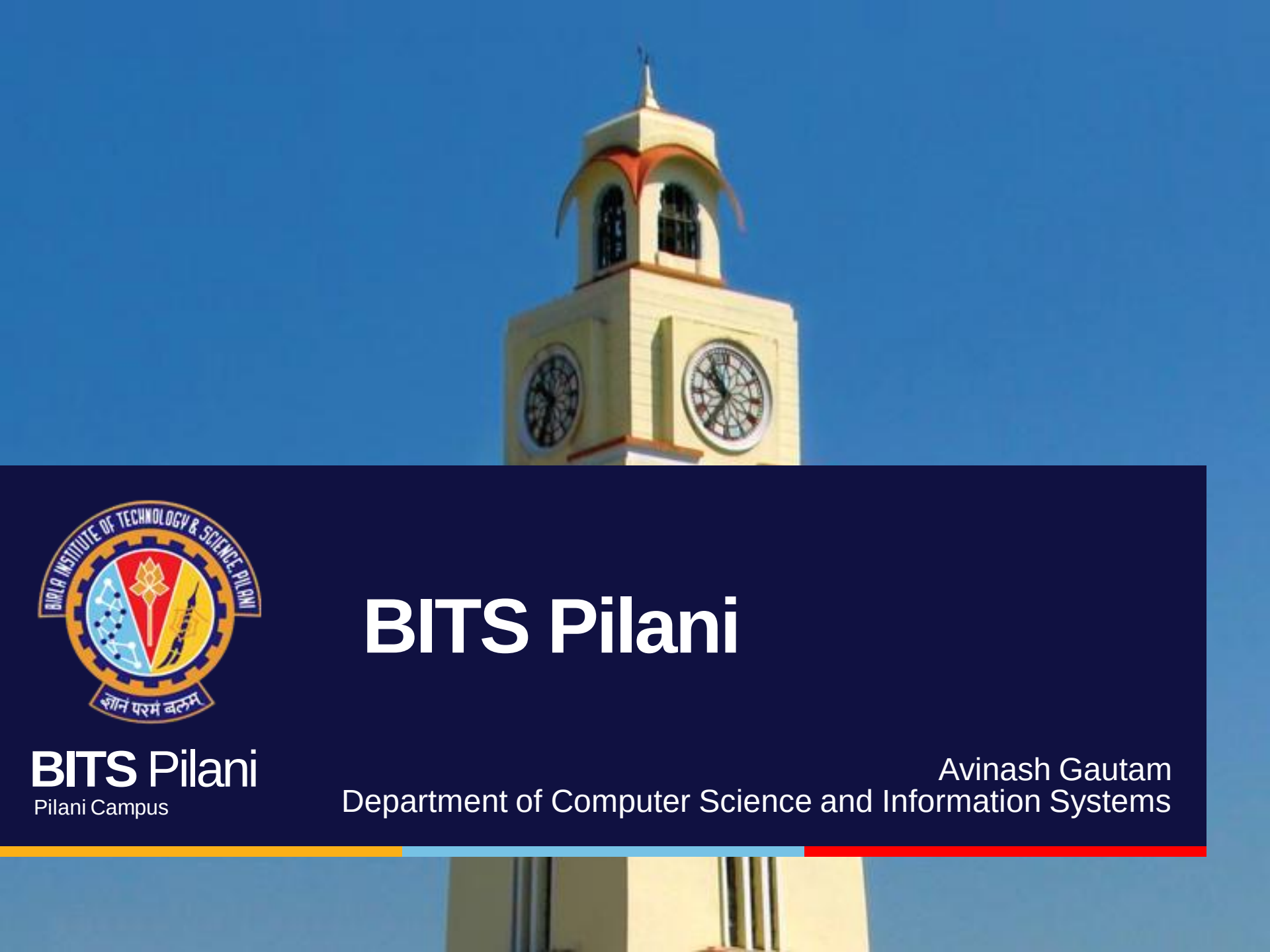


# BITS Pilani


**BITS** Pilani
Pilani Campus

Avinash Gautam
Department of Computer Science and Information Systems

# Chap 11
# Operation Contracts

Chapter 11

Applying UML and Patterns

-Craig Larman

# Operation Contract



- Operation contract identifies ***system state changes*** when an operation happens.
- Define what ***each system operation does***
- An operation is a ***single event*** from the system sequence diagram
- A ***domain model*** is used to help generate an operation contract

# Operation Contract

- When making an operation contract, think of ***the state of the system before the action and the state of the system after the action***.
- The conditions both before and after the action should be described in the operation contract.
- The pre and post conditions describe state, not actions.

# Sections of an Operation Contract

- **Operation**: Name of the operation and parameters
- **Cross References**: Use cases and scenarios this operation can occur within
- **Preconditions**: Noteworthy assumptions about the state of ***the system or objects*** in the Domain Model before execution of the operation.
- **Postconditions:** **This is the most important section**. The ***state of objects in the Domain Model*** after completion of the operation

# Postconditions: most important

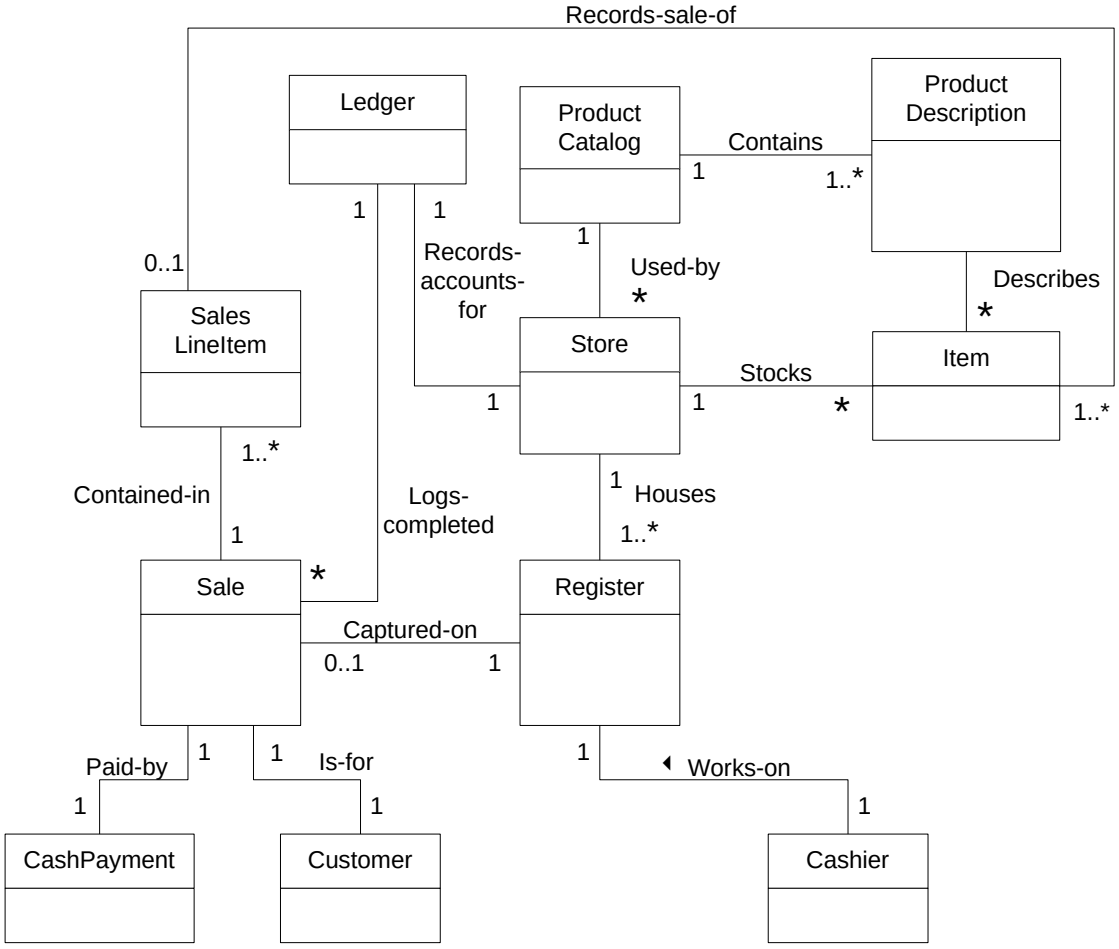
- Describe ***changes in the state of objects in the domain model*** after completion of the operation
  - *what **instances were created** ?*
  - *what associations were formed/broken?*
  - *what **attributes changed**?*
- ***Are not actions to be performed during the operation***

# Examples of Operation Contracts

*From Process Sale Use Case*

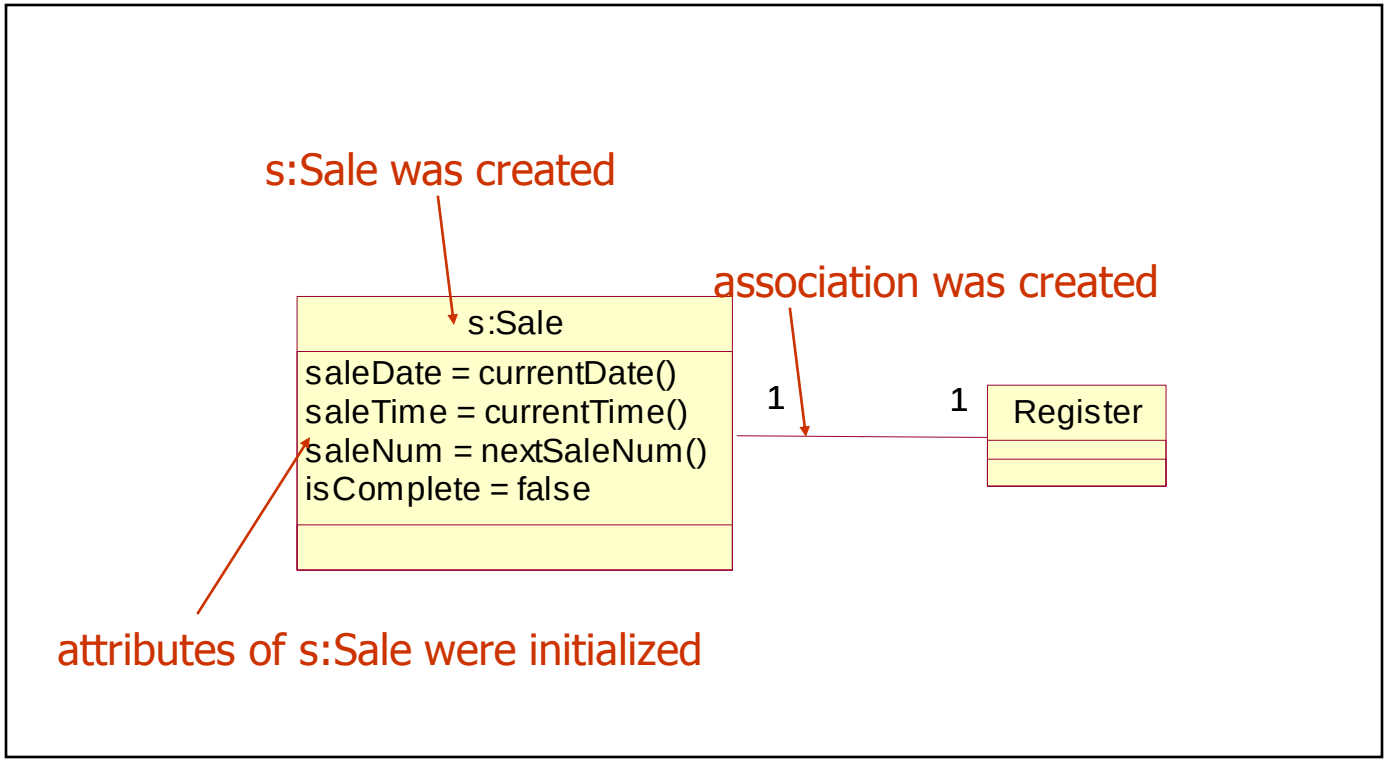
- **Main Success Scenario (or Basic Flow):**
  - *1. Customer arrives at POS checkout with goods and/or services to purchase.*
  - *2. Cashier starts a **new sale**.*
  - *3. Cashier **enters item identifier**.*
  - *4. System records sale line item and presents item description, price, and running total.*
  - *Price calculated from a set of price rules.*
  - *Cashier repeats steps 3-4 until indicates done.*
  - *5. System presents total with taxes calculated.*
  - *6. Cashier tells Customer the total, and **asks for payment**.*
  - *7. Customer pays and System handles payment.*

# POS Domain Model



# Operation Contract for *makeNewSale* operation

- **Operation**: ***makeNewSale***()
- **Cross References**:
  - Use Case: Process Sale
  - Scenario: Process Sale
- **Preconditions**: none
- **Postconditions**
  - *a sale instance "s" was created (instance creation)*
  - *s was associated with the Register (association formed)*
  - *attributes of s were initialized*


s:Sale was created
association was created
s:Sale
saleDate = currentDate()
saleTime = currentTime()
saleNum = nextSaleNum()
isComplete = false
1
1
Register
attributes of s:Sale were initialized

# Operation Contract for *enterItem* operation

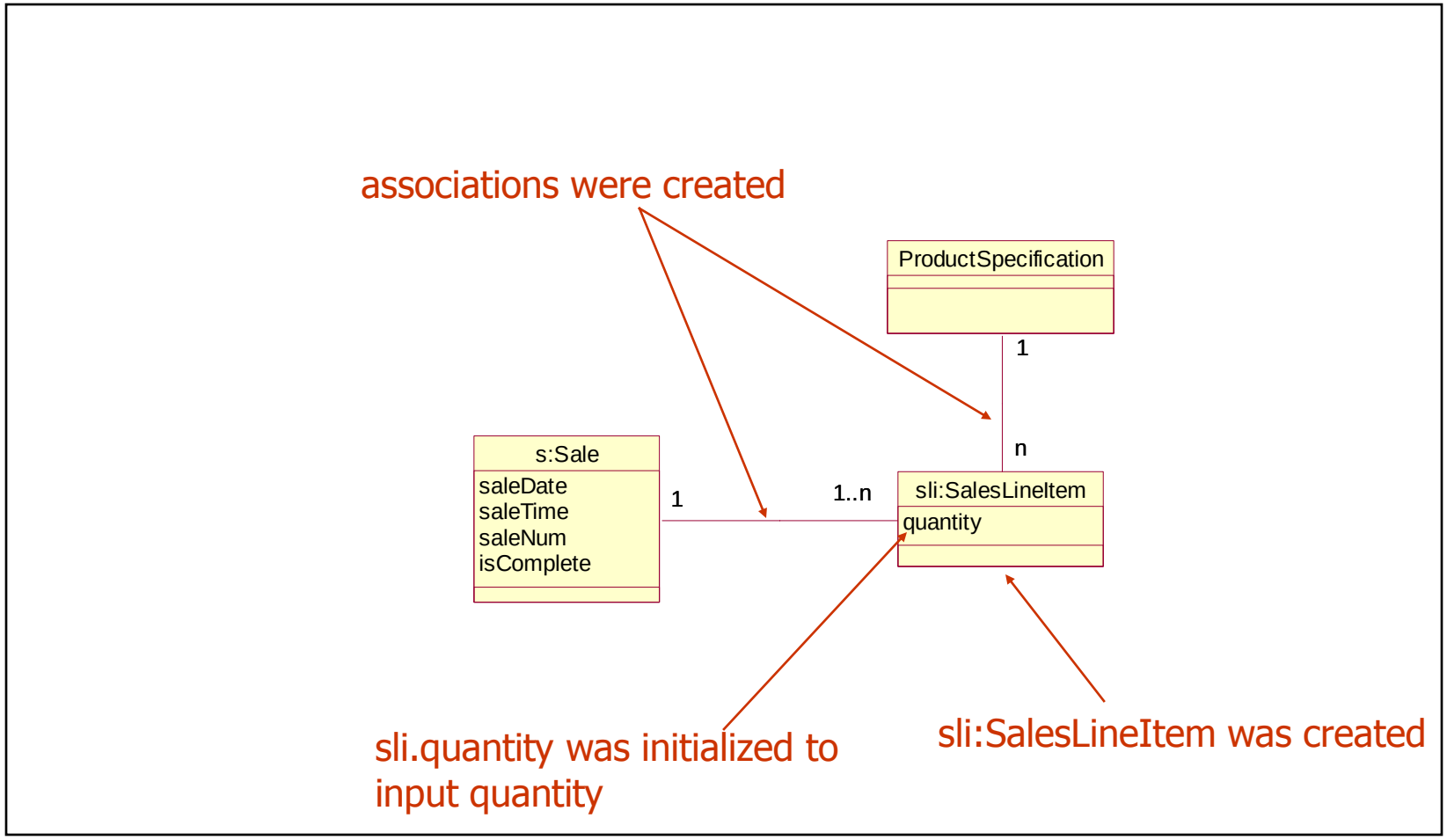


- Operation: enterItem(itemID,quantity)
- Cross References:
  - Use Case: Process Sale
  - Scenario: Process Sale
- Preconditions: There is a sale underway.

# Operation Contract for enterItem operation



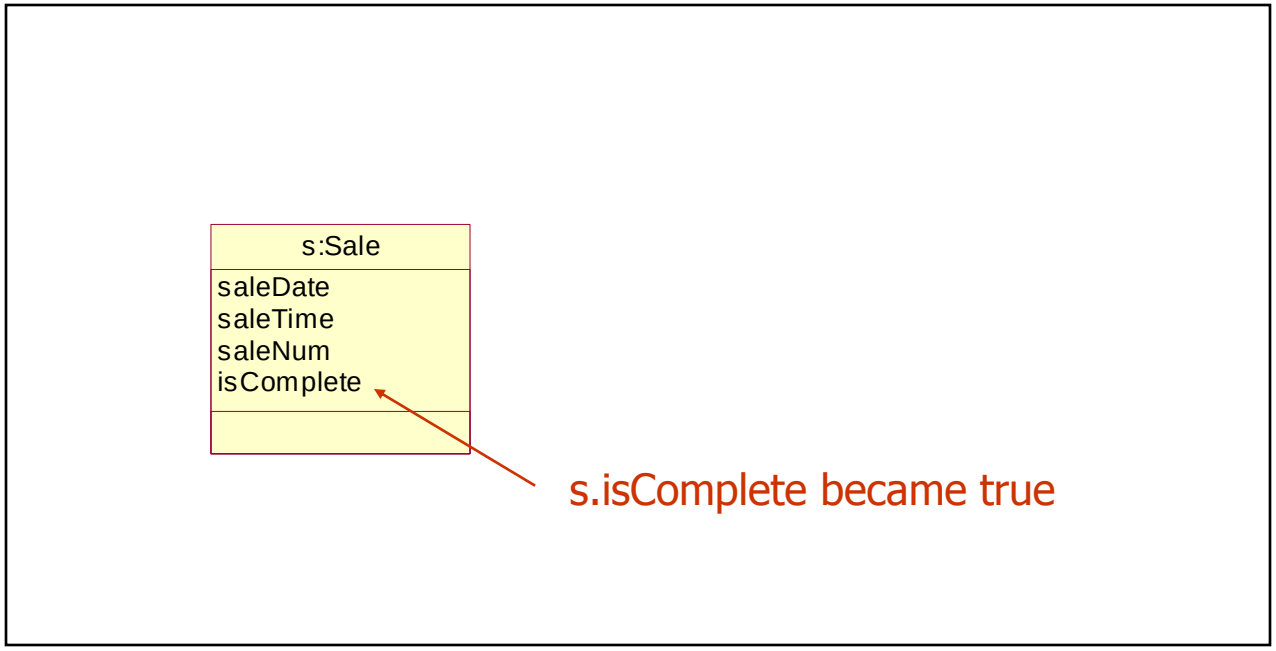
- Postconditions:
  - A salesLineItem instance sli was created (instance creation)
  - sli was associated with the current Sale (association formed)
  - sli.quantity became quantity (attribute modification)
  - sli was associated with a ProductDescription, based on itemId match (association formed)


associations were created
ProductSpecification
1
n
s:Sale
saleDate
saleTime
saleNum
isComplete
1
1..n
sli:SalesLineItem
quantity
sli.quantity was initialized to input quantity
sli:SalesLineItem was created

# Operation Contract for endSale operation



- Operation: endSale()
- Cross References:
  - Use Case: Process Sale
  - Scenario: Process Sale
- Preconditions: There is a sale underway
- Postconditions
  - **s.isComplete** became true (attribute modification)


s:Sale
saleDate
saleTime
saleNum
isComplete
s.isComplete became true

# Operation Contract for makePayment operation

innovate achieve lead

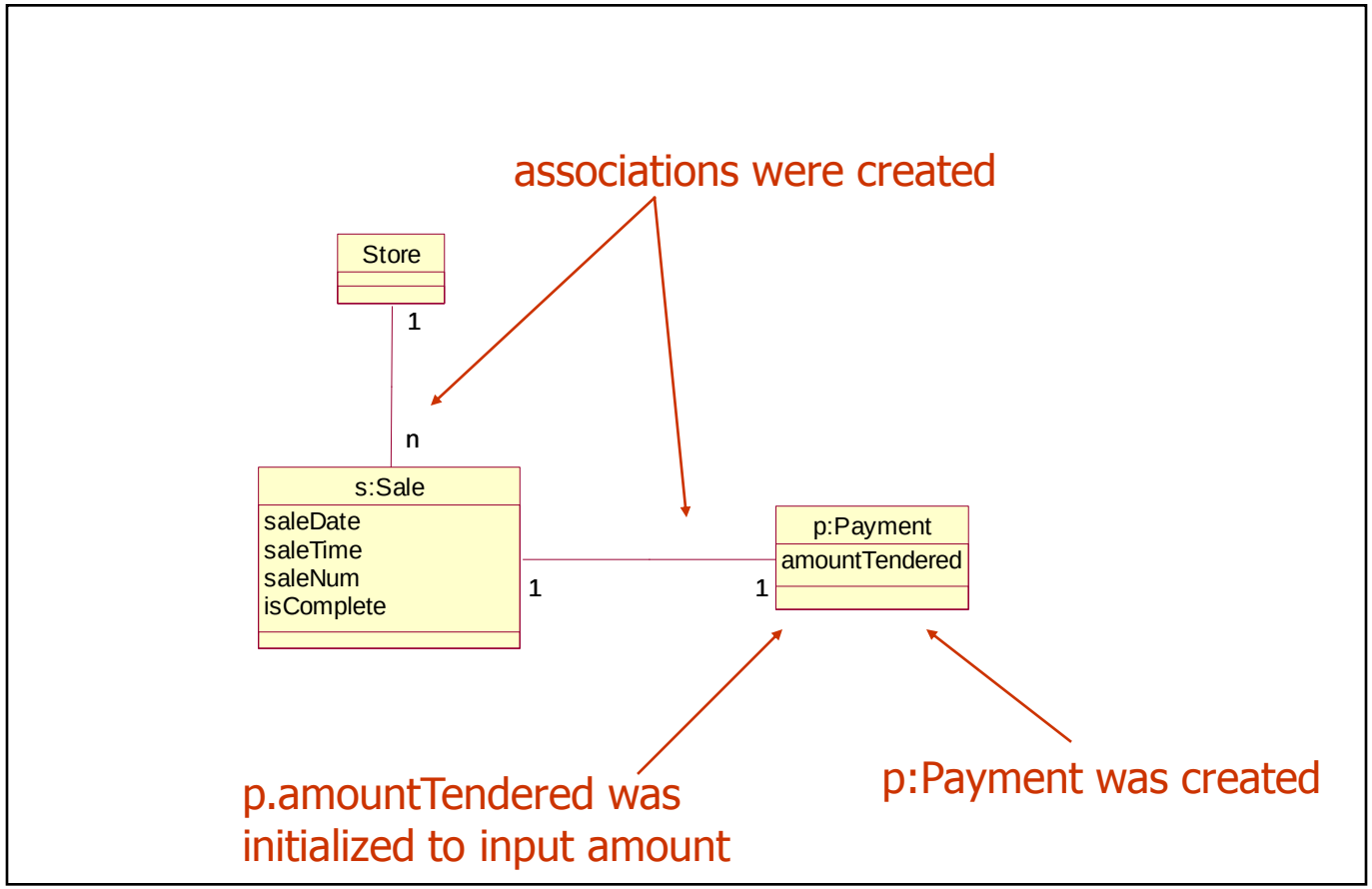
- **Operation: makePayment(amount:Money)**
- Cross References:
  - Use Case: Process Sale
  - Scenario: Process Sale
- Preconditions: There is a sale underway

# Example—Operation Contract for makePayment operation



- Postconditions
  - *a payment instance "p" was created (instance creation)*
  - *p.amountTendered became amount (attribute modification)*
  - *p was associated with s:Sale (association former)*
  - *s:Sale was associated with the Store (association formed)*

# What this looks like:



# Why Operation Contracts?

- An excellent tool of OO requirements analysis that ***describes in great detail the changes required by a system operation*** (in terms of the domain model objects) **without having to describe *how* they are to be achieved**
- Excellent preparation for opening the System black box!

# Chapter 13

## Logical Architecture and UML Package Diagrams

# Software Architecture

- Decisions about the organization of a software system
- Selection of structural elements and their interfaces, along with their behavior.
- Organization of elements into progressively larger systems

# Finally… Software Design

- The logical architecture is the large-scale organization of the classes into packages (namespaces), subsystems, and layers.
- No decision about how they're deployed

# Package Diagrams

- We’ve seen these.  Grouping of UML elements so they can be referred to without having them all in the diagram

# Layers

- A layer is a large grouping of classes, packages, or subsystems that has cohesive responsibility for a major part of the system.
- Higher layers call services in lower layers but not the other way around.

# Typical Layers

- User interface
- Application logic and domain objects
- Technical services – subsystems that support low-level functions, such as interfacing with a database, external hardware, or error logging.
- In strict layering, a layer only calls upon services in the layer directly below it.

# Design with Layers

- Organize the logical structure into distinct layers with related responsibilities, clean separation from lower layers.
- Higher layers are more application-specific

# Layers…

- Address several problems:
  - Source code changes ripple through the entire system
  - Application logic can't be reused because it is intertwined with the UI
  - General business logic is intertwined with application-specific logic
  - High coupling across different areas

# Another view of Layers

- GUI
- Application: handles presentation layer requests, workflow, session state, window/page transitions
- Domain: app layer requests, domain rules, domain services
- Business infrastructure
- Tech services
- Foundation: threads, math, network I/O

# Mapping Code to Layers

Com.mycompany.nextgen.ui.swing

Com.mycompany.nextgen.ui.web

Com.mycompany.nextgen.domain.sales

Com.mycompany.nextgen.domain.payments

Com.mycompany.service.persistence

Org.apache.log4j

Com.mycompany.util

# Domain Vs. Application Logic Layer

- How do we design the application logic with objects?
- Create software objects with names and information similar to the real-world domain. E. g. sales, payments, claims, etc. These are domain objects

# Tiers, Layers, Partitions

- Tiers were originally logical layers, but now the term has come to mean physical nodes.
- Service Oriented Architecture
- Layers are vertical slices, while partitions are horizontal divisions of subsystems within a layer.  E. g. tech services may contain Security and Reporting

# Model-View Separation

- What kind of visibility should packages have to the UI layer?
- How should non-window classes communicate with windows?

# Model-View

- Don’t connect non-UI objects directly to UI objects.  Don’t let a Sale object reference directly a Jframe
- Don’t put application logic, such as tax calculation, in a UI object’s methods.  UI objects should only initialize the elements, receive UI events, and delegate requests for application logic to non-UI objects

# Observer Pattern

- Domain objects send messages to UI objects viewed only in terms of an interface such as PropertyListener.
- Domain classes encapsulate the information and behavior related to the application logic. Windows classes are thin.

# Why Model-View Separation?

- Focus on domain processes rather than the UI
- Allow separate development of UI and the model
- Minimize impact of requirements changes in the UI on the domain layer
- Allow views to be connected to existing domain layer
- Multiple views of same domain object
- Allow porting of UI to another framework
- Allow execution of model independently of UI

# Connections

- SSDs illustrate system operations but hide the UI layer. Yet the UI capture the operation requests
- The UI layer delegates the requests to the domain layer
- Thus messages from the UI to the domain are the messages in the SSDs, such as enterItem.

# Chapter 15

## UML Interaction Diagrams

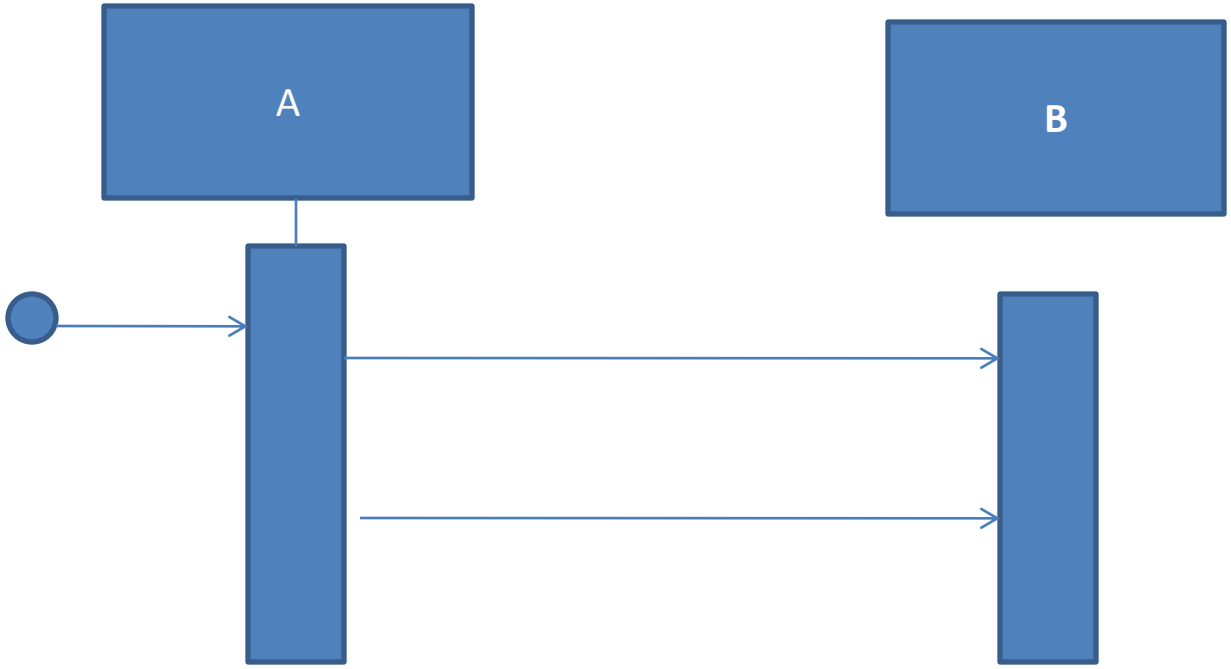
# What are Interaction Diagrams?

- Illustrate how objects interact through messages
- Two types: Sequence Diagrams and Communication Diagrams

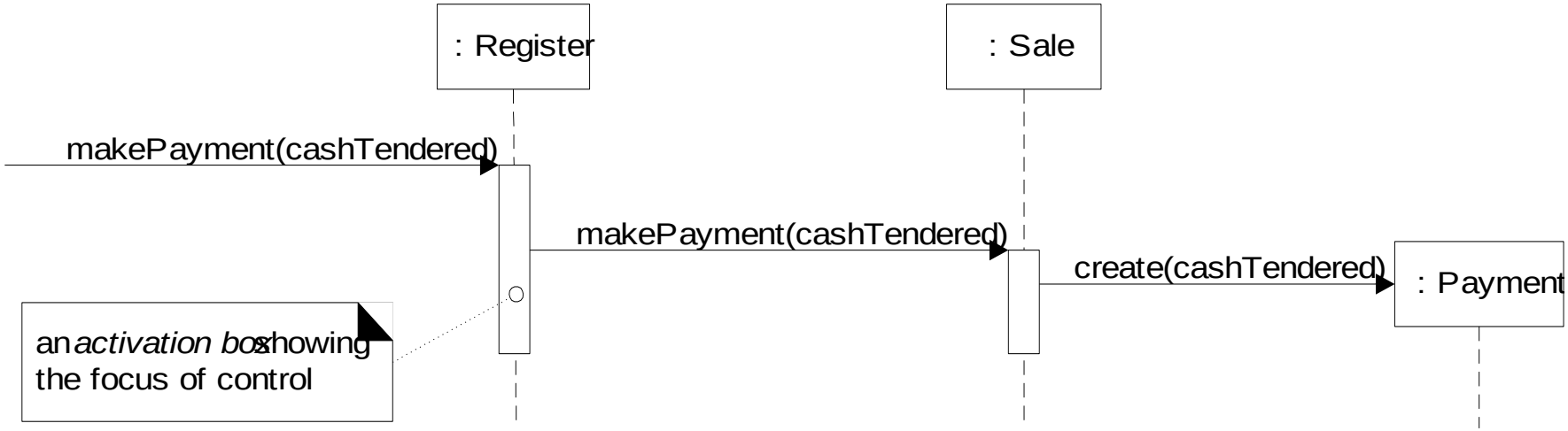
# Example Code

```
Public class A
{
   Private BB myB = new BB();
   Public void doOne()
   {
     myB.doTwo();
     myB.doThree();
   }
}
```

# Sequence Diagram



# Sequence Diagram



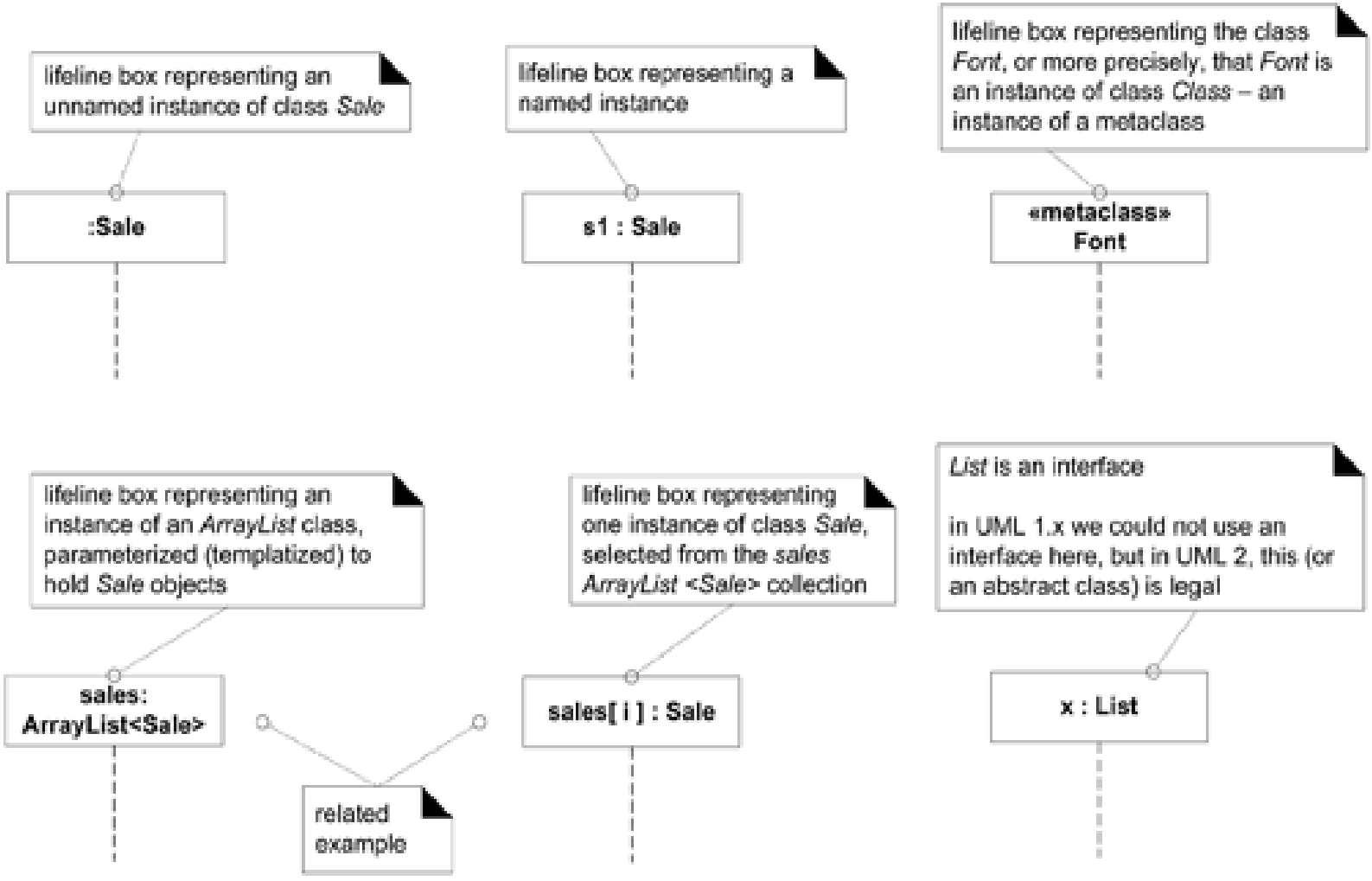
# Guideline

- Spend time doing dynamic modeling with interaction diagrams, not just static object modeling with class diagrams
- This makes you think about how things happen, not just the objects you need

# Sequence vs. Communication

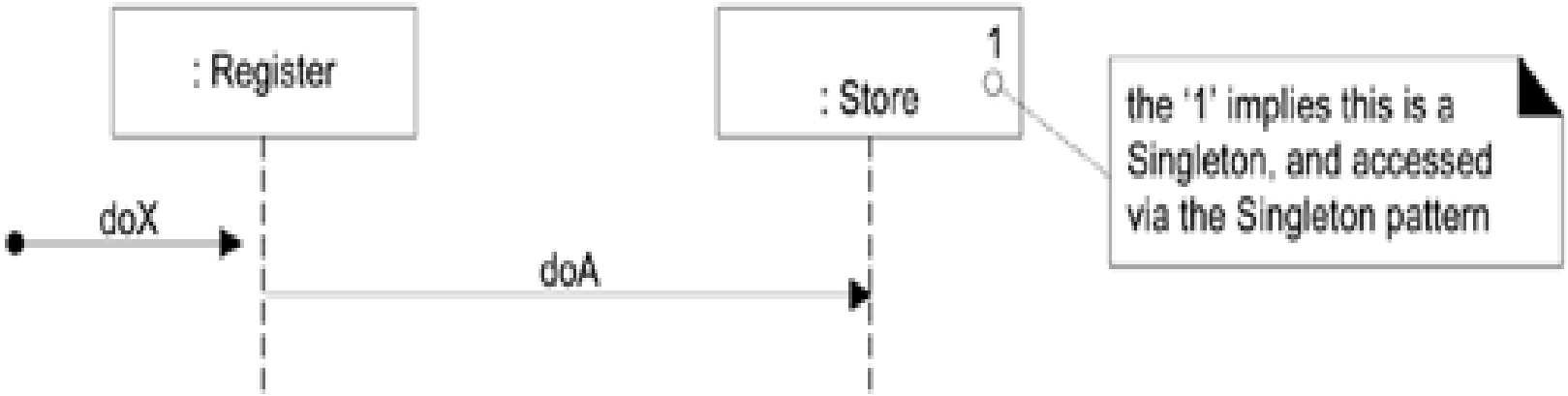
- UML tools usually emphasize sequence
- Easier to see call flow sequence—read from top to bottom
- Communication diagrams are more space-efficient
- Communication Diagrams expand top to bottom; Sequence Diagrams expand left to right

# Common Notation



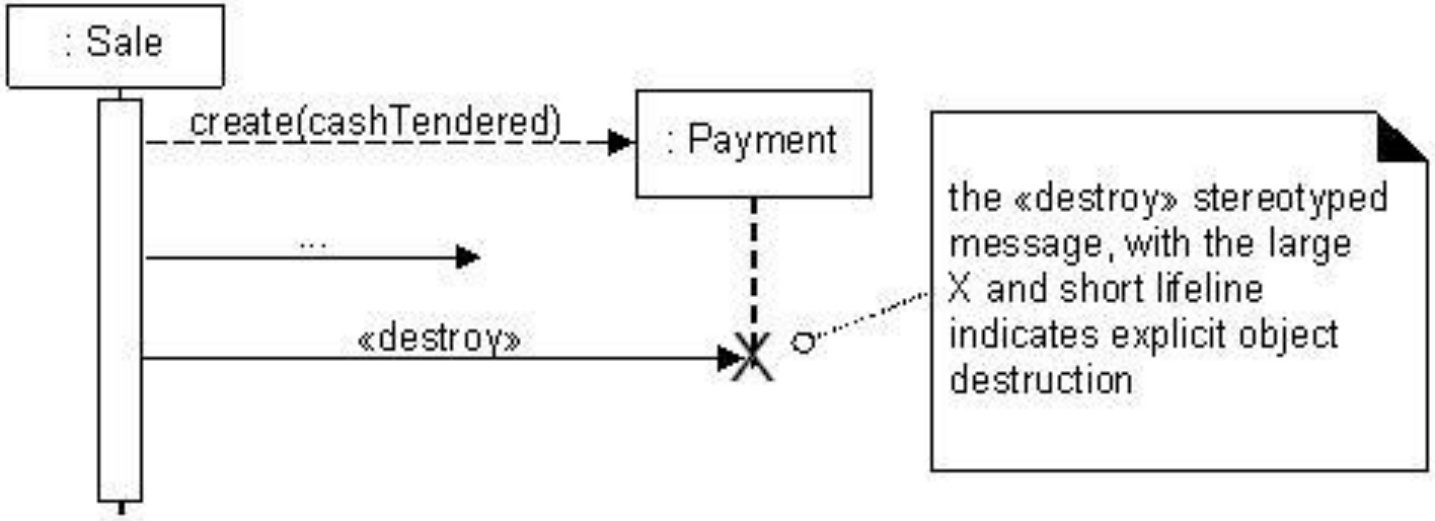
# Singleton Objects

- Only one instance of the class instantiated at any time.

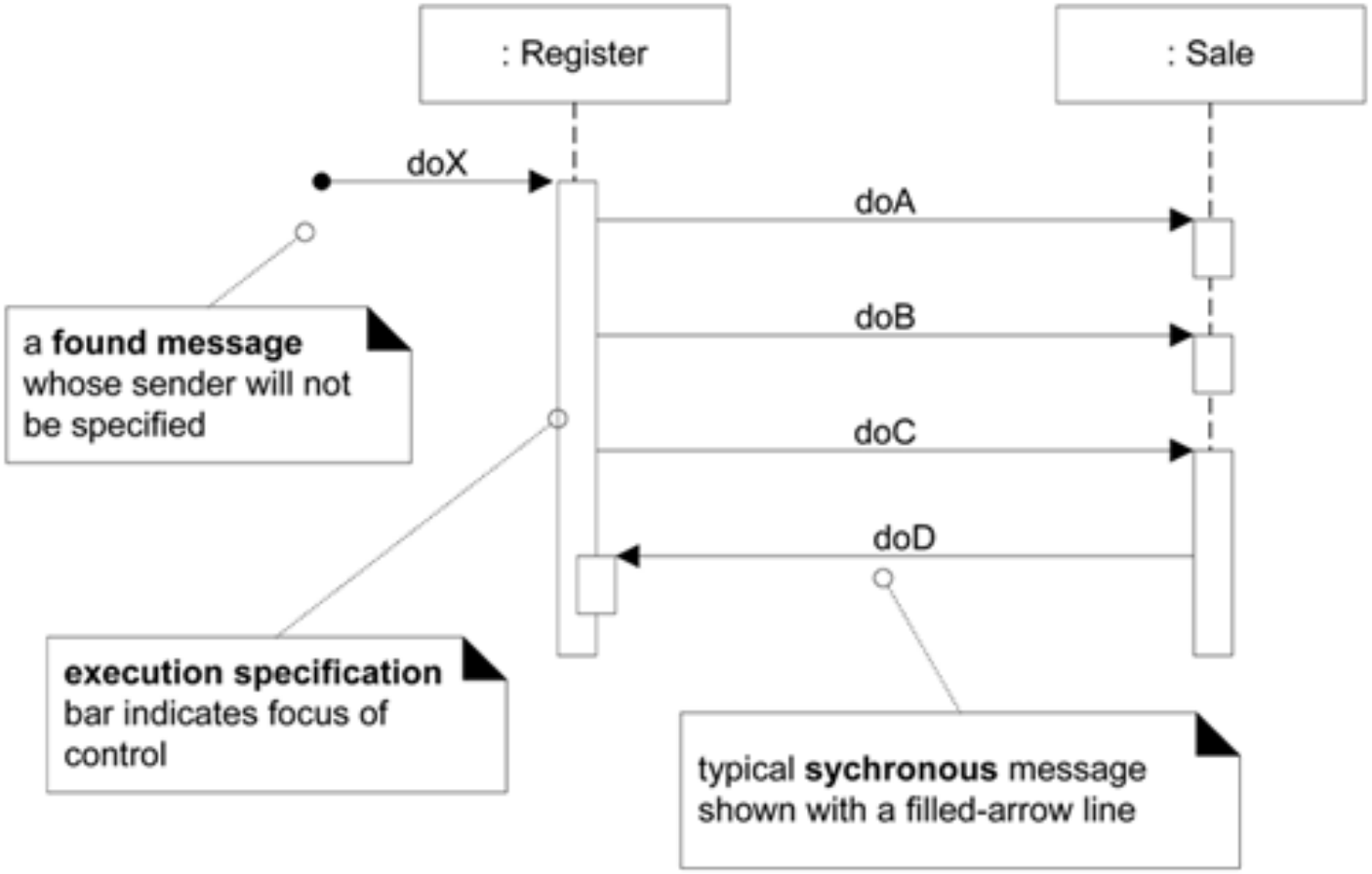


# Lifeline Boxes

- Lifeline is a dashed (or solid, in UML 2) line below the object.



# Messages

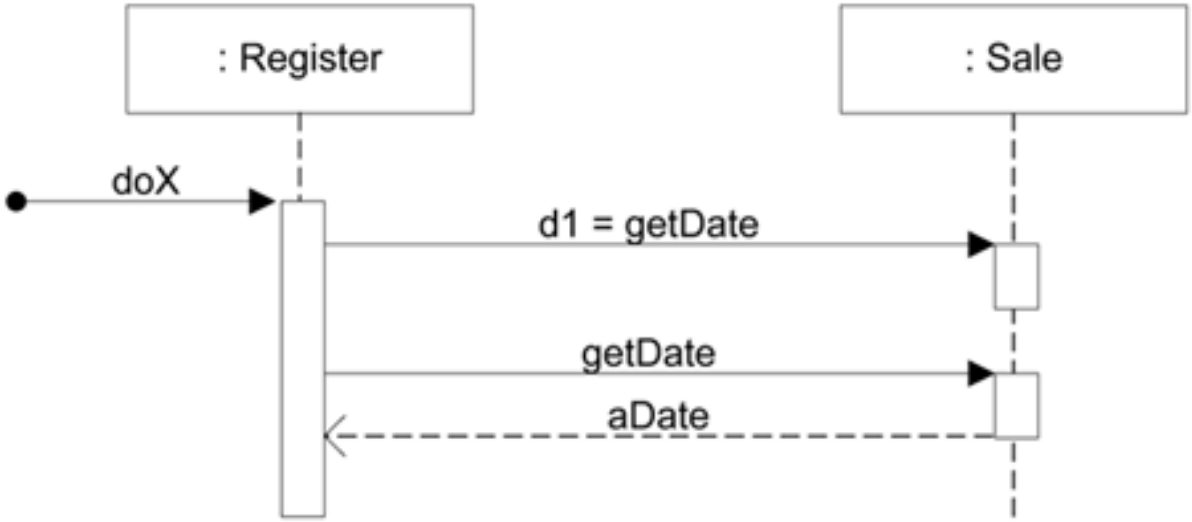

: Register
: Sale
doX
doA
doB
doC
doD
a **found message** whose sender will not be specified
**execution specification** bar indicates focus of control
typical **sychronous** message shown with a filled-arrow line

# Focus of Control

- The bar on the left is an Execution Specification bar. Indicates focus of control

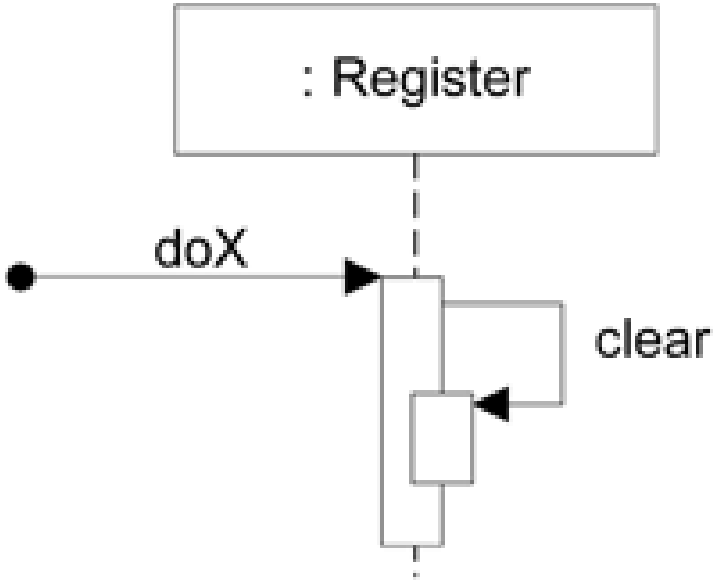
# Reply or Returns

- Use the message syntax returnVar=Message(parameter)
- Using a reply or return message at the end of an activation bar



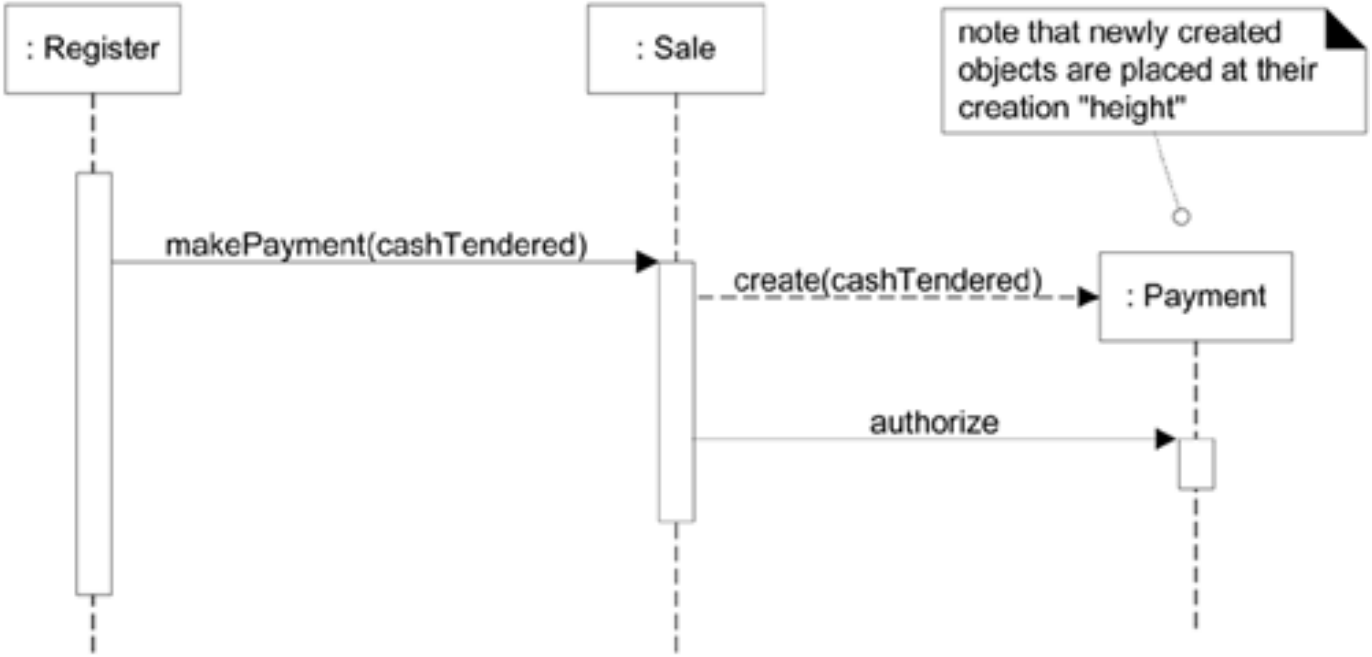
# Messages to Self

- Message sent to itself with a nested activation bar



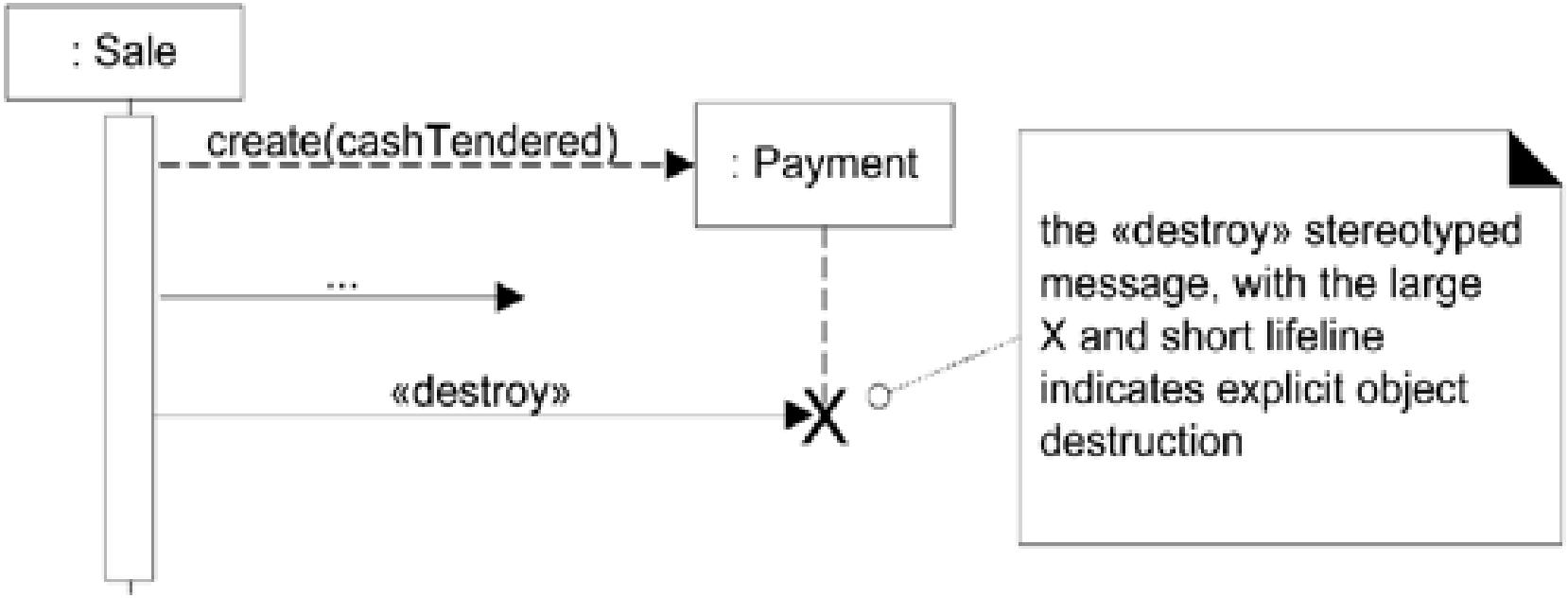
# Creation of Instances

- Dashed line with filled arrow for synchronous message, open if async. Name “create” is not required; it’s a UML idiom



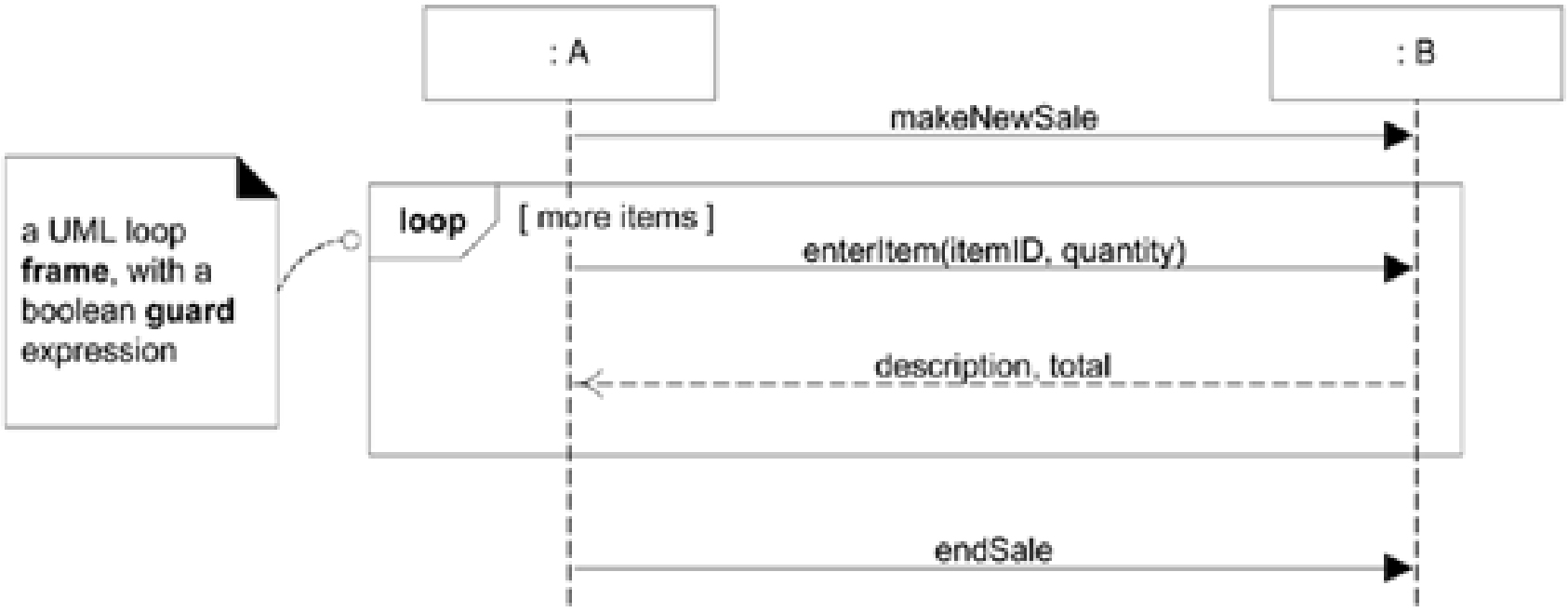
# Lifelines and Object Destruction

- Show explicit destruction of an object, such a when a database connection is closed



# Diagram Frames

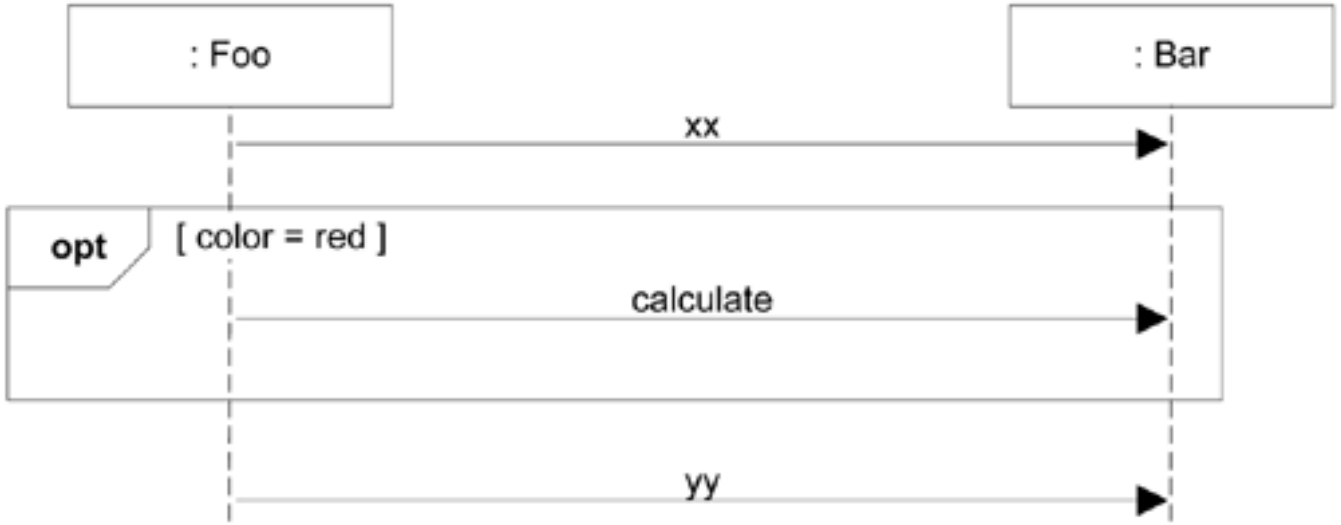
- These support conditionals and looping



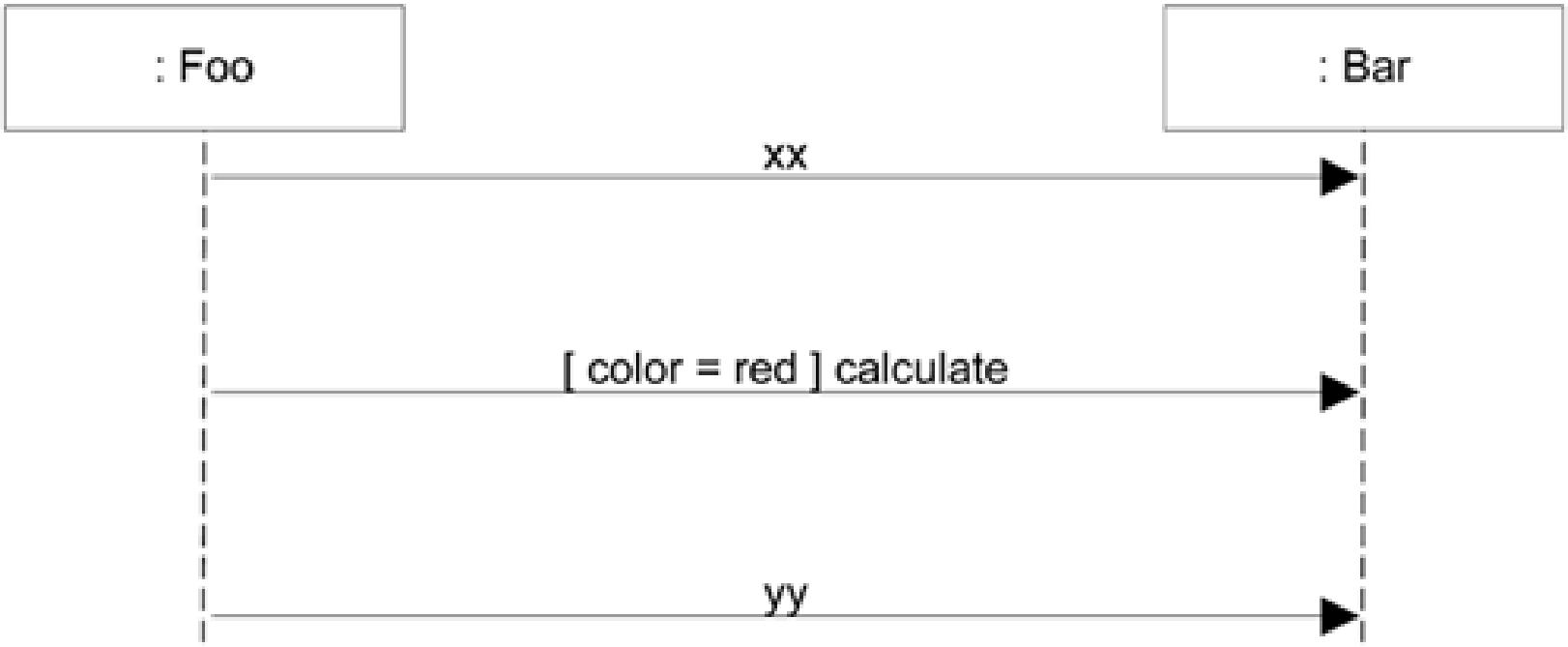
# Frame Operators

| 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. Can be enhanced to define for loop |
| 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 |
| | A “guard” is a Boolean test in brackets over the line to which it belongs |

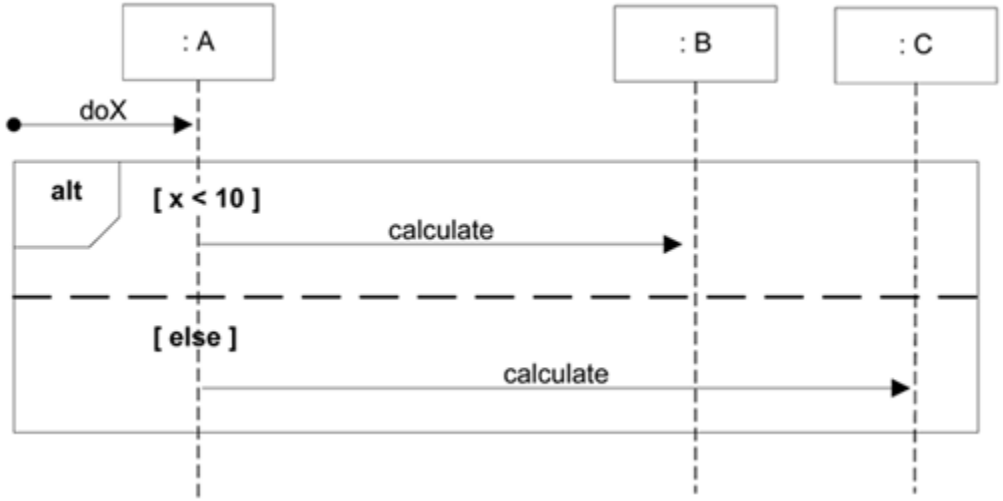
# Conditional Messages



# UML 1.x Conditionals



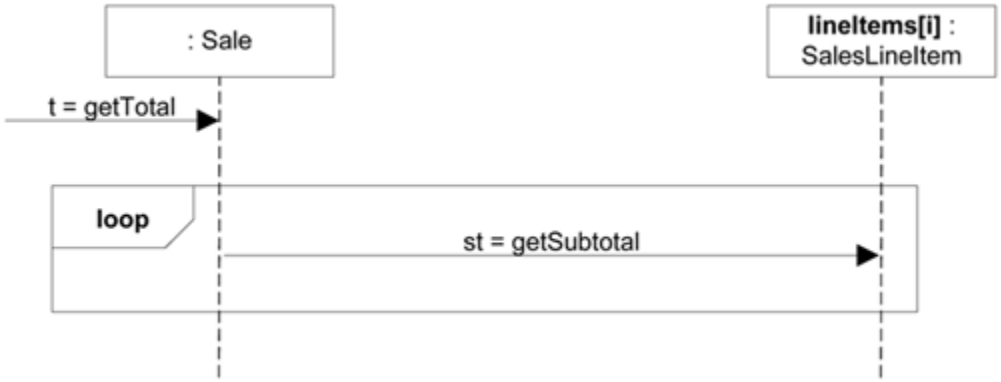
# Conditional If…Else



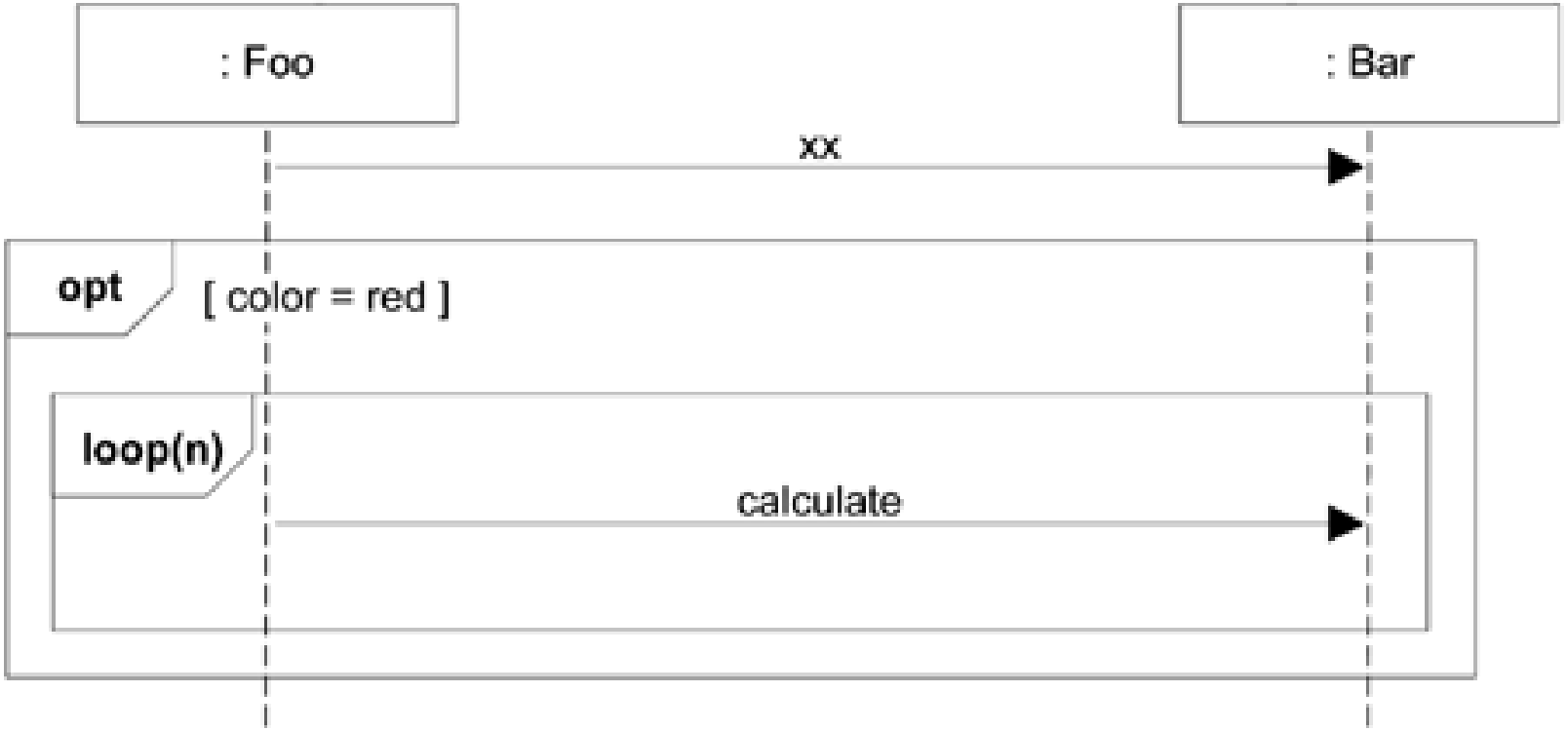
# Code to Iterate Over Collection

```
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; }
// . .  }
```

# Implicit Iteration

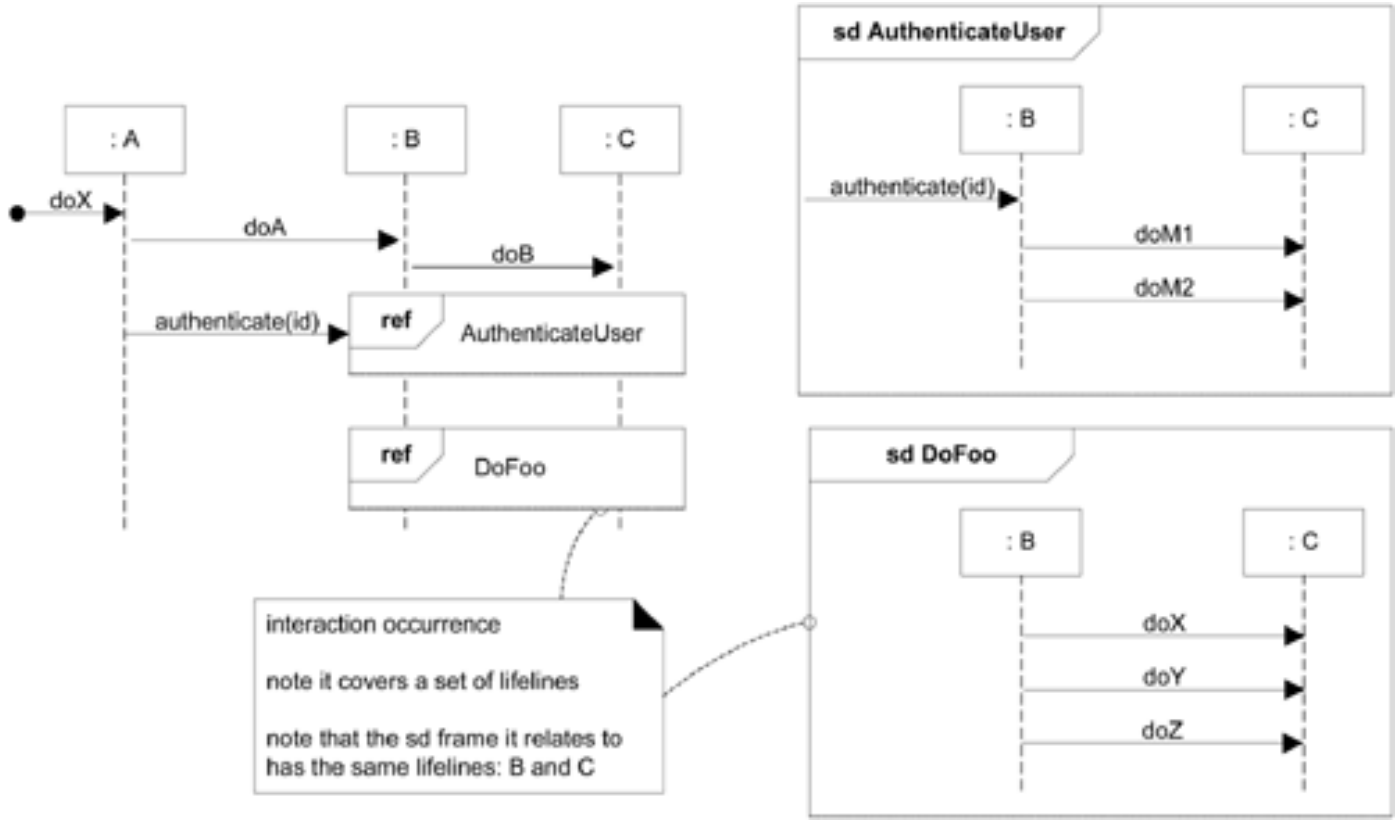


# Nesting Frames



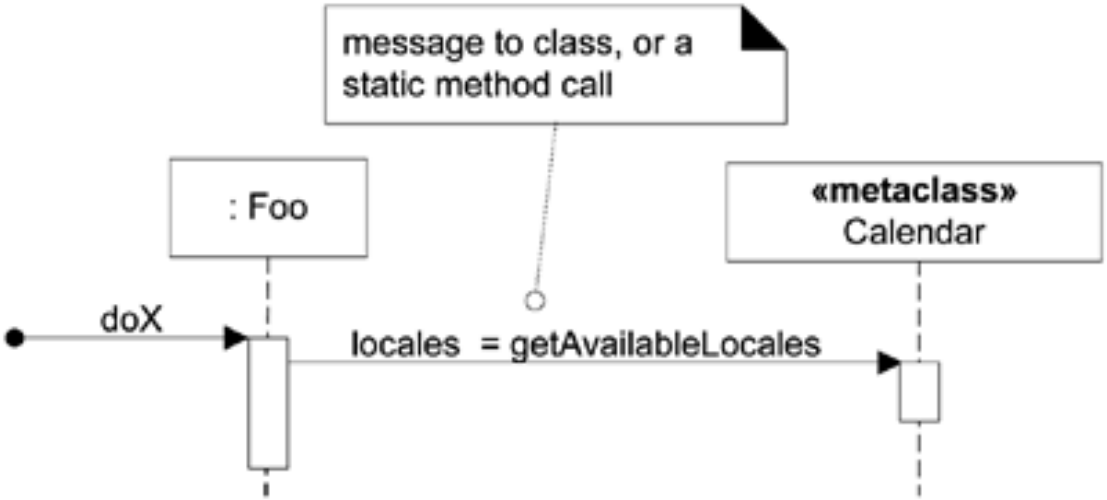
# Relating Interaction Diagrams

- Interaction Occurrence (or Use) is a reference to an interaction within another interaction.



# Invoking Static Methods

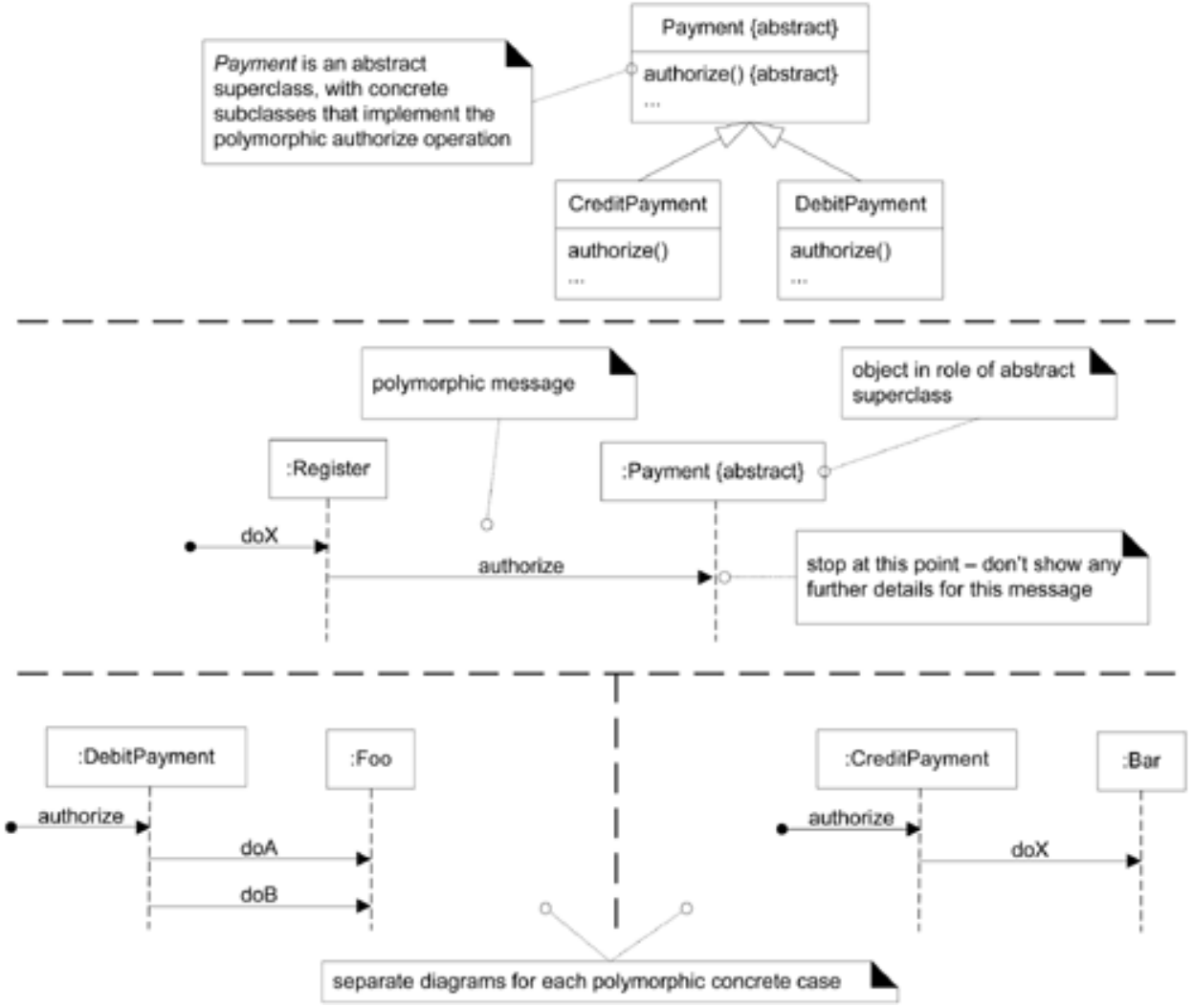
- Metaclasses or static classes are instances of themselves. Calling a static method uses the name of the class, not the name of the instance.



# Polymorphic Messages

- Multiple sequence diagrams: One to the superclass, one to each case, each starting with a found polymorphic message (next slide)

# Polymorphic Messages



# Async and Sync Calls

- Async (non-blocking) calls are a stick (unfilled) arrow, sync are a filled arrow
- This can be used for multi-threading

```
Public class ClockStarter {
Public void startClock {
Thread t = new Thread(new Clock));
t.start();
System.runFinalization();      //
}
}
```

# Async Calls

a stick arrow in UML implies an asynchronous call

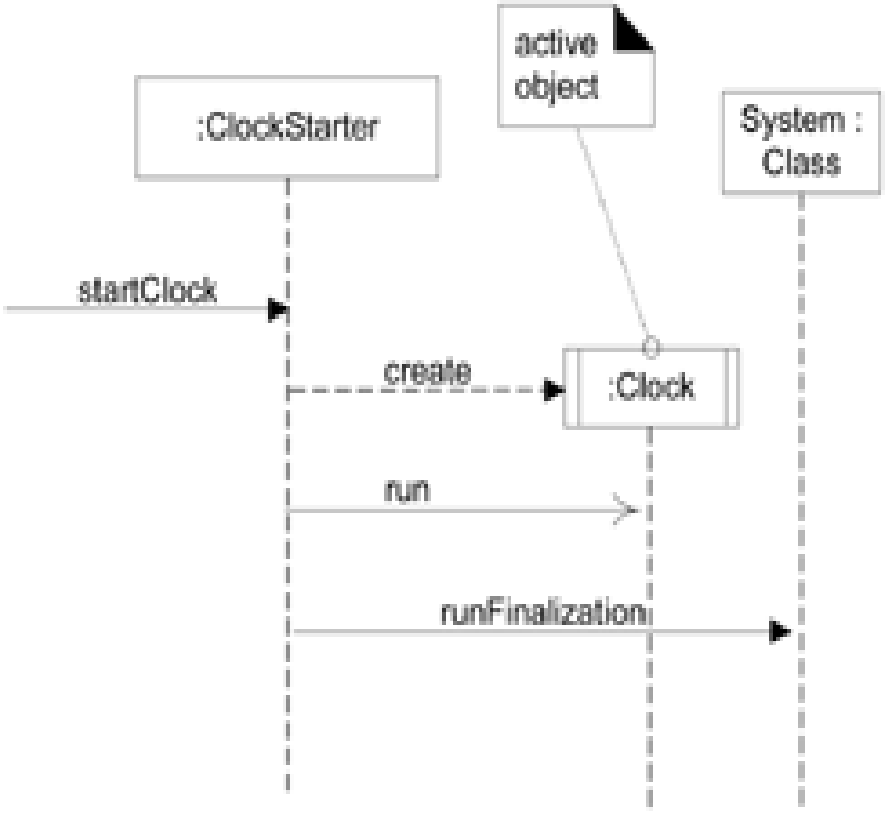
a filled arrow is the more common synchronous call

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

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

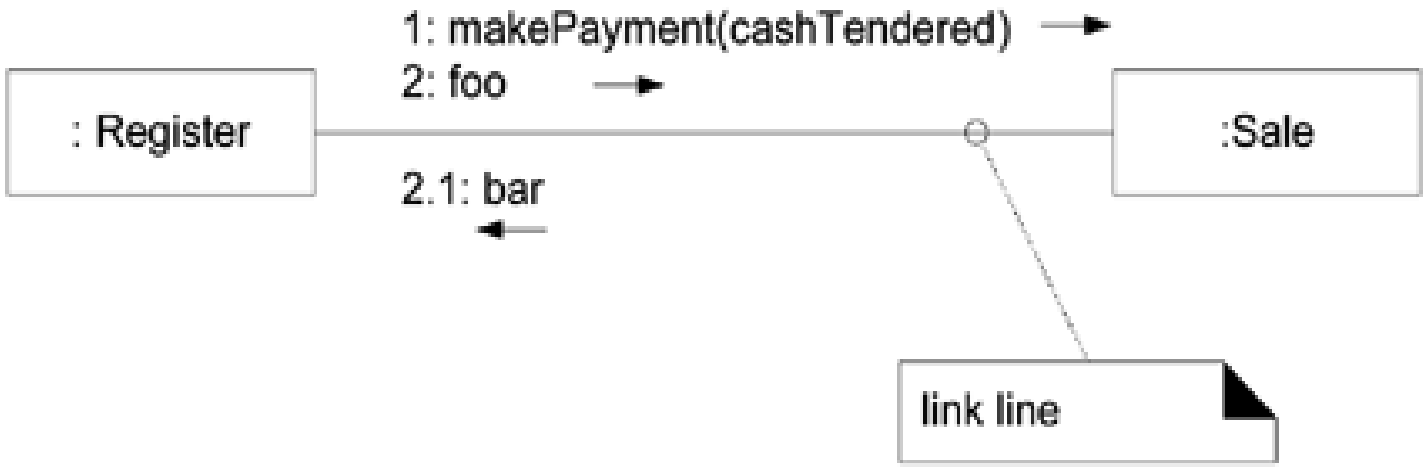
the asynchronous *start* call always invokes the *run* method on the *Runnable* (*Clock*) object

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



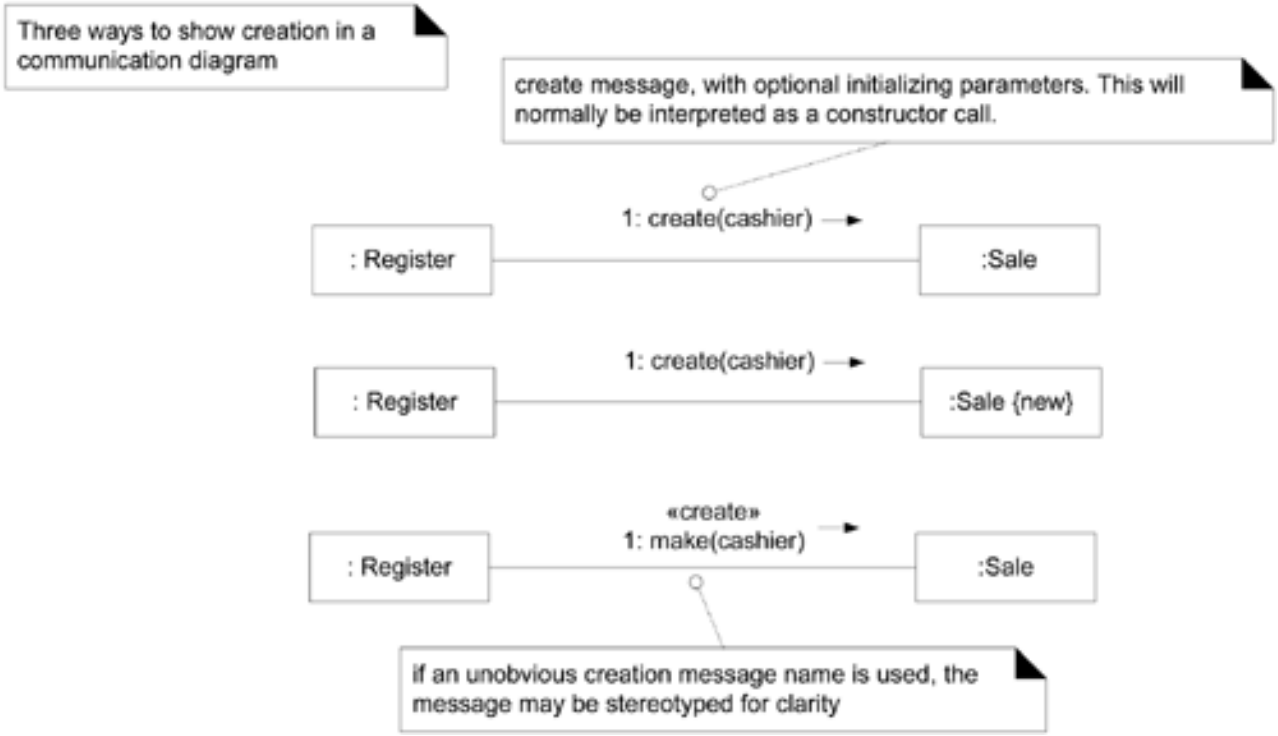
# Links

- Connection path between objects. Navigation and visibility. Instance of an association.
- Multiple messages and messages both ways



# Creation of Instances

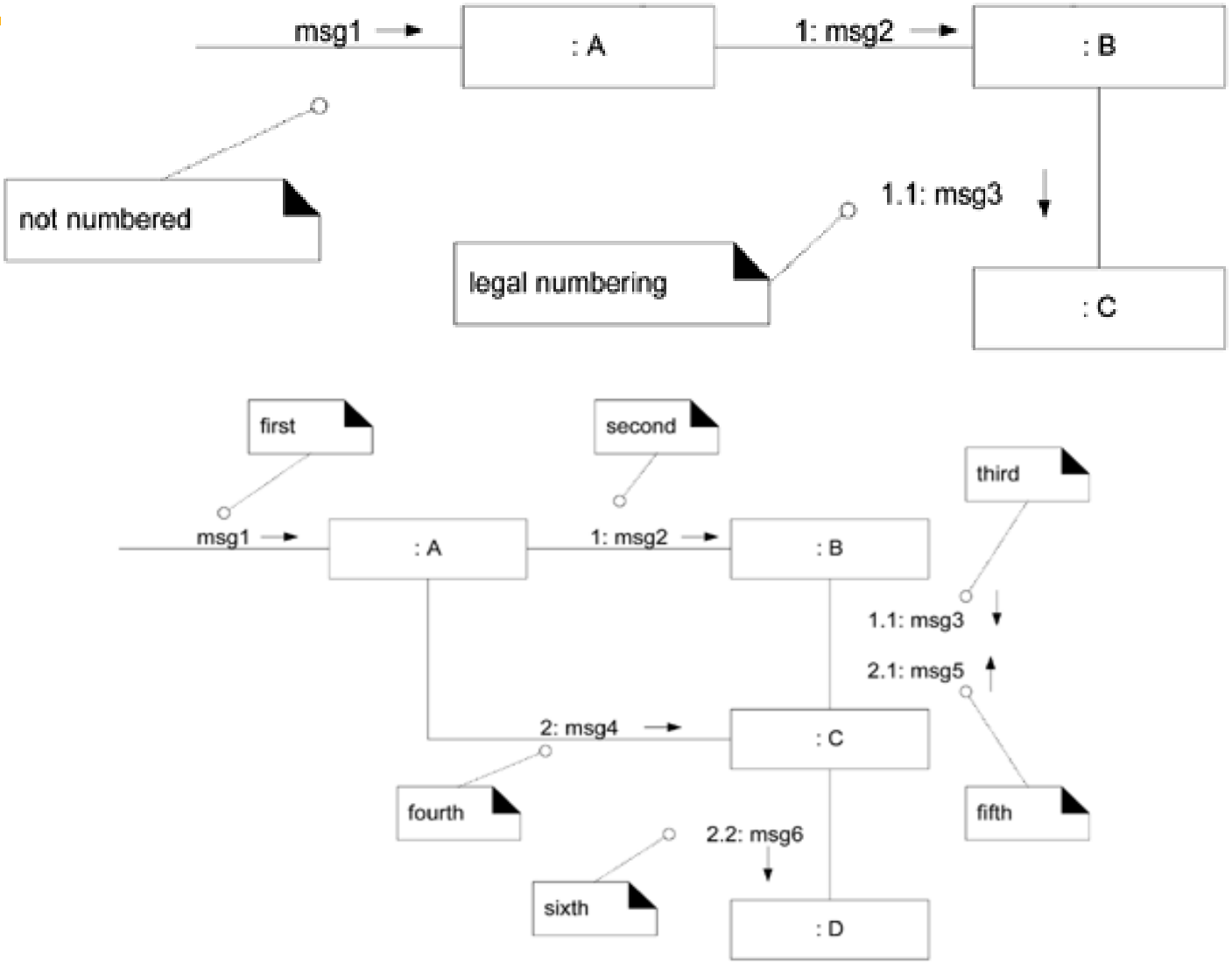
- Message named Create. If some other name is used, use the UML stereotype <<Create>> to indicate. Message can include parameters.
- UML tagged value [new] to indicate creation



# Message Number Sequencing

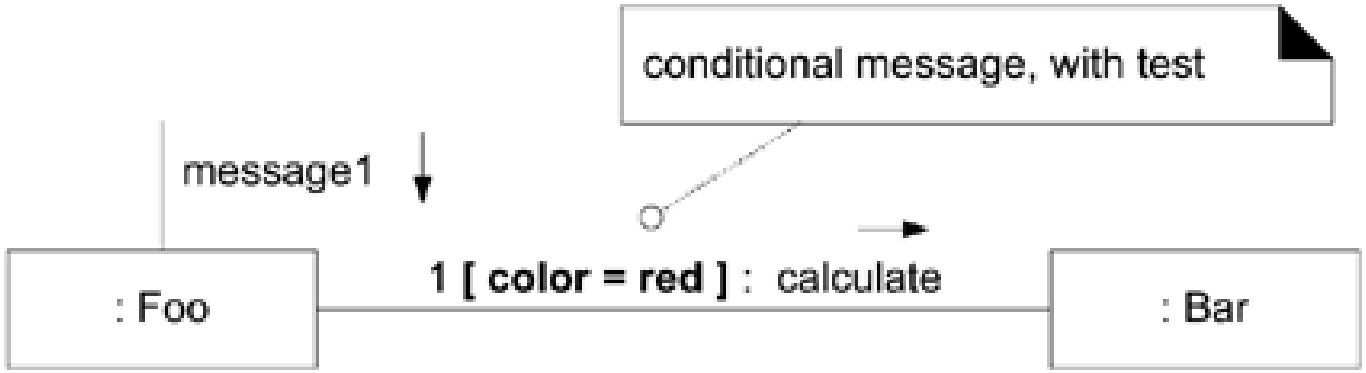
- If ordering is important, indicate this with sequence numbers.
- The first message is not numbered
- Order and nesting of subsequent numbers is shown with levels (next slide)

# Sequencing


msg1
: A
1: msg2
: B
not numbered
1.1: msg3
legal numbering
: C
first
second
third
msg1
: A
1: msg2
: B
1.1: msg3
2.1: msg5
2: msg4
: C
fourth
fifth
2.2: msg6
sixth
: D

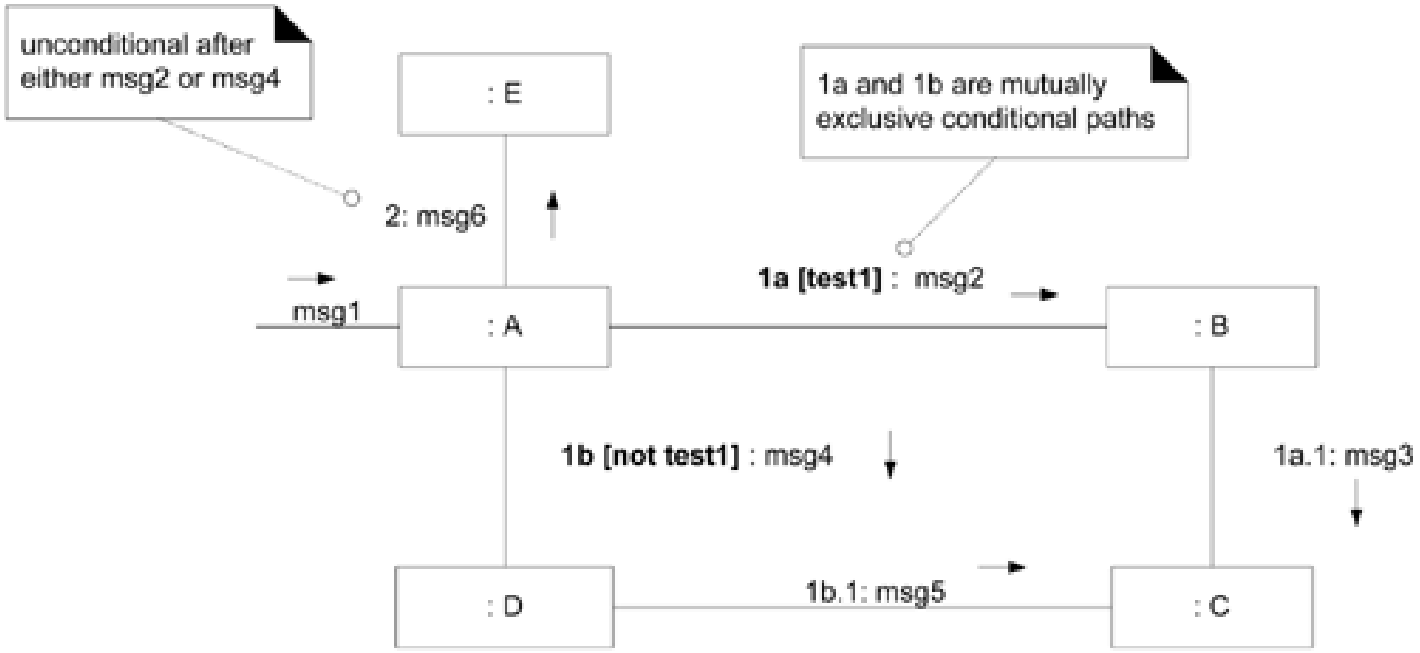
# Conditional Messages

- Show this with the condition in brackets

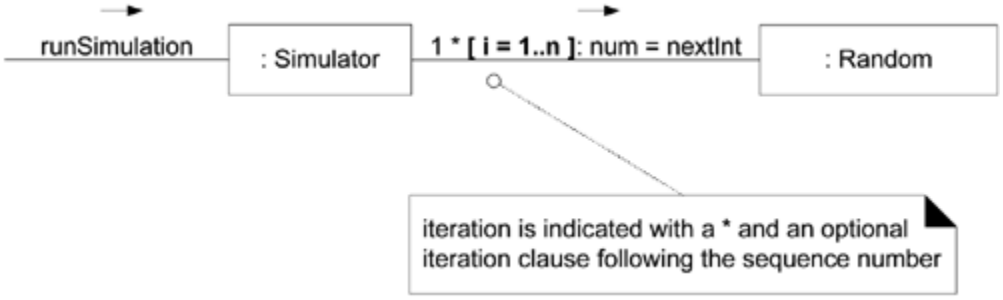


# Mutually Exclusive Conditionals

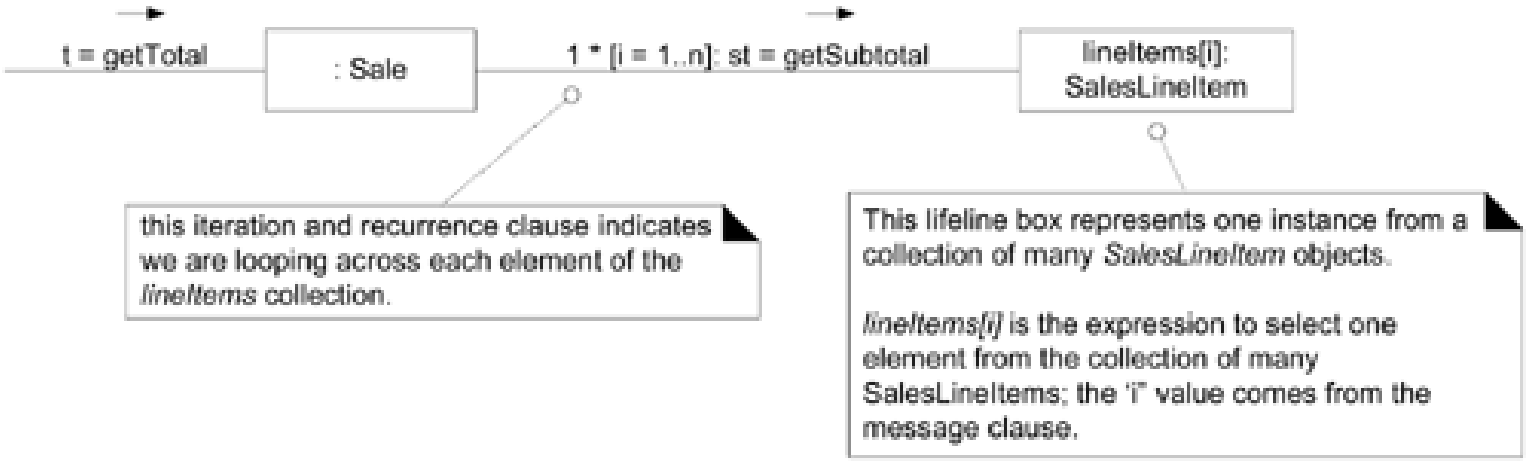
- Equivalent of If…else. Modify the sequence expression with a letter:

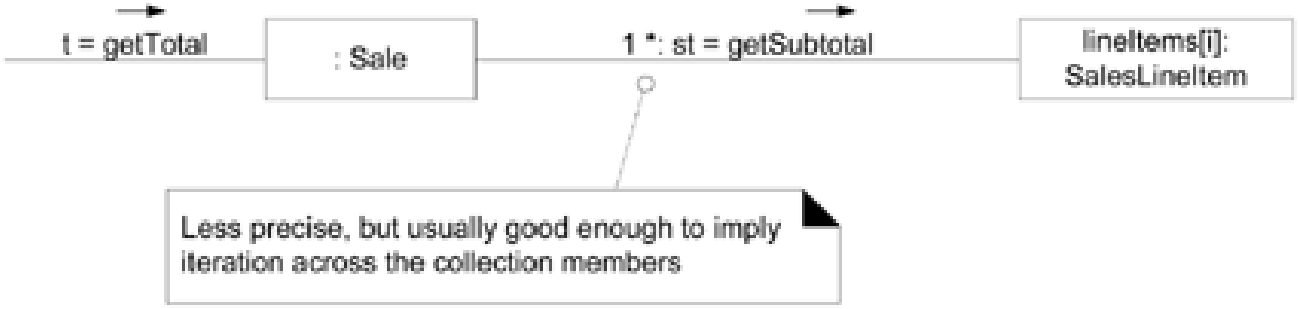


# Iteration


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



# THANK YOU