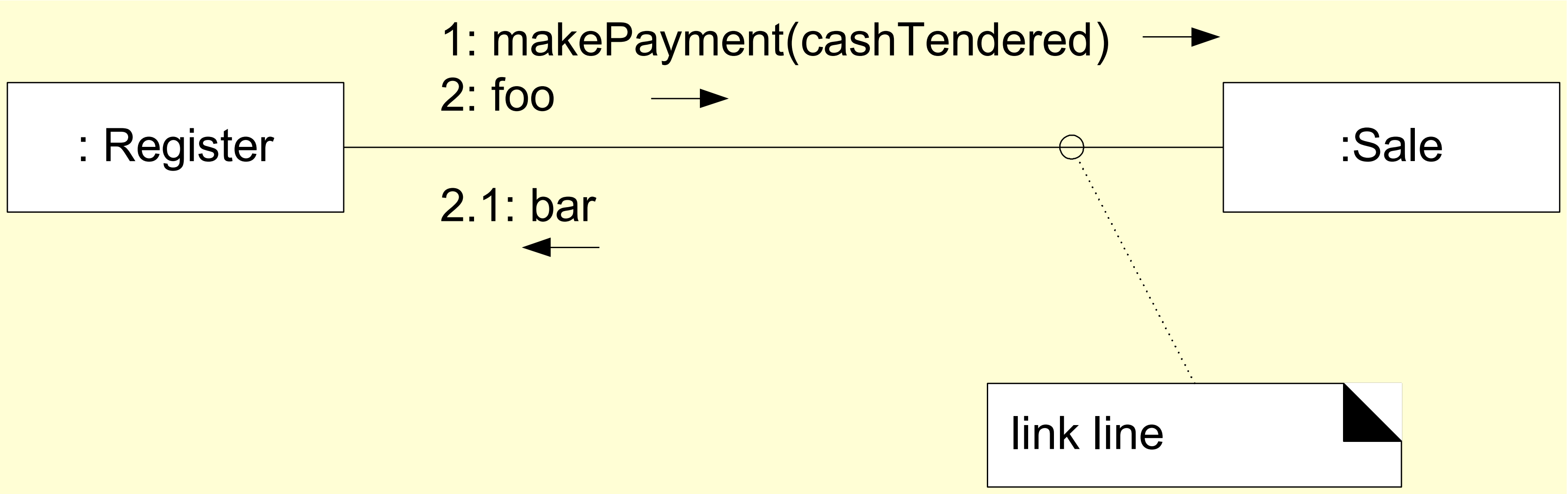
polymorphic message

object in role of abstract superclass

stop at this point œdon‘t show any further details for this message

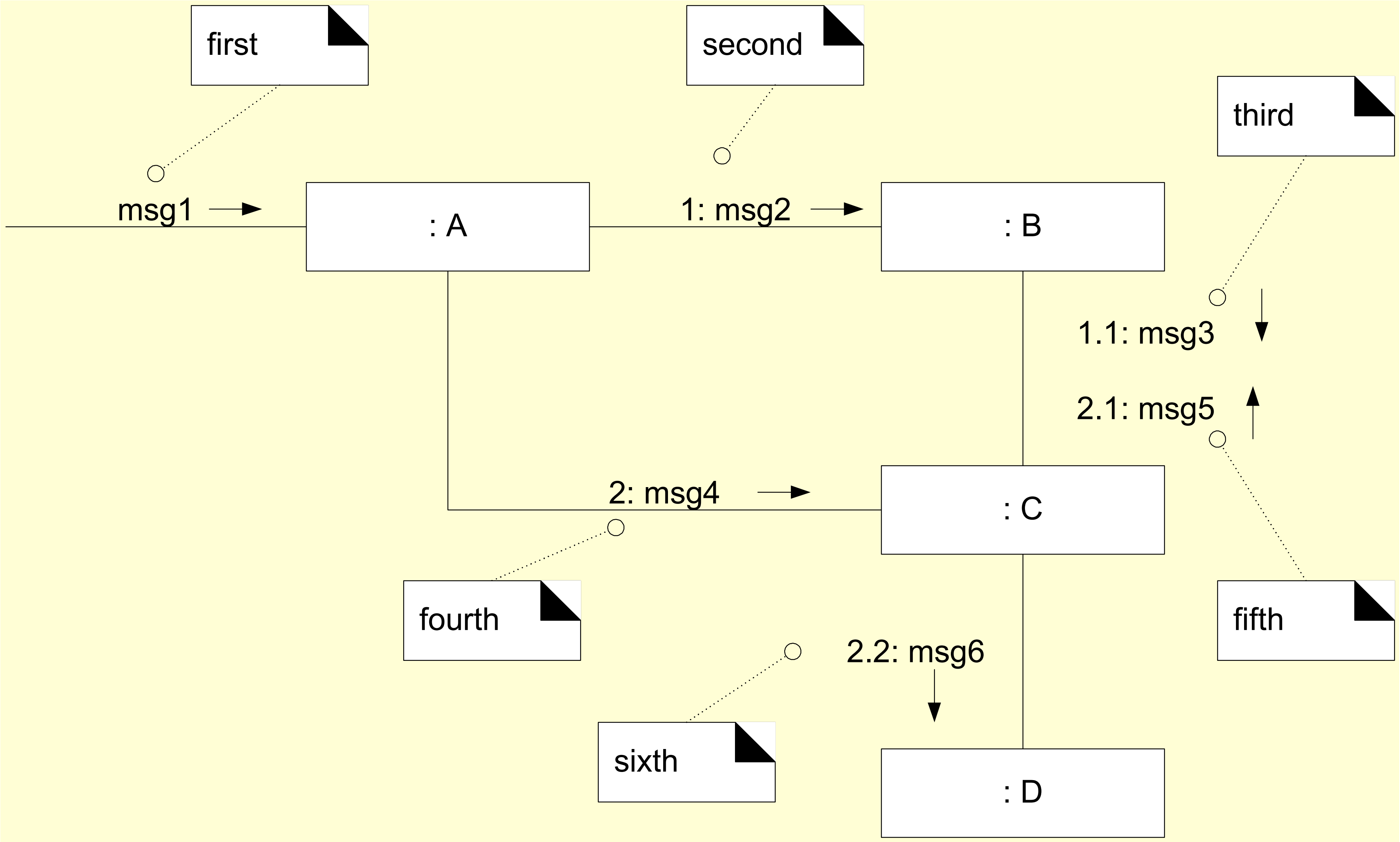
separate diagrams for each polymorphic concrete case

Communication Diagrams

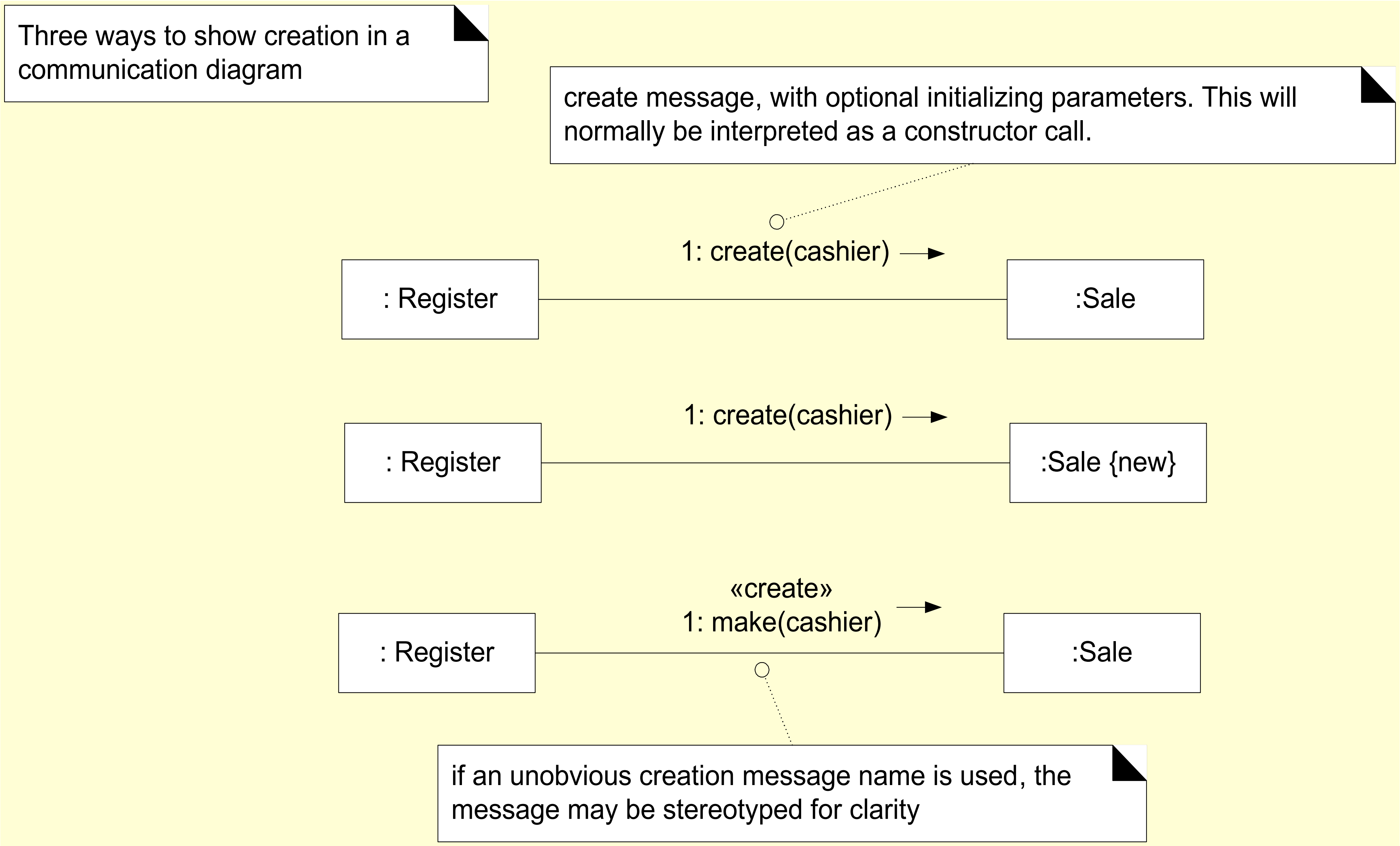


- Numbering follows legalistic ordering
- 1 < 2 < 2.1 < 3 < ...

Communication Diagrams



Communication Diagrams



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}

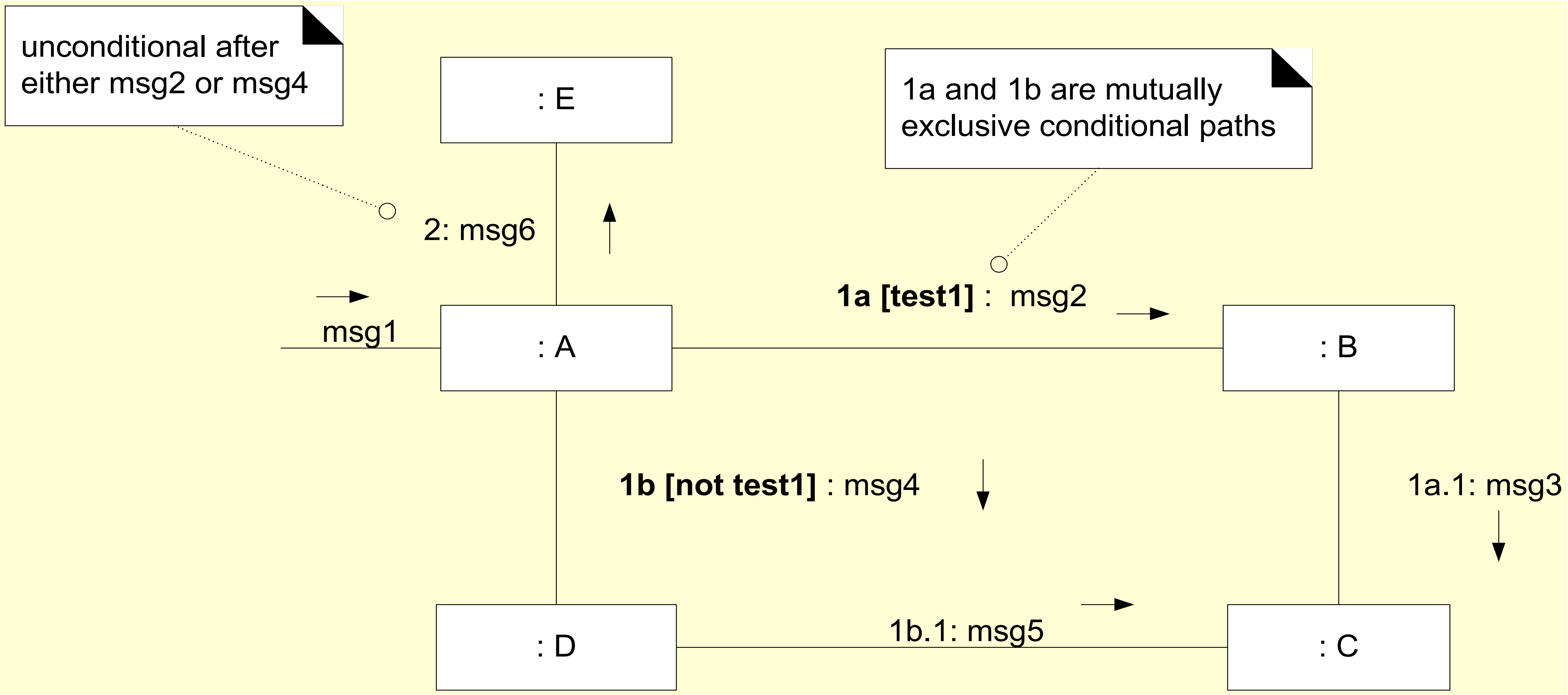
«create»
1: make(cashier) →

: Register

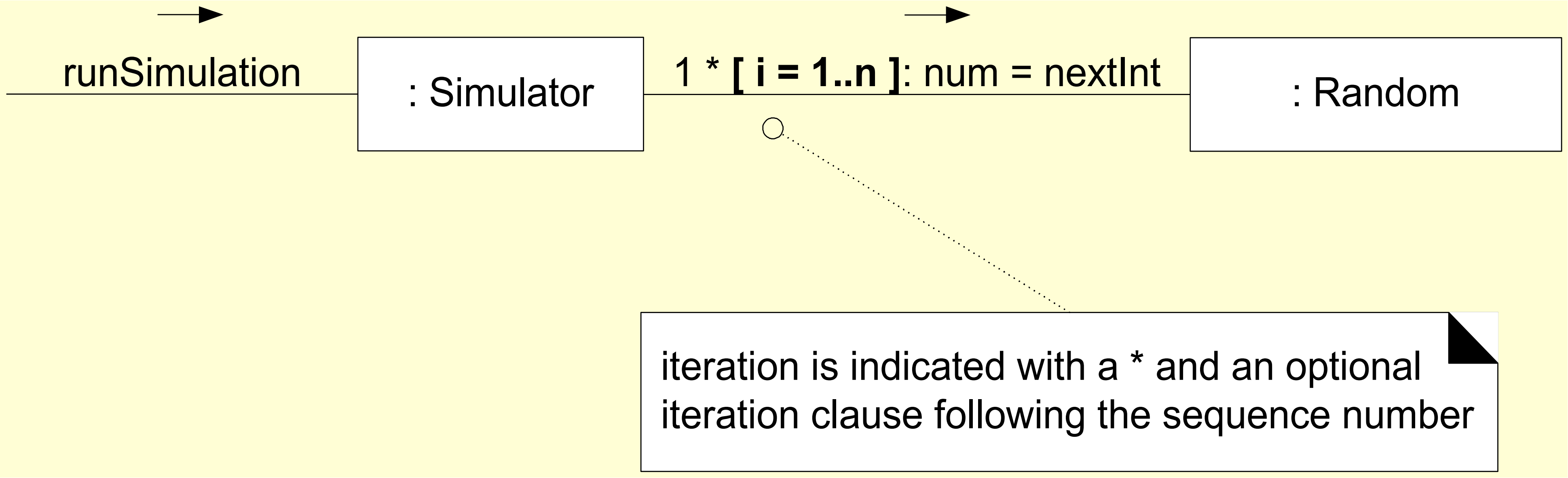
:Sale

if an unobvious creation message name is used, the message may be stereotyped for clarity

Communication Diagrams


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

Communication Diagrams



runSimulation

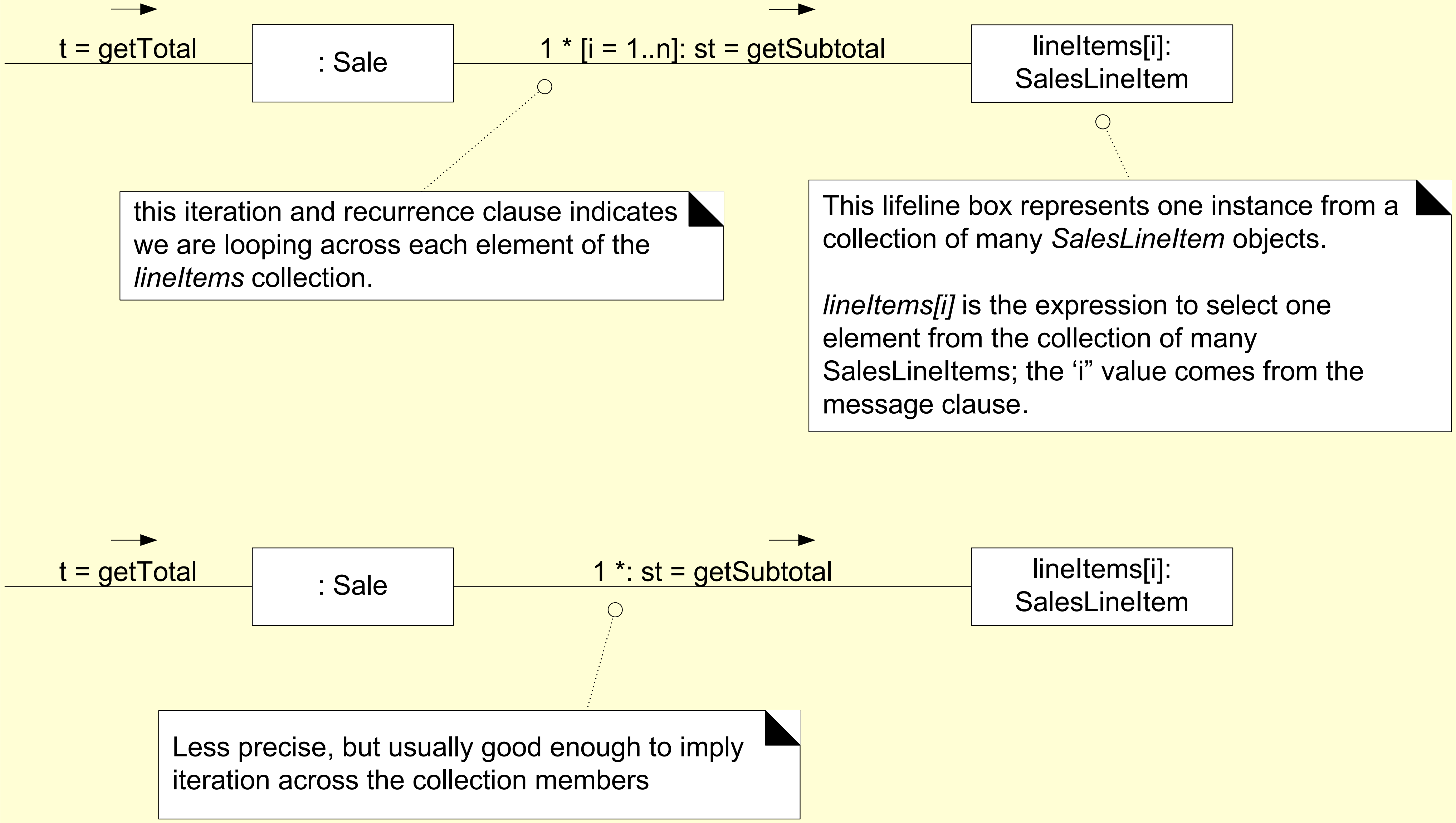
: Simulator

1 * **[i = 1..n]**: num = nextInt

: Random

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

Communication Diagrams



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 ‘i” 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

Concurrency

a stick arrow in UML implies an asynchronous call

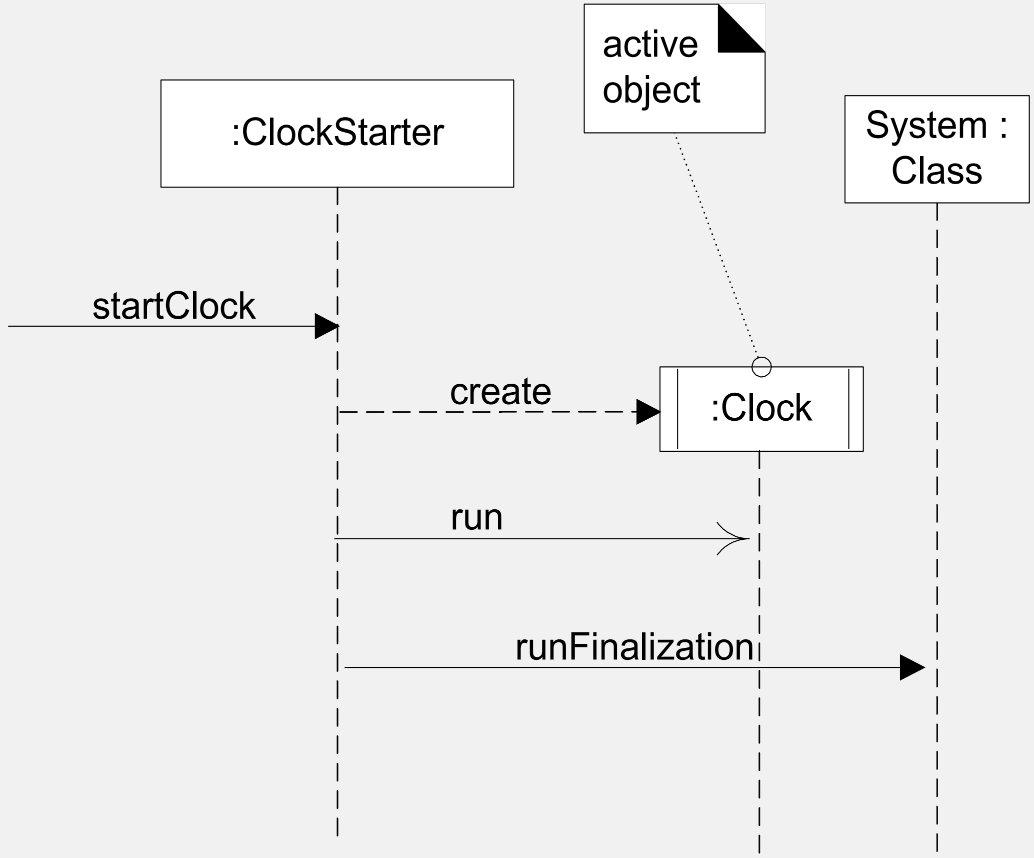
a filled arrow is the more common synchronous call

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

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

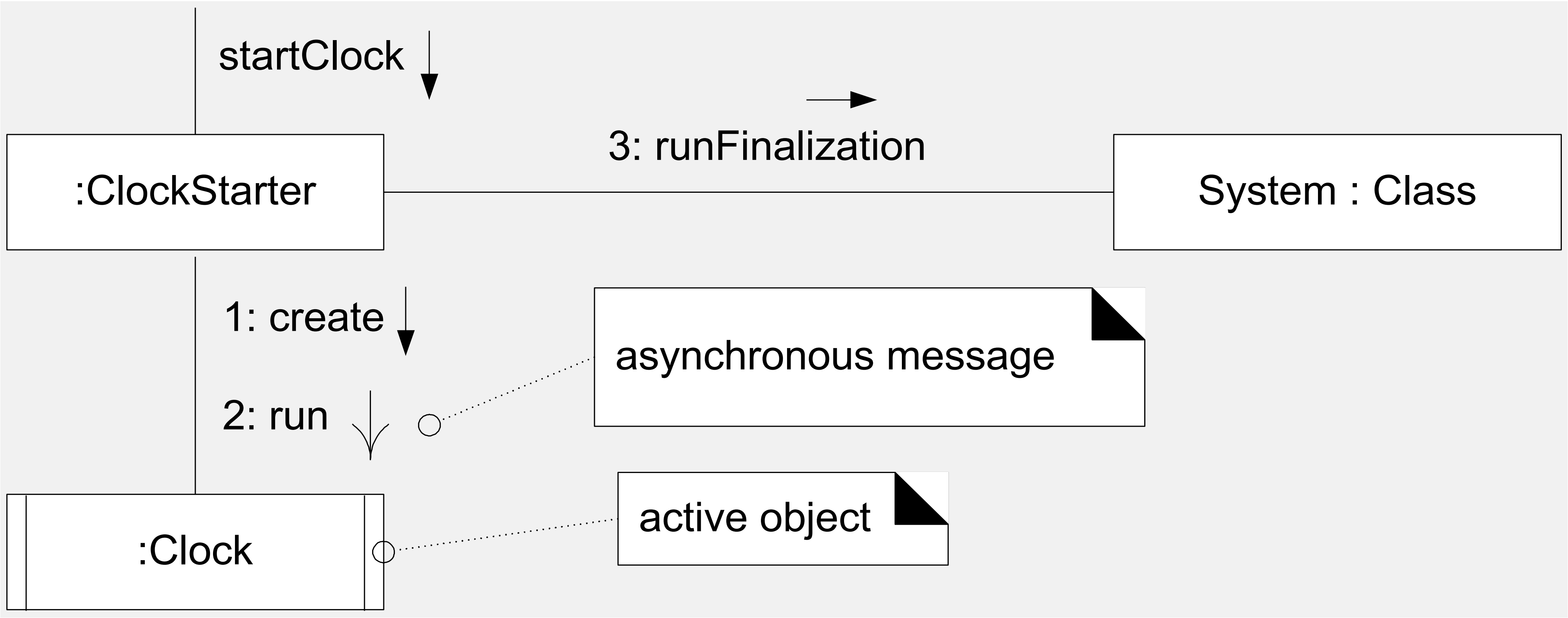
the asynchronous *start* call always invokes the *run* method on the *Runnable* (*Clock*) object

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



Note the dependency with the programming language. For the sake of abstraction and generality, you may want to express concurrency in its simplest form here.

Concurrency



Credits

Notes and figures adapted from

Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development by C. Larman. 3rd edition. Prentice Hall/Pearson, 2005.