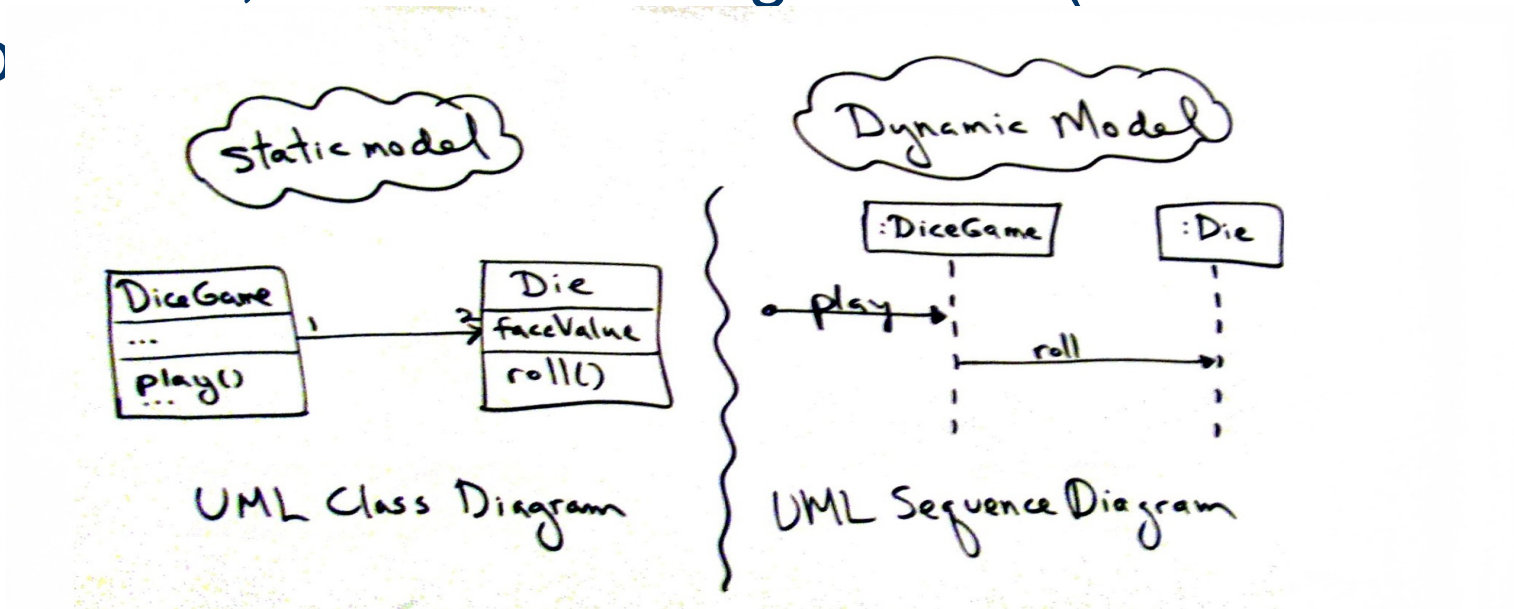
CS 619 Introduction to OO Design and Development

UML Modeling

Fall 2014

Static and Dynamic Modeling

- **Dynamic models**,Such as UML interaction diagrams (sequence diagrams or communication diagrams), help design the logic, the behavior of the code or the method bodies.
- **Static models**, such as UML class diagrams, help design the definition of packages, class names, attributes, and method signatures (but not method bo



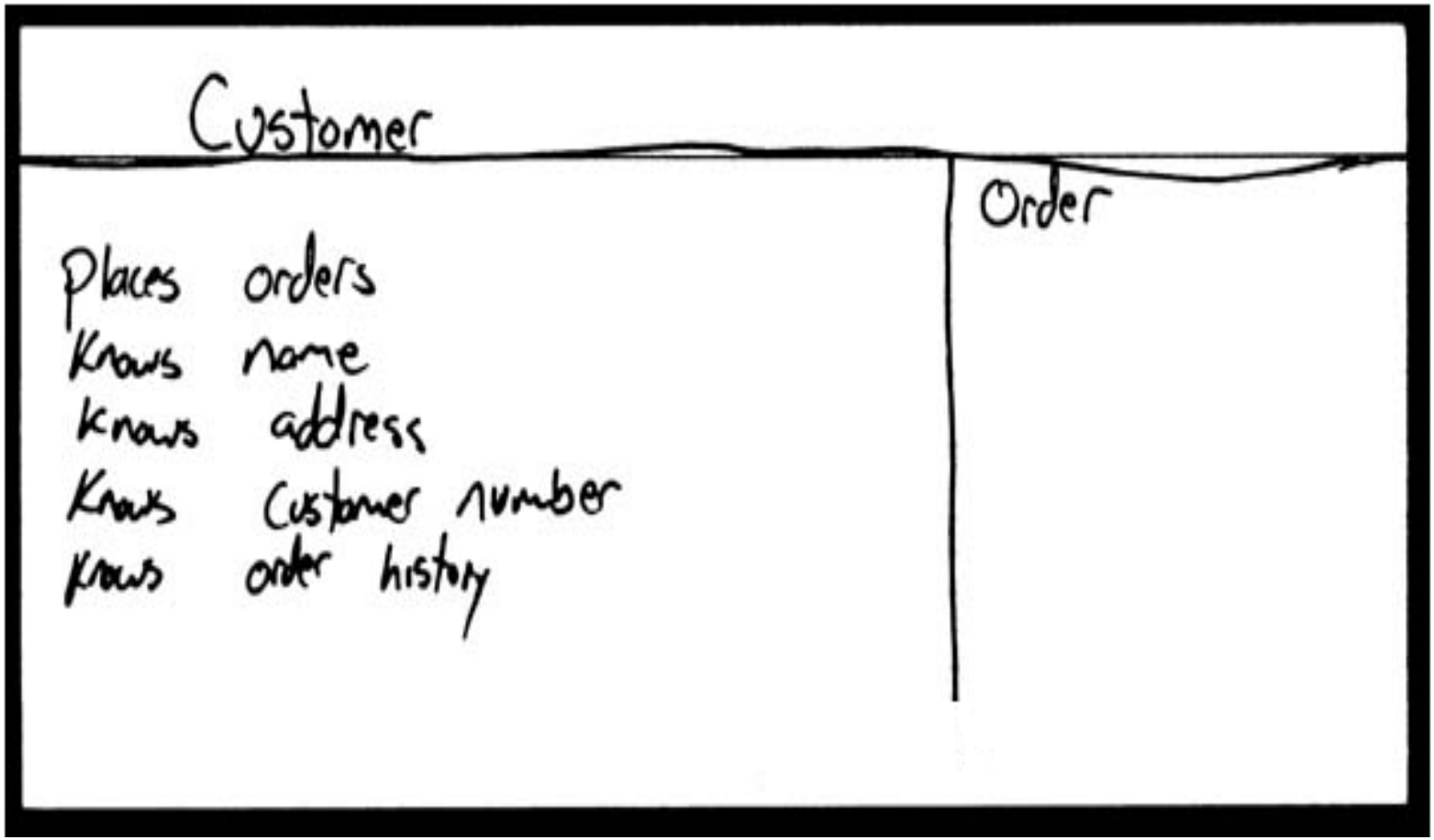
Class-level design

- What classes will we need to implement a system that meets our requirements?
- What fields and methods will each class have?
- How will the classes interact with each other?

How do we design classes?

- Class identification from Software Requirement Specification
  - nouns are potential classes, objects, fields
  - verbs are potential methods or responsibilities of a class
- CRC card exercises
  - write down classes' names on index cards
  - next to each class, list the following:
    - responsibilities: problems to be solved; short verb phrases
    - collaborators: other classes that are sent messages by this class (asymmetric)

CRC cards

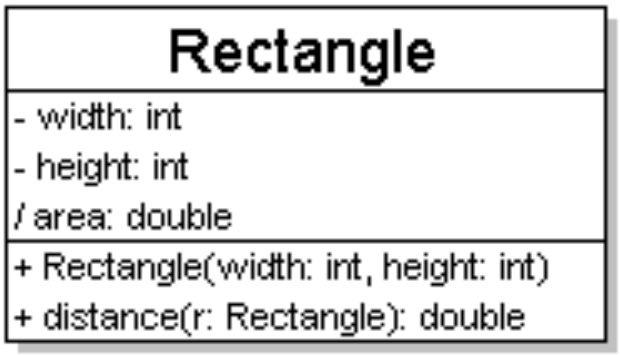

Customer
Order
Places orders
Knows name
knows address
Knows Customer number
Knows order history

Introduction to UML

- **Unified Modeling Language (UML)**: depicts an OO system
  - programming languages are not abstract enough for OO design
  - UML is an open standard; lots of companies use it
    - many programmers either know UML or a "UML-like" variant
- UML is ...
  - a *descriptive* language: rigid formal syntax (like programming)
  - a *prescriptive* language: shaped by usage and convention
  - UML has a rigid syntax, but some don't follow it religiously

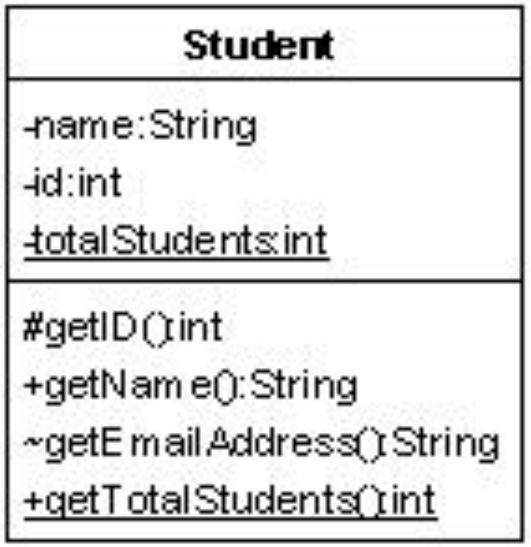
Diagram of one class

- **class name** in top of box
  - write **<<interface>>** on top of interfaces' names
  - use *italics* for an *abstract class* name



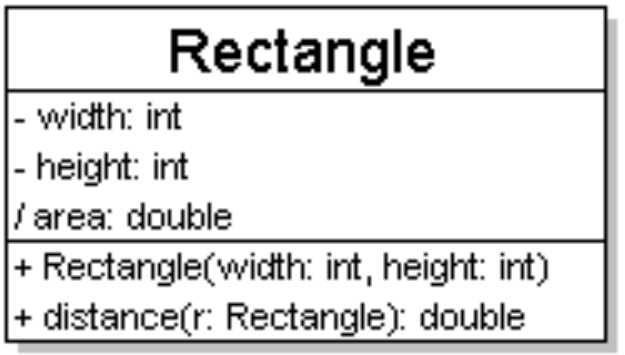
- **attributes**
  - should include all fields of the object
  - also includes derived "properties"

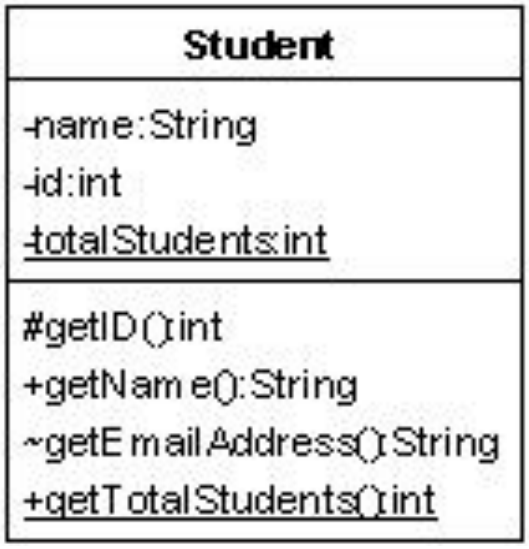
- **operations** / methods
  - may omit trivial (get/set) methods
    - but don't omit any methods from an interfac
  - should not include inherited methods



Class attributes

- attributes (fields, instance variables)
  - *visibility name* : *type* [*count*] = *defaultValue*
  - visibility: + public
    # protected
    – private
    ~ package (default)
    / derived
  - underline static attributes
  - **derived attribute**: not stored, but can be computed from other attribute values
  - attribute example:
    - balance : double = 0.00

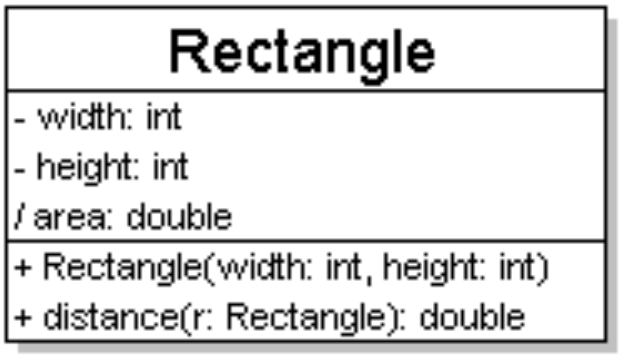


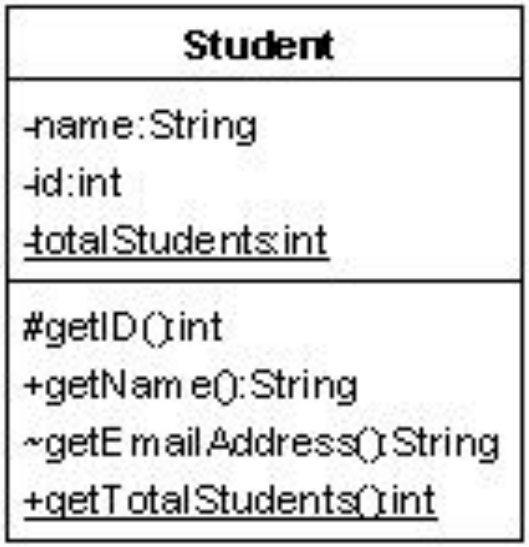


Class operations / methods

- operations / methods
  - *visibility name* (*parameters*) : *returnType*
  - underline static methods
  - parameter types listed as (name: type)
  - omit *returnType* on constructors and when return is `void`
  - method example:

    + distance(p1: Point, p2: Point): double



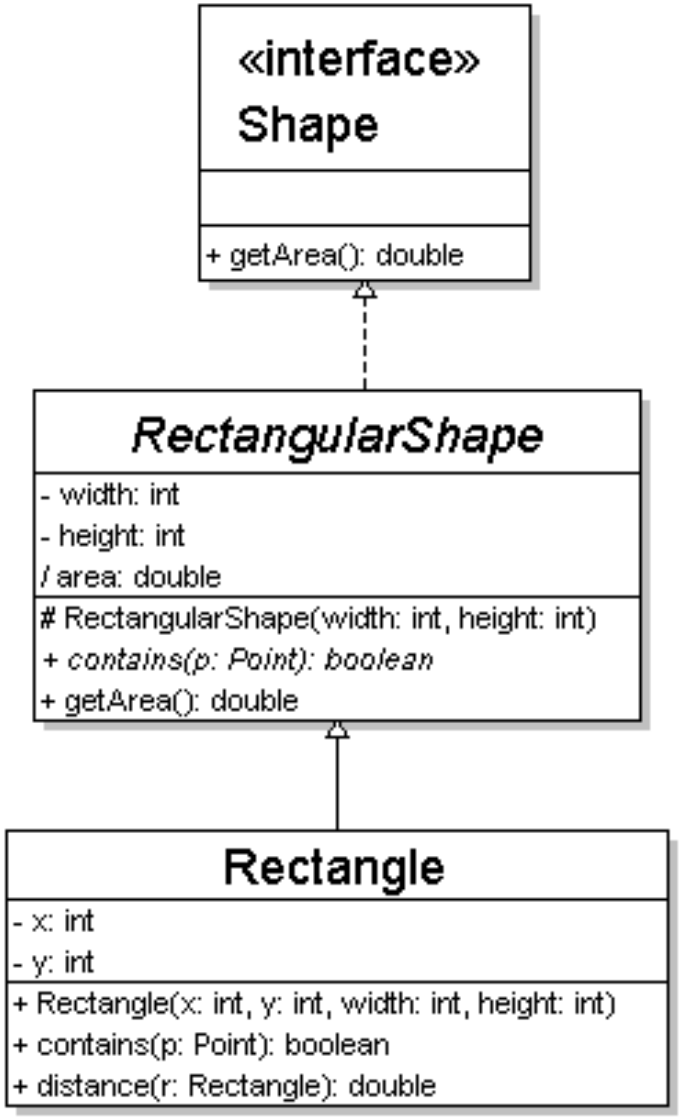


Inheritance relationships

- hierarchies drawn top-down with arrows pointing upward to parent

- line/arrow styles differ based on parent:
  - *class* : solid, black arrow
  - *abstract class* : solid, white arrow
  - *interface* : dashed, white arrow

- Often don't draw trivial / obvious relationships, such as drawing the class `Object` as a parent



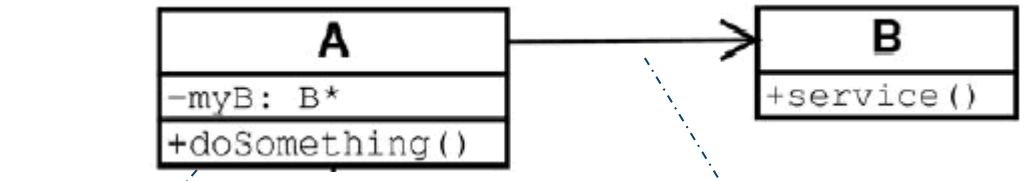
Association

navigability (direction)

multiplicity (how many are used)

- * ⇒ 0, 1, or more
- 1 ⇒ 1 exactly
- 2..4 ⇒ between 2 and 4, inclusive
- 3..* ⇒ 3 or more

name (stereotype, what relationship the objects have)



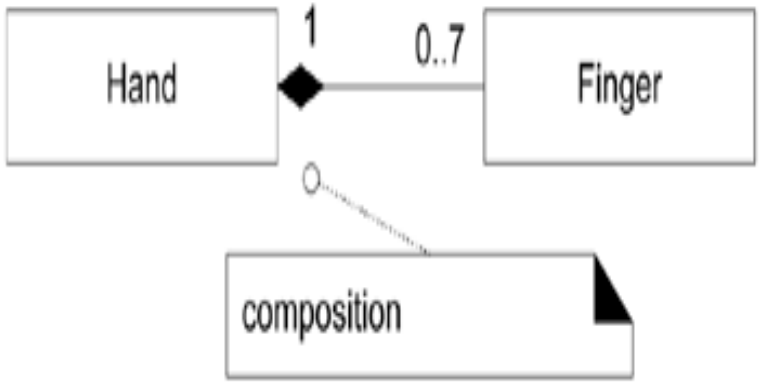

```
myB->service();
```

Arrow points in direction of the dependency

Composition

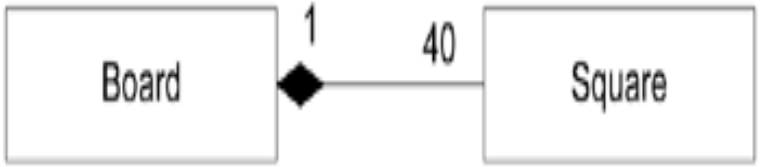
- A strong kind of ***whole-part aggregation***
- A composition relationship implies that
  - an instance of the part (such as a Square) belongs to only one composite instance (such as one Board) at a time,
  - the part must always belong to a composite (no free-floating Fingers), and
  - the composite is responsible for the creation and deletion of its parts either by itself creating/deleting the parts, or by collaborating with other objects.

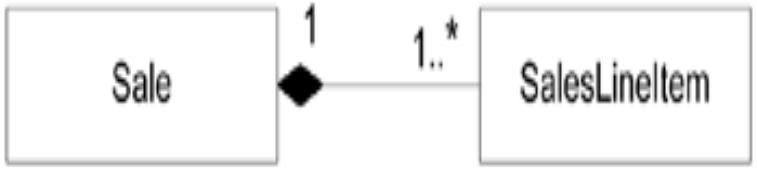
Composition



composition means
-a part instance (*Square*) can only be part of one composite (*Board*) at a time

-the composite has sole responsibility for management of its parts, especially creation and deletion

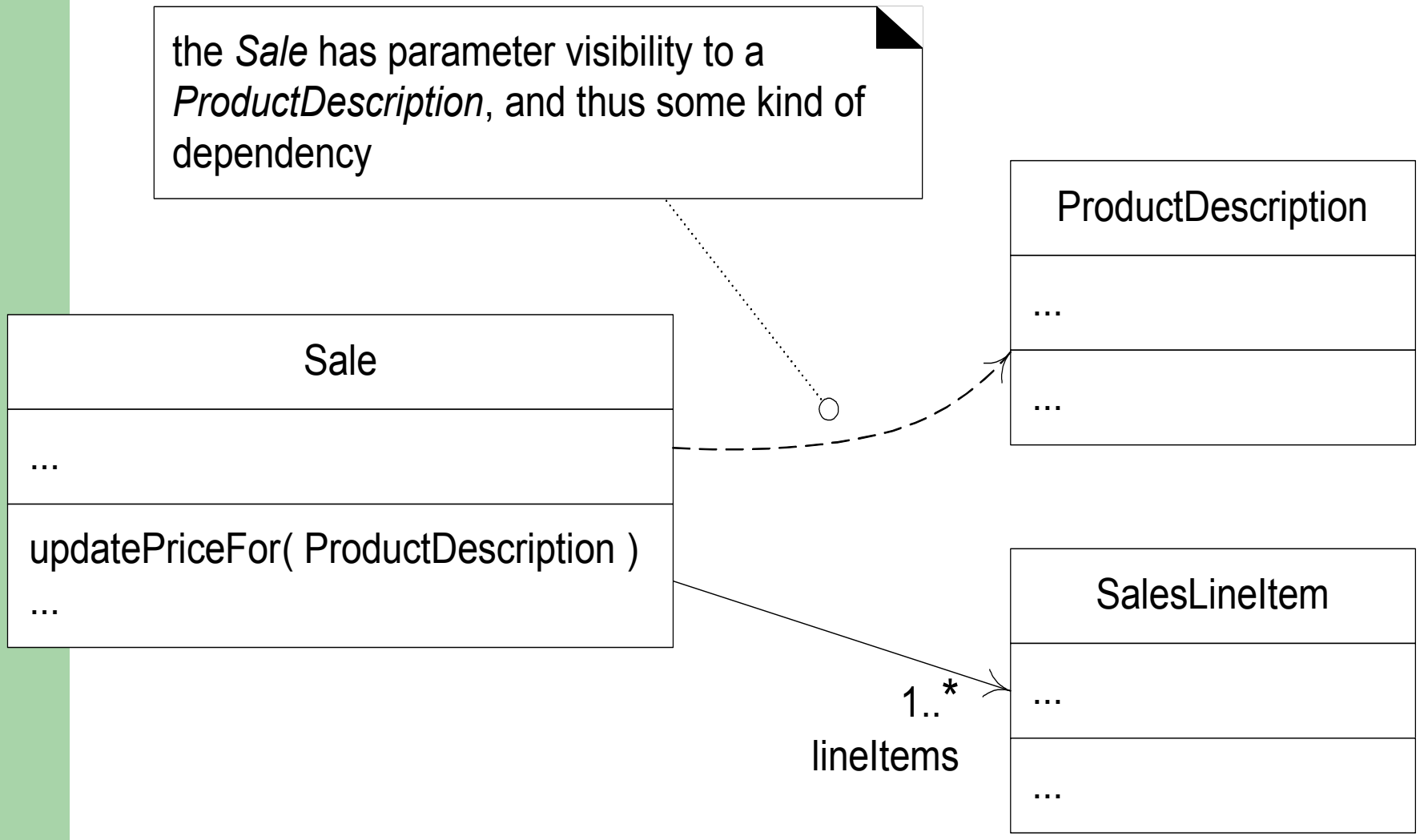




Dependency

- Dependency is illustrated with a dashed arrow line from the client to supplier.
- some common types in terms of objects and class diagrams :
  - having an attribute of the supplier type
  - sending a message to a supplier; the visibility to the supplier could be:
    - an attribute, a parameter variable, a local variable, a global variable, or class visibility (invoking static or class methods)
  - receiving a parameter of the supplier type
  - the supplier is a superclass or interface

Dependency


the *Sale* has parameter visibility to a *ProductDescription*, and thus some kind of dependency
ProductDescription
...
...
Sale
...
updatePriceFor(ProductDescription)
...
SalesLineItem
...
...
1..*
lineItems

UML Sequence Diagram

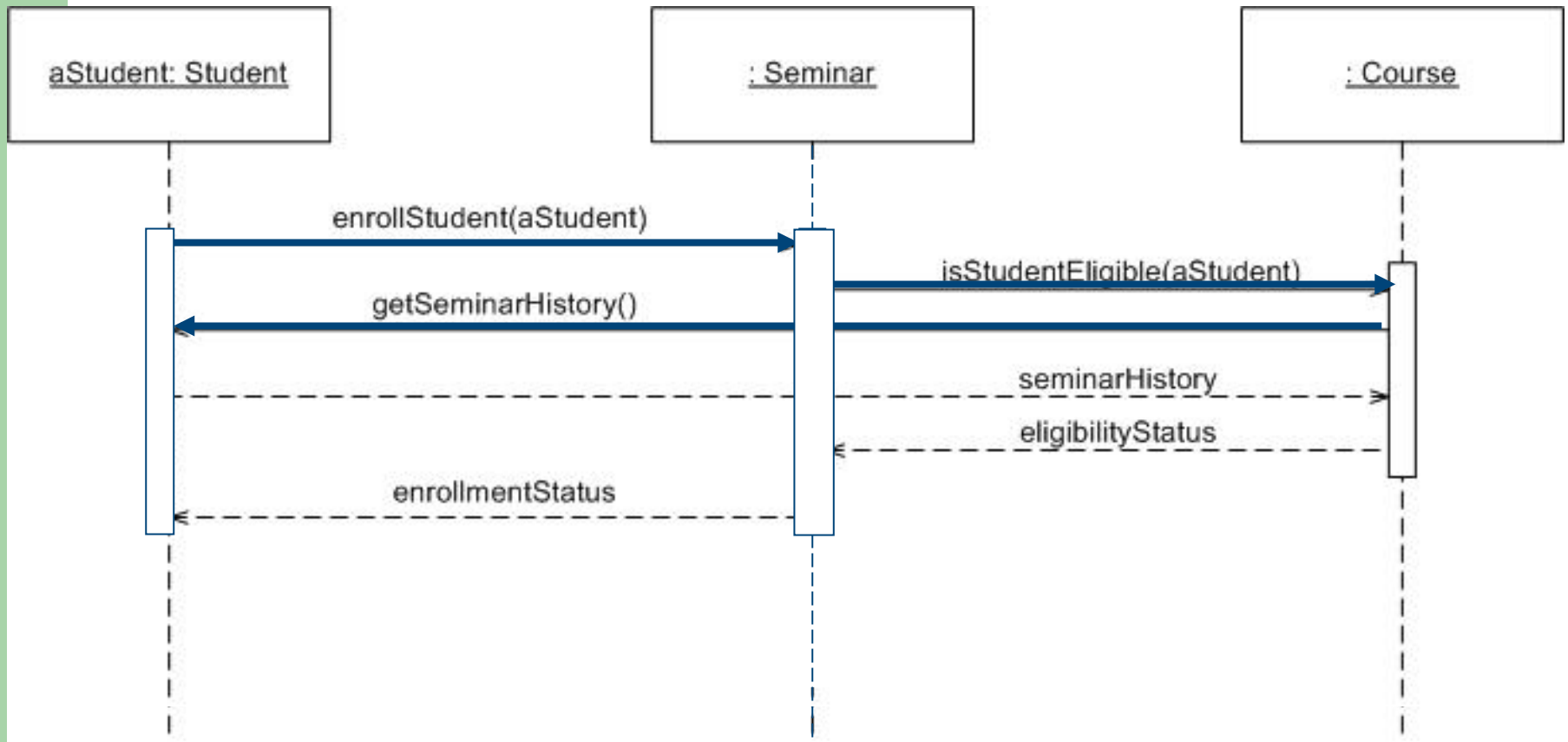
An "interaction diagram" that models a single scenario executing in the system

- Describes the flow of messages, events, actions between objects.
- Used during analysis and design to document and understand the logical flow of your system.
- Emphasis on time ordering!

UML Sequence Diagram

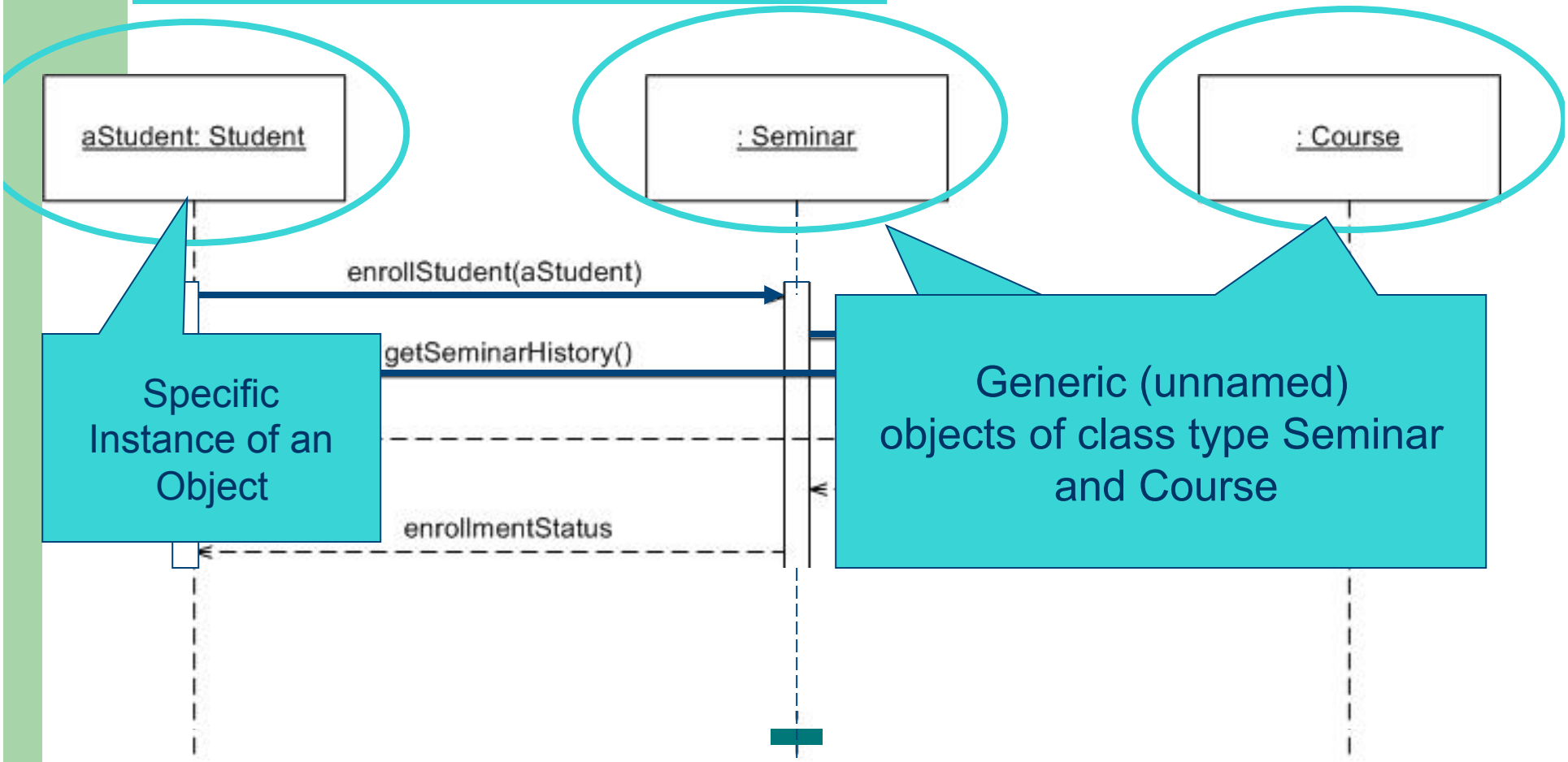
- **participant**: object or entity that acts in the diagram
  - diagram starts with an unattached "found message" arrow
- **message**: communication between participant objects
- the axes in a sequence diagram:
  - horizontal: which object/participant is acting
  - vertical: time (down -> forward in time)

UML Sequence Diagram

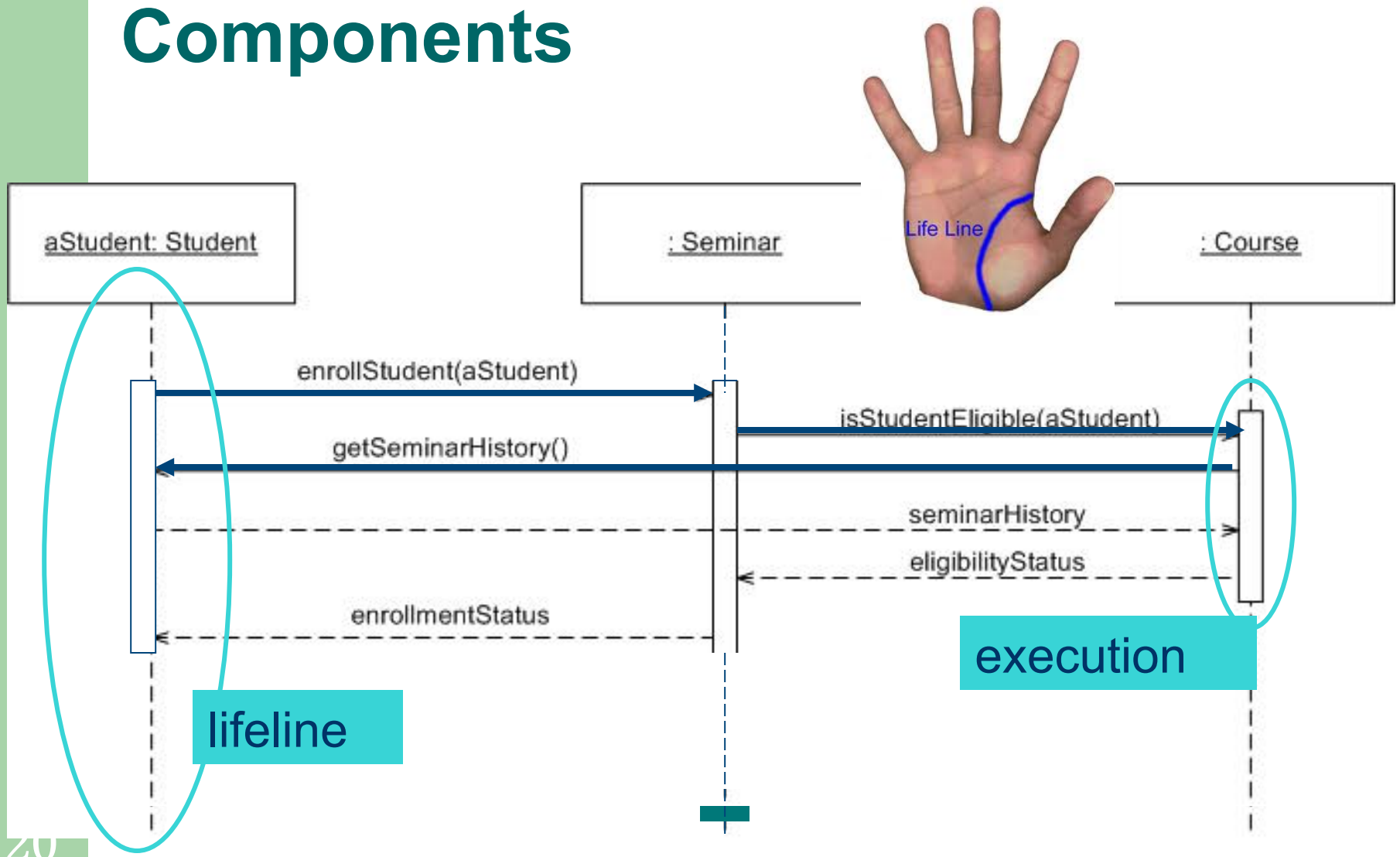


Components

Objects: aStudent is a specific instance of the Student class

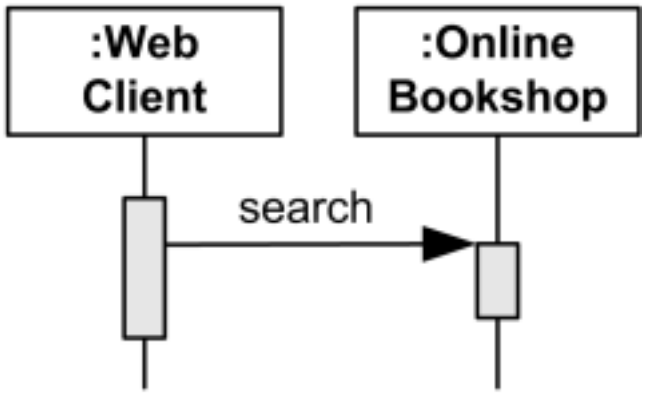


Components

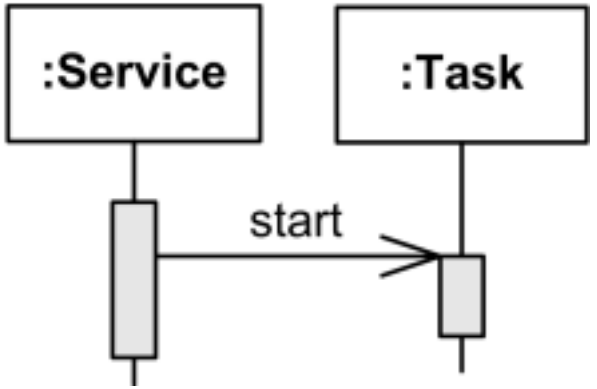

aStudent: Student
: Seminar
: Course
Life Line
enrollStudent(aStudent)
isStudentEligible(aStudent)
getSeminarHistory()
seminarHistory
eligibilityStatus
enrollmentStatus
execution
lifeline

Messages between objects

- messages (method calls) indicated by arrow to other object
  - write message name and arguments above arrow
  - Synchronized call : method call waits for response
  - Asynchronized call: without waiting for return value



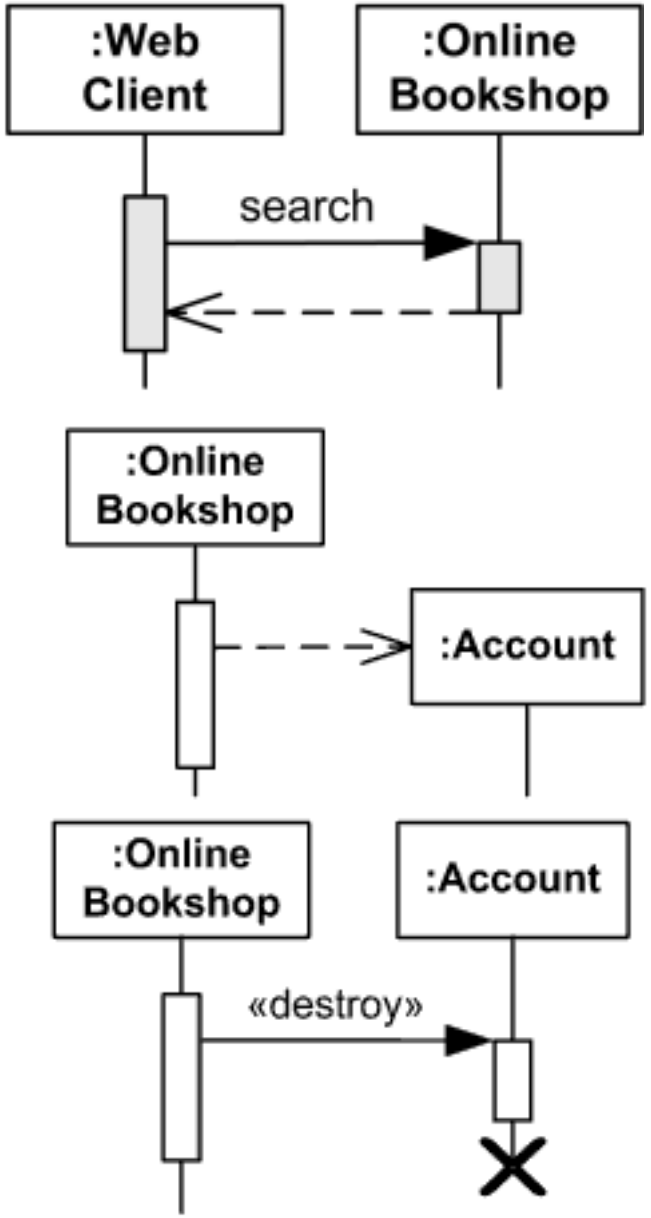
Synchronized call



Asynchronized call

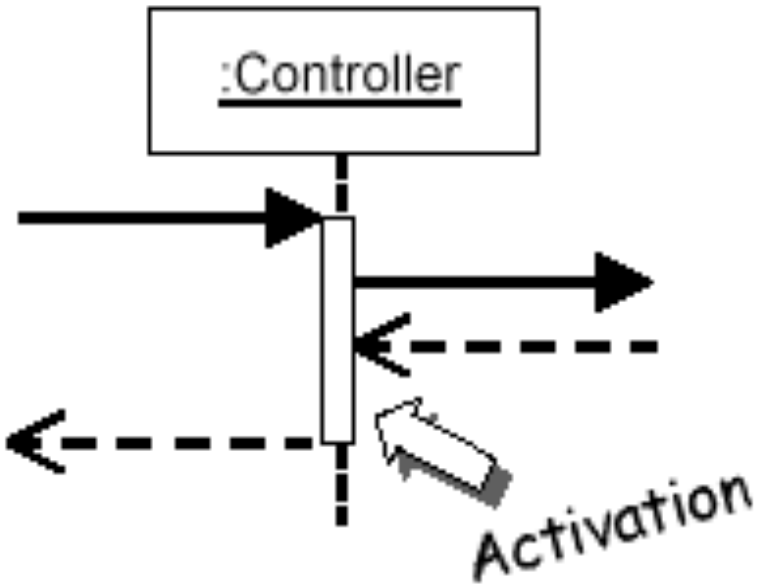
Messages between objects

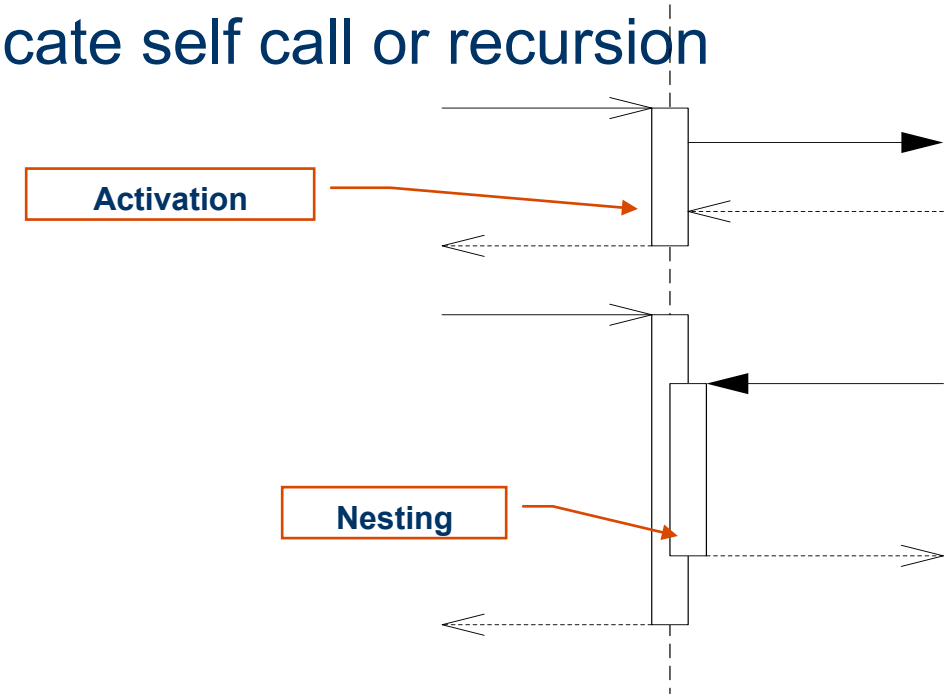
- Reply message:
- Create message: send to a life line to create itself.
- Delete message: send to terminate another lifeline



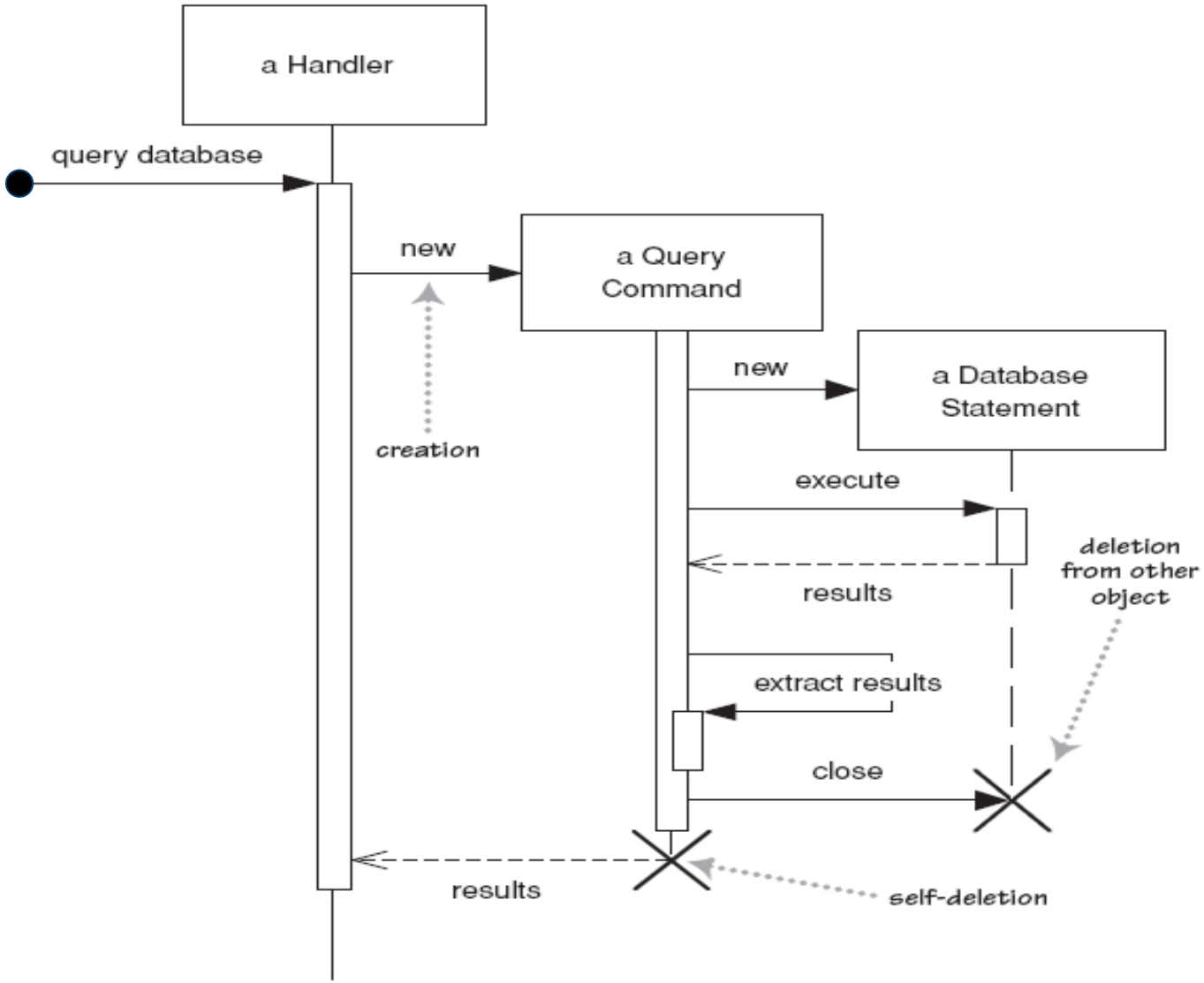
Indicating method calls

- **activation**: thick box over object's life line; drawn when object's method is on the stack
  - either that object is running its code, or it is on the stack waiting for another object's method to finish
  - nest activations to indicate self call or recursion



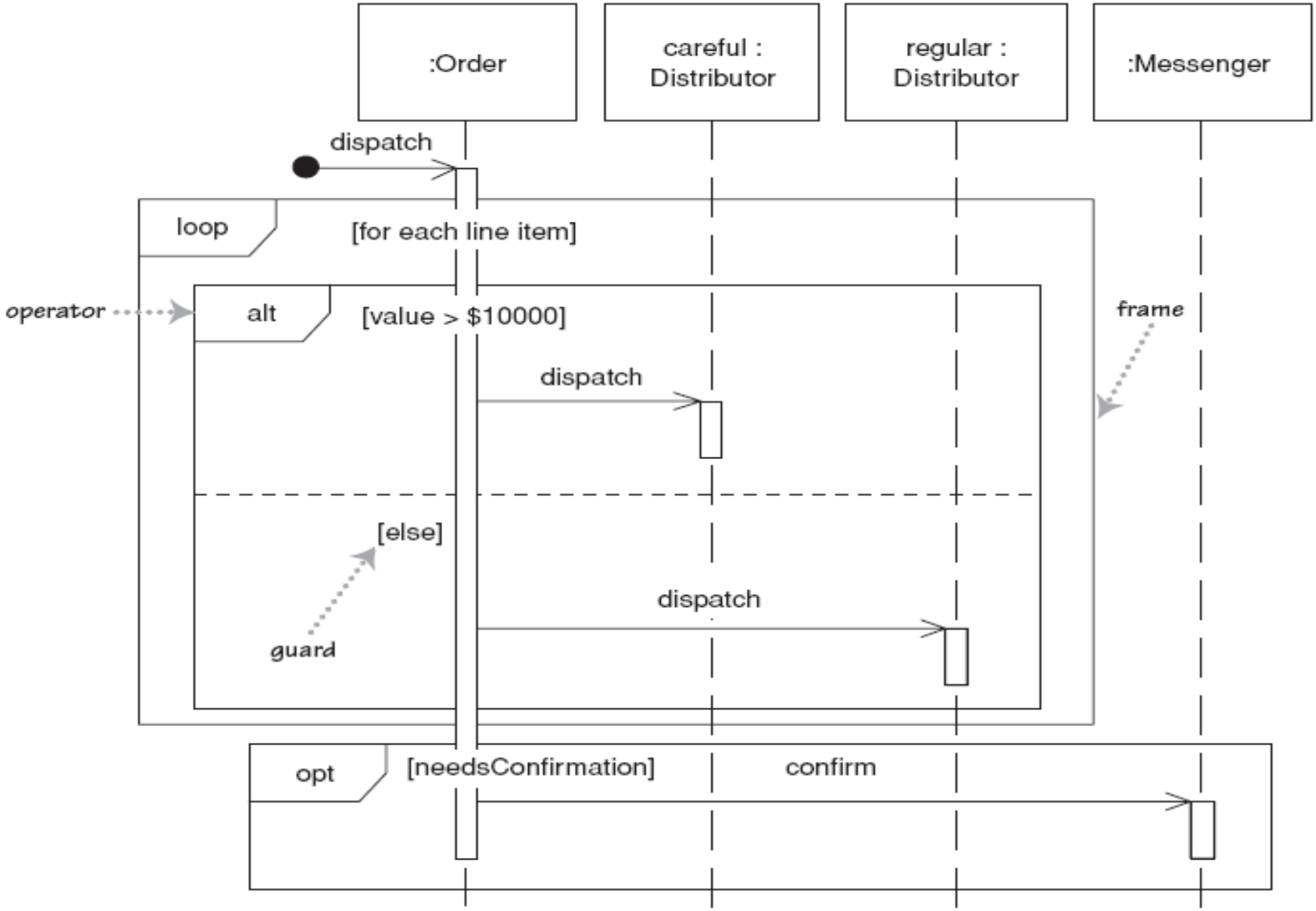


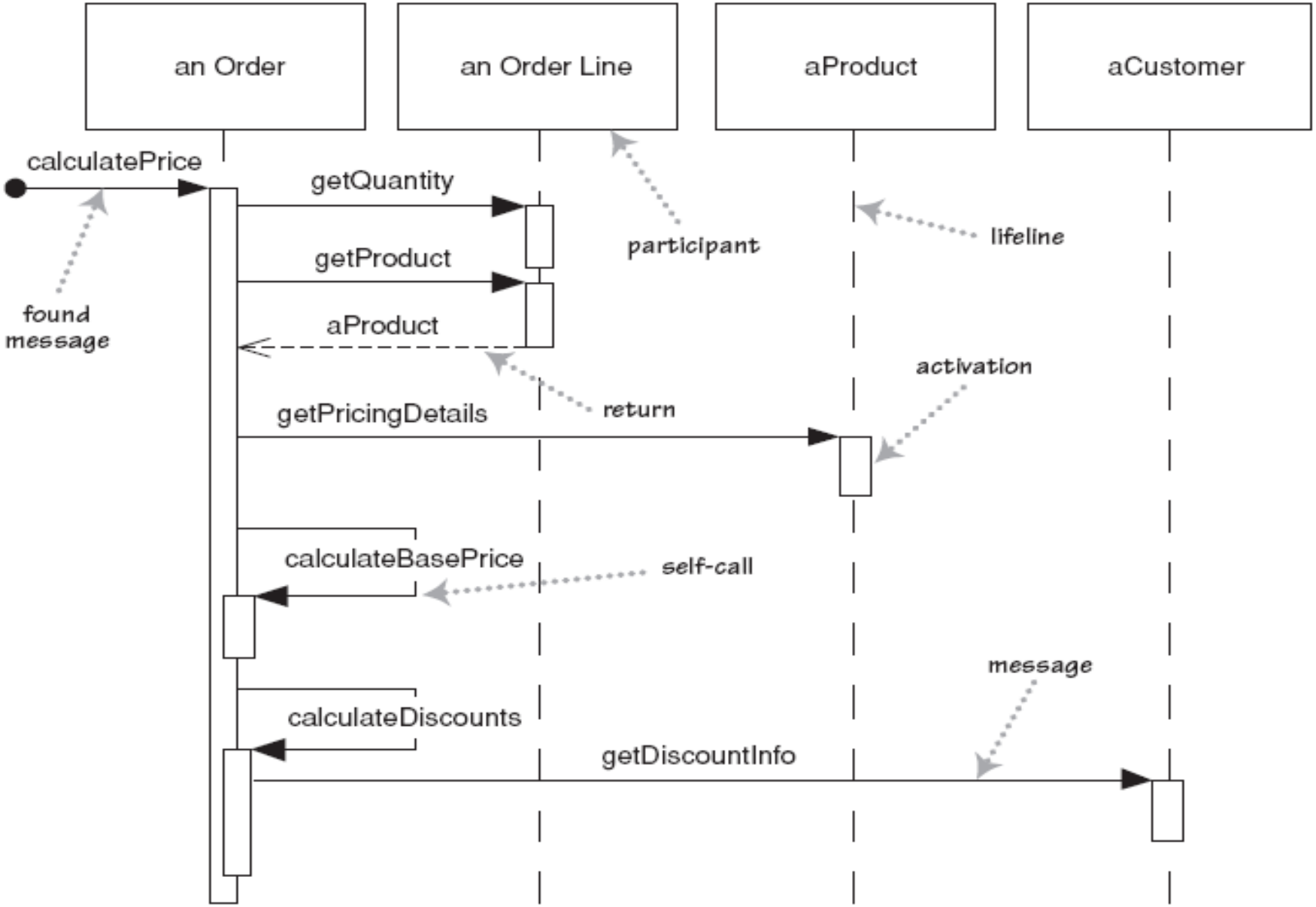
Lifetime of objects



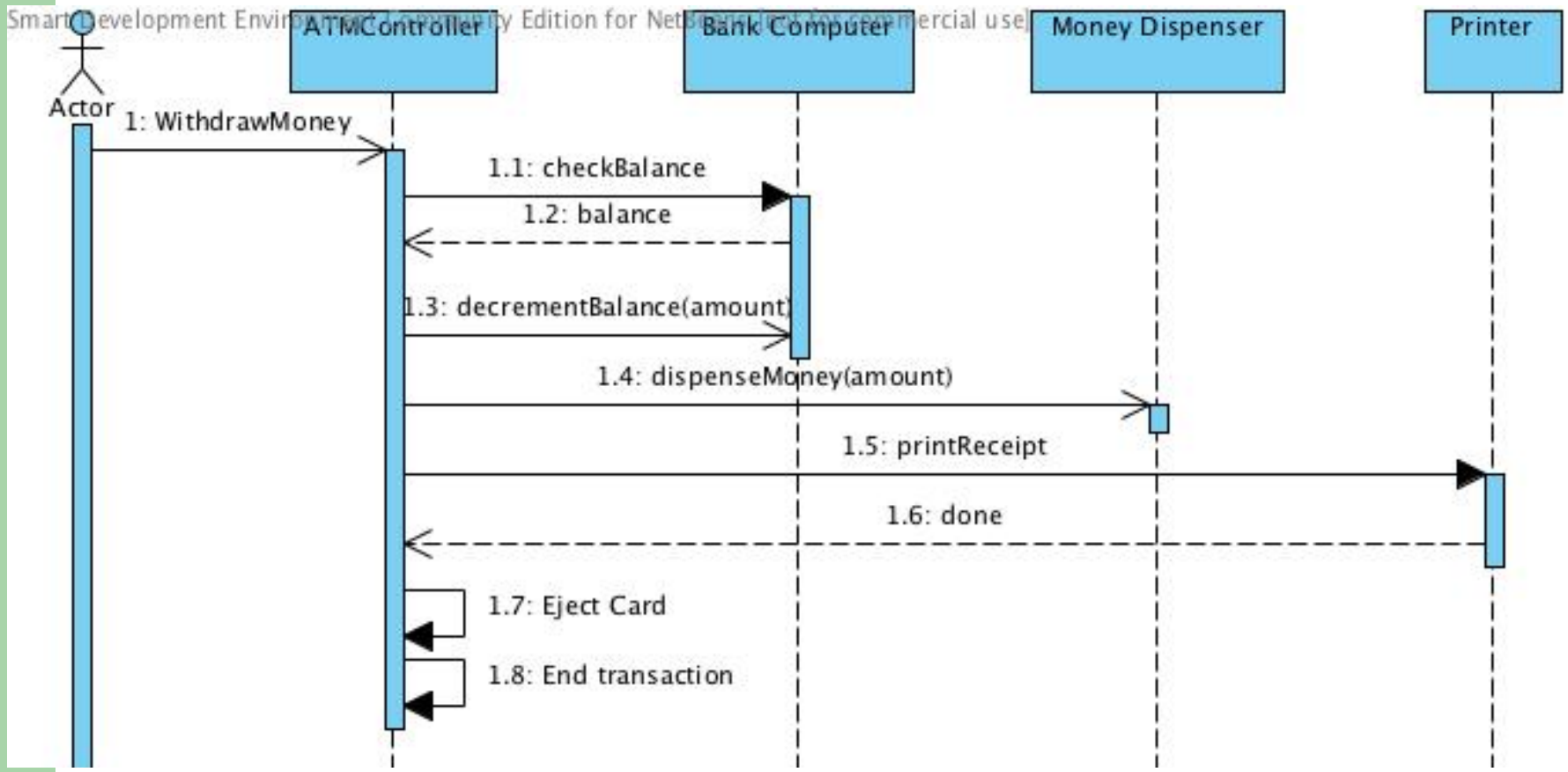
Note: older create and delete message notations are used here

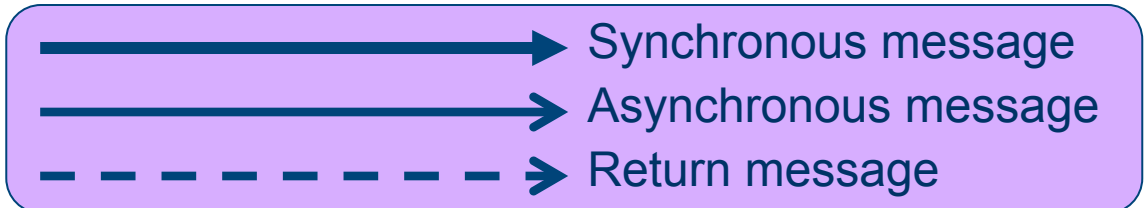
Selection and loops




an Order
an Order Line
aProduct
aCustomer
calculatePrice
getQuantity
getProduct
aProduct
getPricingDetails
calculateBasePrice
calculateDiscounts
getDiscountInfo
found message
participant
lifeline
return
activation
self-call
message

Exercise


Actor
ATMController
Bank Computer
Money Dispenser
Printer
1: WithdrawMoney
1.1: checkBalance
1.2: balance
1.3: decrementBalance(amount)
1.4: dispenseMoney(amount)
1.5: printReceipt
1.6: done
1.7: Eject Card
1.8: End transaction


Synchronous message
Asynchronous message
Return message

Why not just code it?

Sequence diagrams can be close to the code level. So why not just code up that algorithm rather than drawing it as a sequence diagram?

- a good sequence diagram is still a bit above the level of the real code (not all code is drawn on diagram)
- sequence diagrams are language-agnostic
- non-coders can do sequence diagrams
- easier to do sequence diagrams as a team
- can see many objects/classes at a time on same page (visual bandwidth)

The goal of UML is communication and understanding

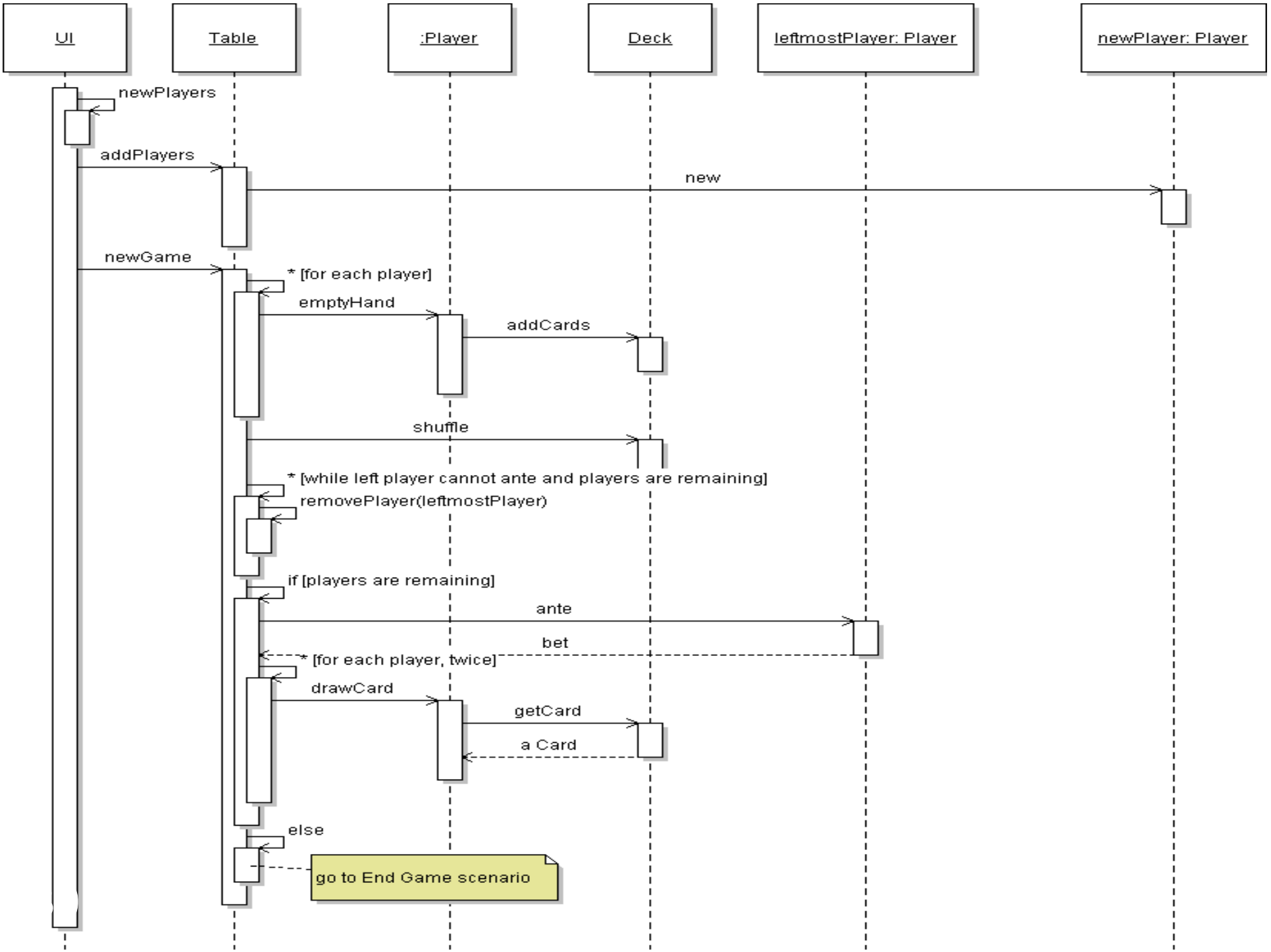
Sequence diagram exercise

- Write a sequence diagram for the following poker casual use case, *Start New Game Round*

The scenario begins when the player chooses to start a new round in the UI. The UI asks whether any new players want to join the round; if so, the new players are added using the UI.

All players' hands are emptied into the deck, which is then shuffled. The player left of the dealer supplies an ante bet of the proper amount. Next each player is dealt a hand of two cards from the deck in a round-robin fashion; one card to each player, then the second card.

If the player left of the dealer doesn't have enough money to ante, he/she is removed from the game, and the next player supplies the ante. If that player also cannot afford the ante, this cycle continues until such a player is found or all players are removed.


UI
Table
:Player
Deck
leftmostPlayer: Player
newPlayer: Player
newPlayers
addPlayers
new
newGame
* [for each player]
emptyHand
addCards
shuffle
* [while left player cannot ante and players are remaining]
removePlayer(leftmostPlayer)
if [players are remaining]
ante
bet
* [for each player, twice]
drawCard
getCard
a Card
else
go to End Game scenario