ΗΥ 351: Ανάλυση και Σχεδίαση Πληροφοριακών Συστημάτων
CS 351: Information Systems Analysis and Design

Behavioral Modeling

…

Lecture : 11
Date : 15-11-2005

Yannis Tzitzikas
University of Crete, Fall 2005

Outline

- *What is Behavioral Modeling?*
- *Interaction Diagrams*
  - Sequence Diagrams
  - Communication Diagrams
- State diagrams

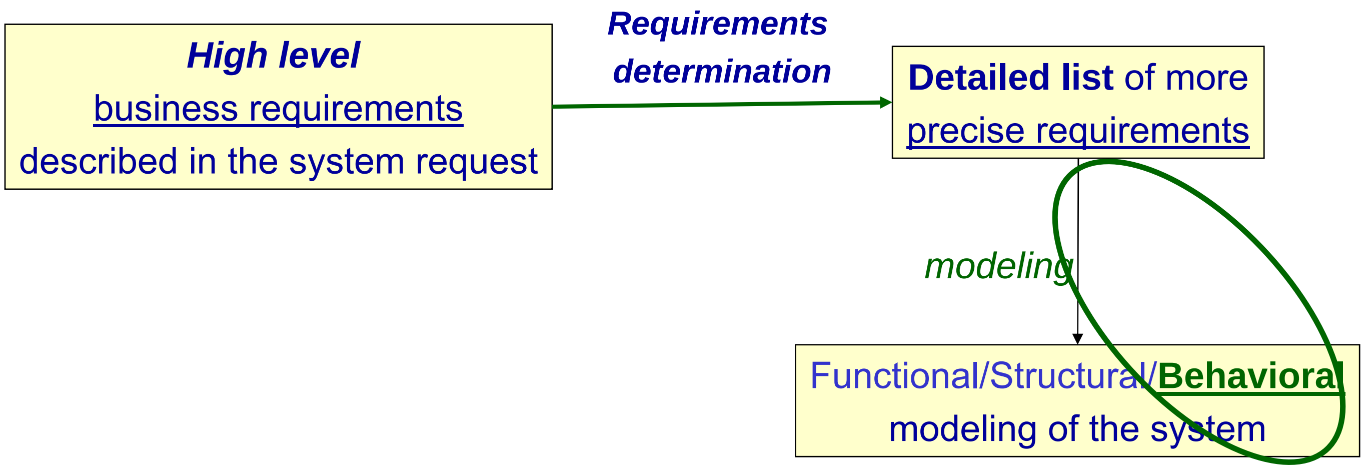
What is Behavioral Modeling?

Its objective is to describe:

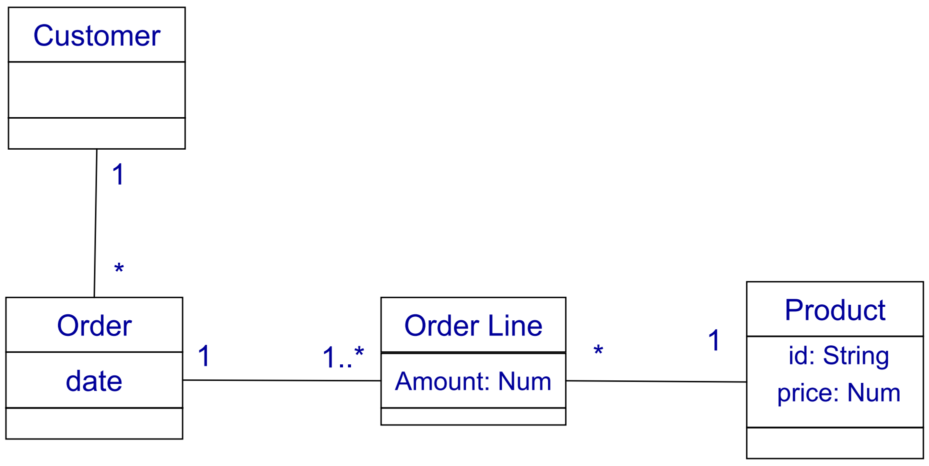
- internal dynamic aspects of an information system that supports business processes in an organization

Why to do Behavioral Modeling ?

- To depict the internal view of business processes
- To show the messages that pass between objects for a particular use-case


High level business requirements described in the system request
Requirements determination
Detailed list of more precise requirements
modeling
Functional/Structural/**Behavioral** modeling of the system

How the objects of this model interact ?



E.g. how the price of an order is calculated ?

How we model the behavior in OO Analysis and Design?

Usually we employ 3 types of models:

- ***Sequence diagrams***
- ***Communication diagrams***
  - **(in UML 1. they were called "Collaboration Diagrams")**
- ***Statechart diagrams***

} *Interaction diagrams*

Remarks:

- Modeling the behaviour in detail is like … implementing the system!
- So we should model ***the key aspects***
  - like storyboarding in film making (I.e. key frames)

Interaction Diagrams

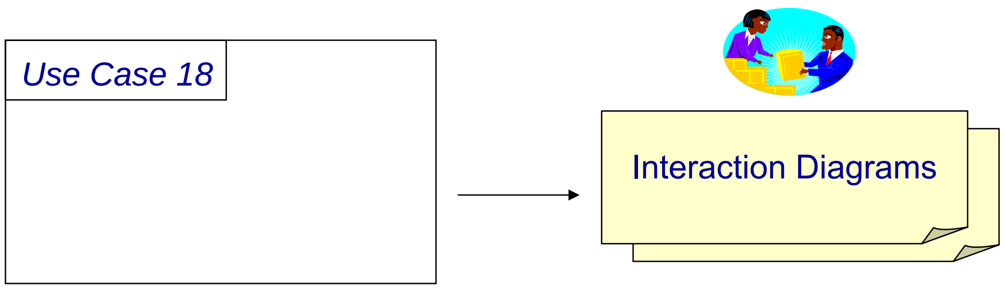
Interaction Diagrams (Διαγράμματα Αλληλεπίδρασης)

- **Sequence Diagrams** (Διαγράμματα Ακολουθίας)
- **Communication/Collaboration Diagrams** (Διαγράμματα Συνεργασίας)

Interaction Diagrams

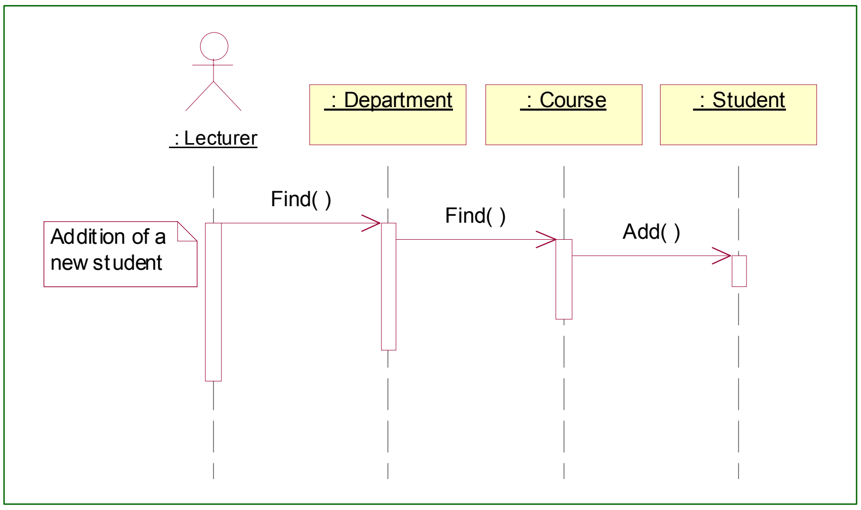
- They describe how groups of objects collaborate in some behaviour.

- Typically, an Interaction Diagram captures the behaviour of a single Use Case and shows a number of example objects and the messages that are passed between these objects within the use case



Sequence Diagrams

[A] Sequence Diagrams
(διαγράμματα ακολουθίας/διαδοχής/αλληλουχίας)



- *Horizontal line*: objects shown as boxes
- *Vertical line*: object's lifeline
- *Activation box*: shows when object is active (at the stack)
- *messages*: between the lifelines of 2 objects

Messages

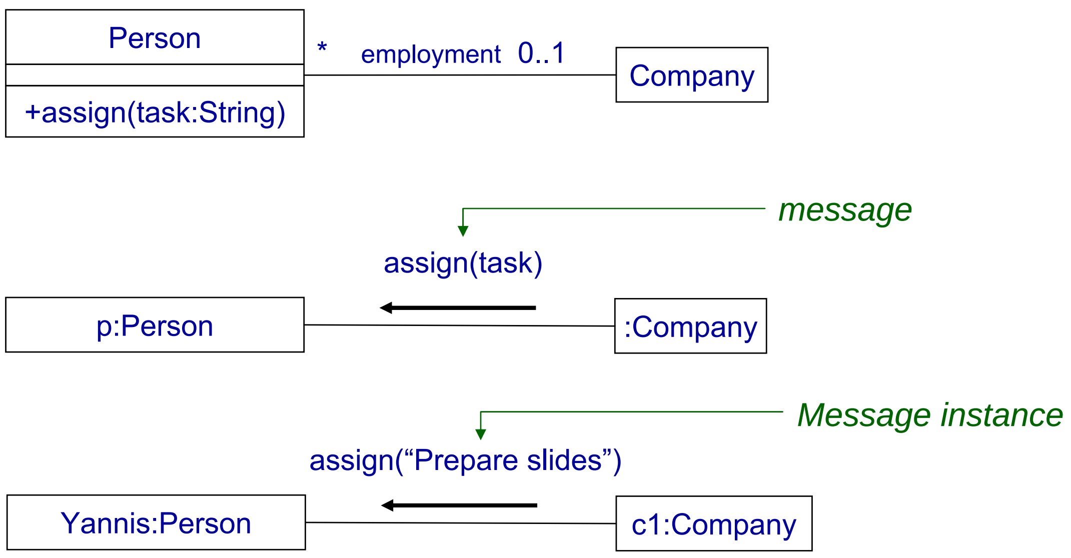
- A message is a specification of a communication between objects
- Types of messages
  - ***Call***: Invocation of an operation
    - an object can also send a message to itself (local invocation of an operation)
  - ***Return***: returns a value to the caller
  - ***Send***: sends a signal to an object
  - ***Create***: creates an object
  - ***Destroy***: destroys an object

- A signal is an object value communicated to a target object asynchronously.
- After sending a signal, the sending object continues its own execution.
- When the target object receives the signal message,it independently decides what to do about it.

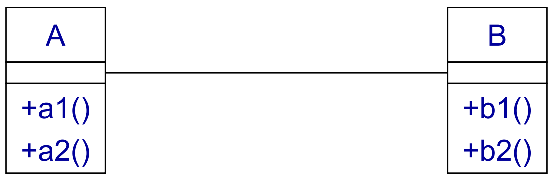
How we depict messages?

- As an arrow between the lifelines of 2 objects
- The arrow is accompanied by
  - message name (e.g. name of called operation)
  - possible arguments
  - control info
    - <u>condition</u>: indicates when a message is sent, e.g. [outOfStock]
    - <u>iteration marker</u>: indicates a message sent many times to multiple receiver objects, e.g. *[for all order lines] // for UML 1.
- Return messages are denoted by dashed line (<- -)
  - we can omit it and not draw every return message but only the crucial

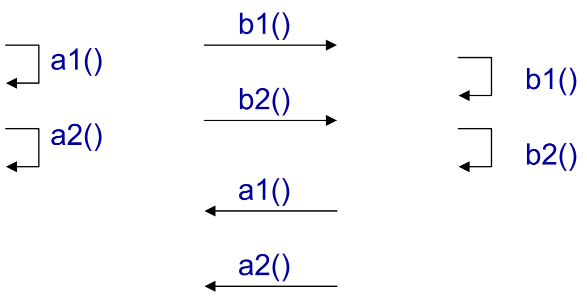
Examples of Messages



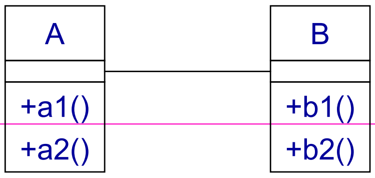
Example of Messages



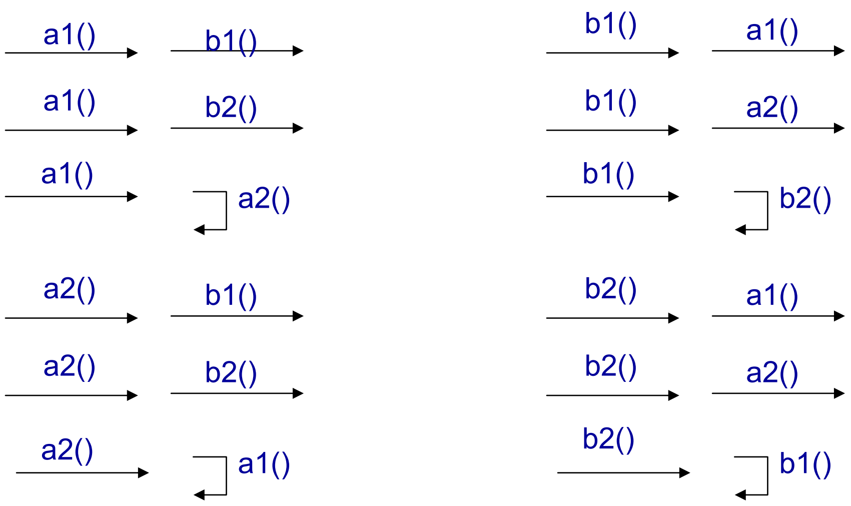
All possible messages:



Example

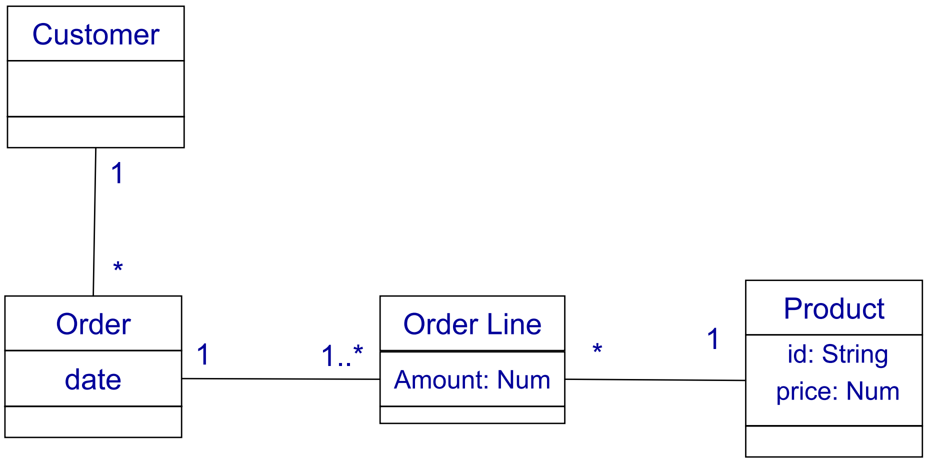


All possible flows of control that consist of 2 messages:



All possible flows of control that consist of 3 messages:

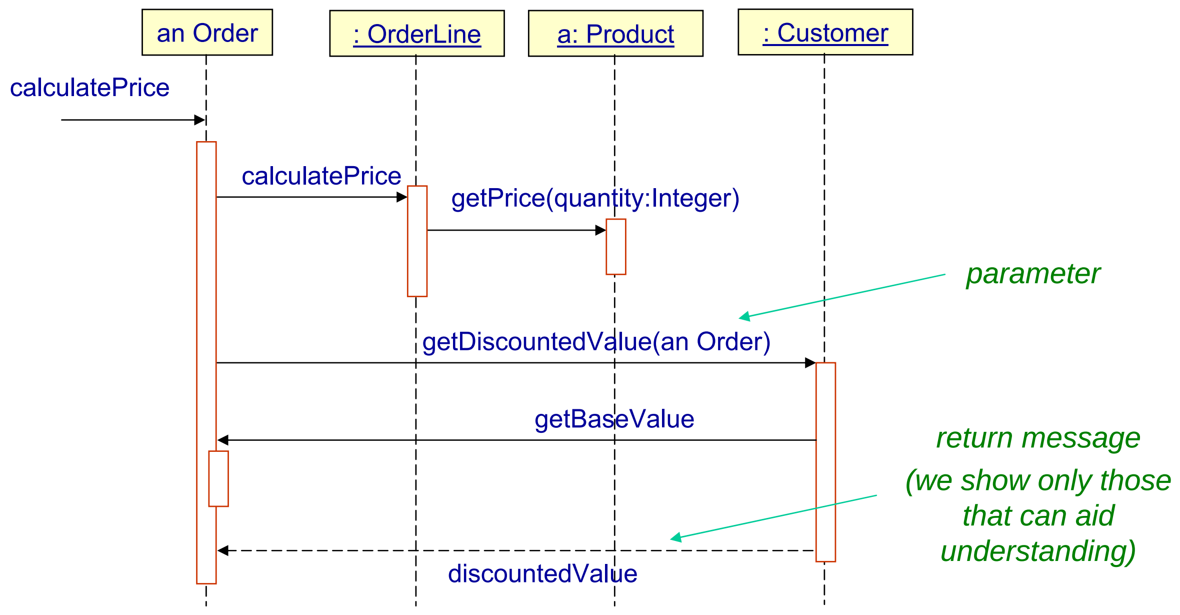
How the objects of this model interact ?



E.g. how the price of an order is calculated ?

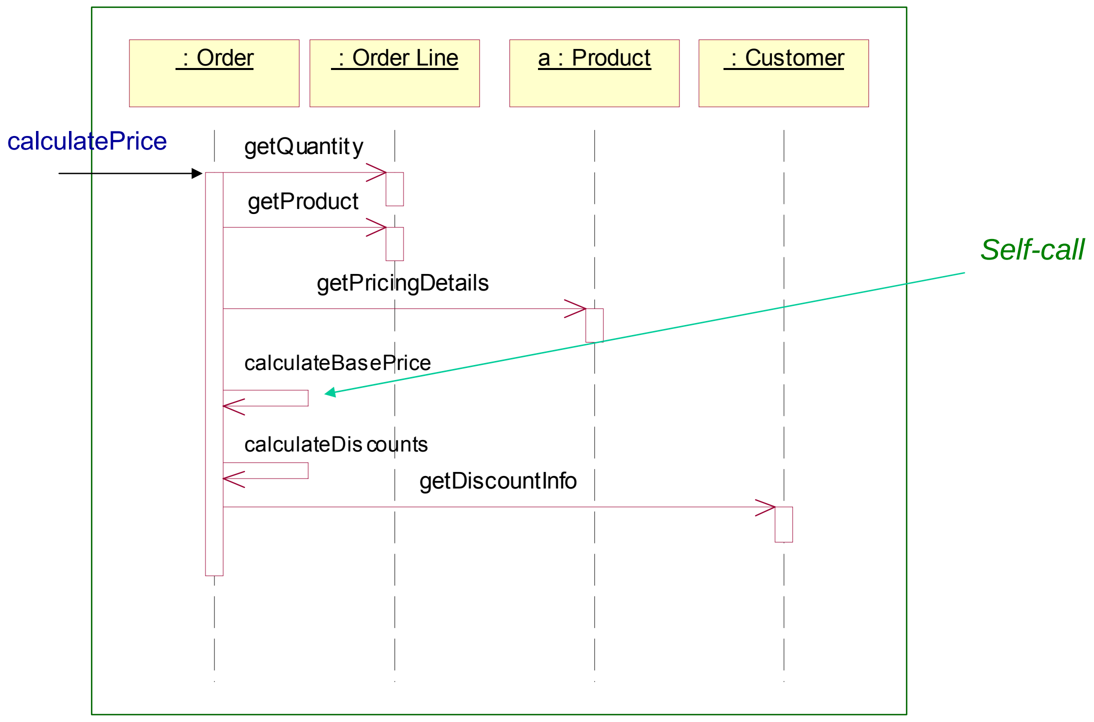
Example of a sequence diagram

Calculation of the price of an order line of an order

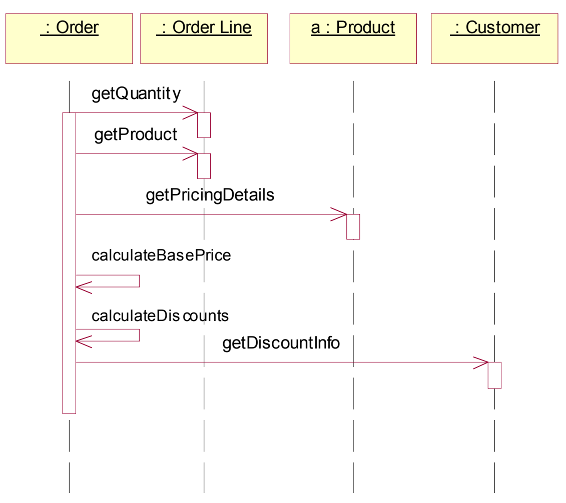


Sequence diagram of a different implementation

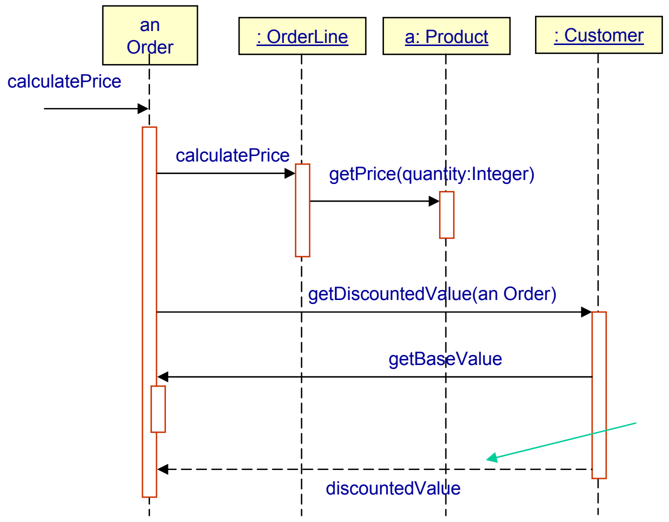
Calculation of the price of an order line of an order



Comparing the two diagrams



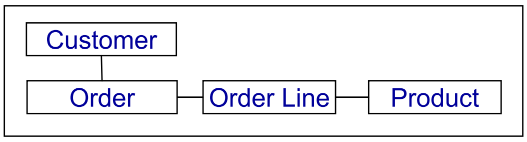
centralized control

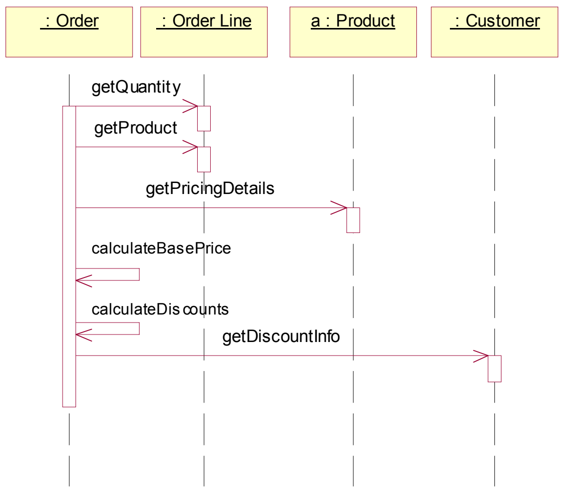


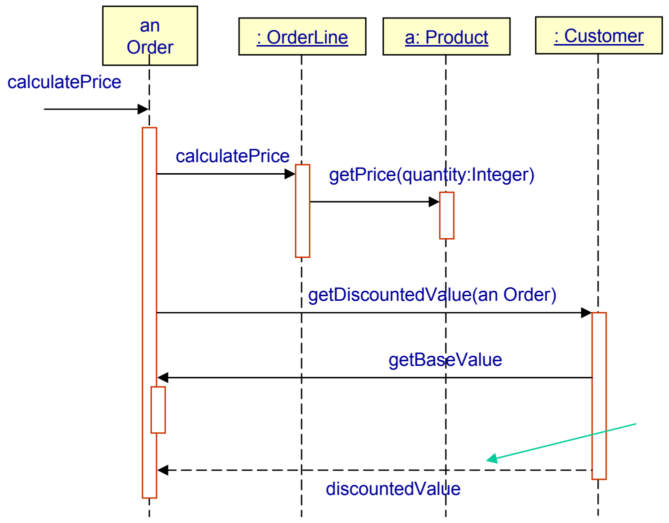
distributed control

Sequence diagrams are not very good at showing details (algs with loops and conditions), but they make the calls between participants very clear and give a good picture about which participants are doing which processing.

Comparing the two diagrams (II)



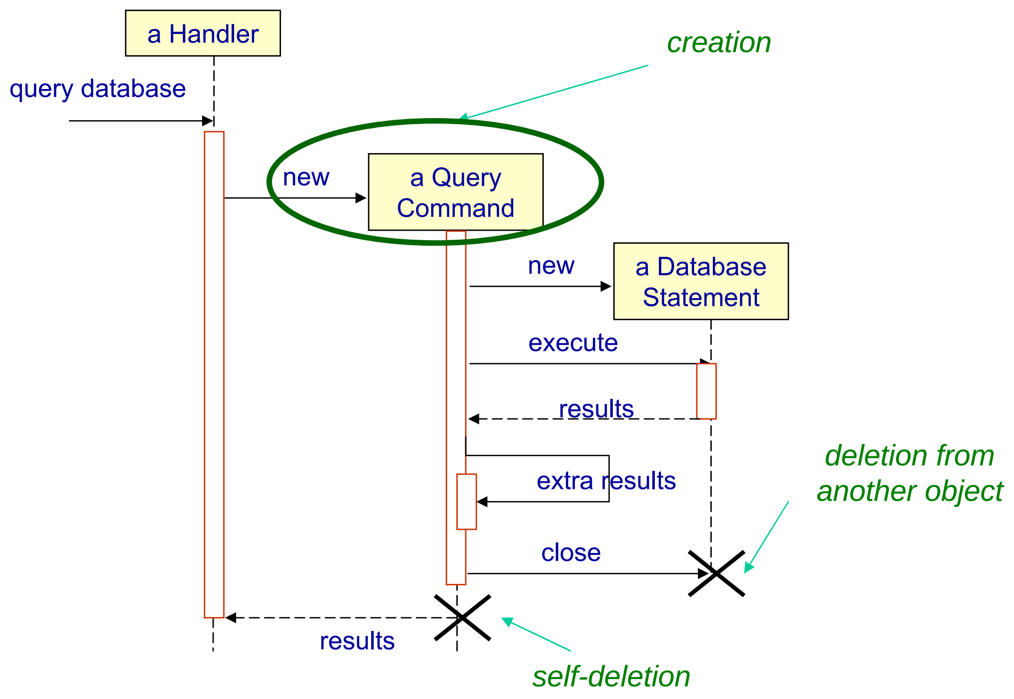




Here an Order communnicates with a Product (although they are not associated in the class diagram)

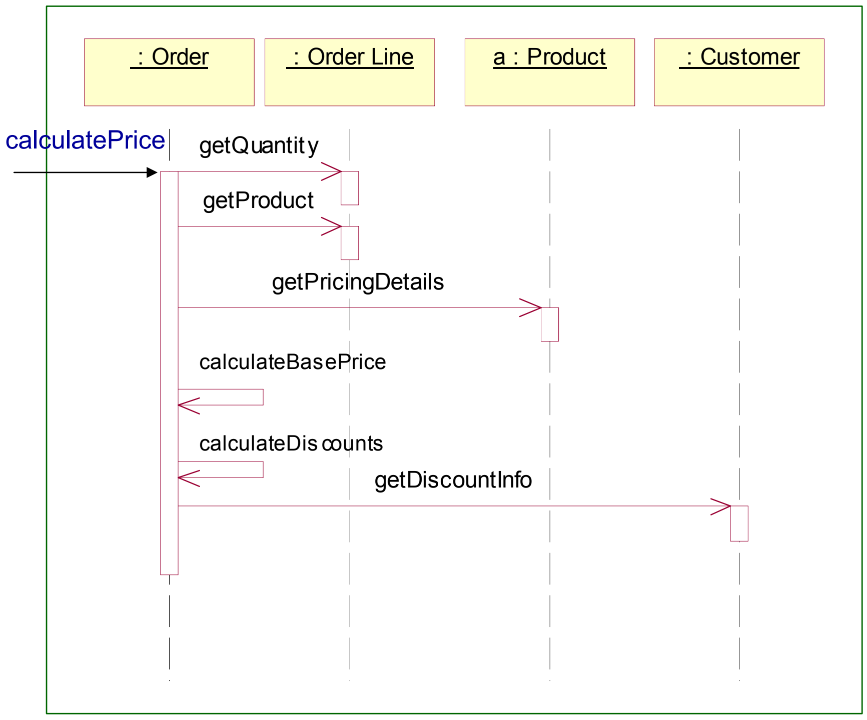
Here only those objects that are associated communicate

Creating and Deleting Participants



In a garbage-collected environment we don’t delete objects directly, but it is still worth using X to know when an object is no longer available and can be deleted

What about loops?



From this diagram it is not clear that the above calls should be done for every OrderLine of an Order

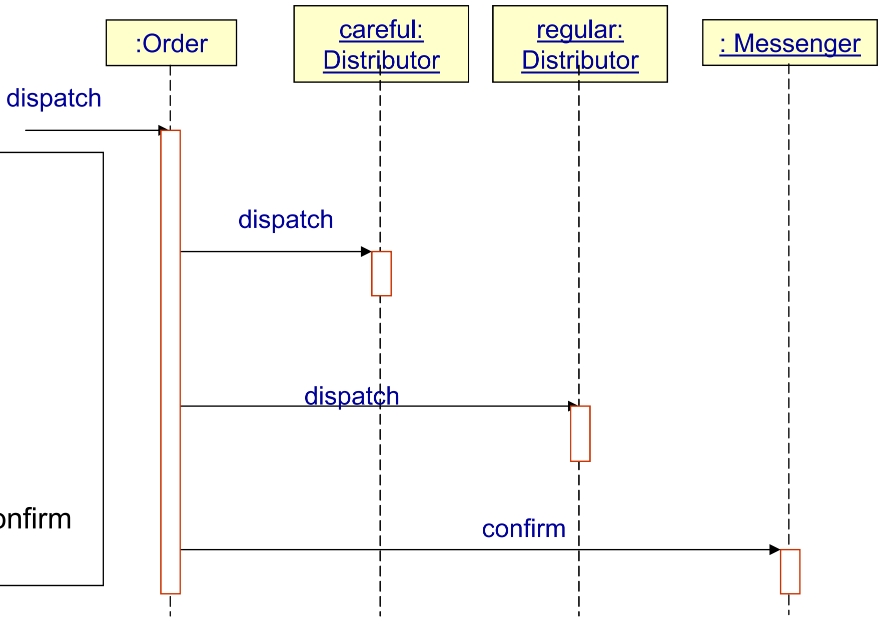
Loops and Conditionals (modeling control logic)

This is not the focus of Sequence Diagrams.
We could use Activity Diagrams or Pseudo-code instead.

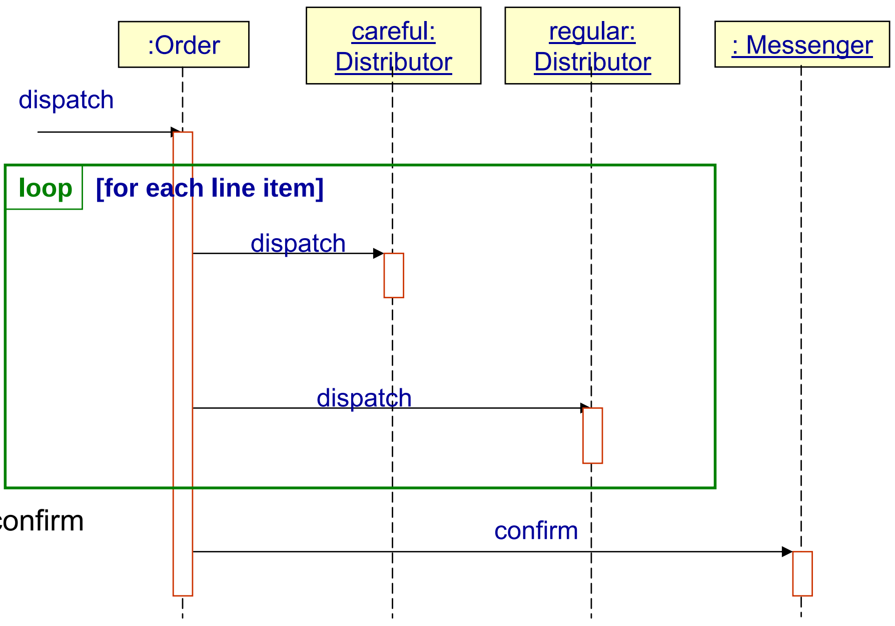
```
procedure dispatch
  foreach (lineitem)
      if (product.value > $10K)
          careful.dispatch
      else
          regular.dispatch
      endif
    endfor
  if (needsConfirmation) messenger.confirm
end
```

Loops and Conditionals (modeling control logic)

```
procedure dispatch
  foreach (lineitem)
      if (product.value > $10K)
          careful.dispatch
      else
          regular.dispatch
      endif
    endfor
  if (needsConfirmation) messenger.confirm
end
```

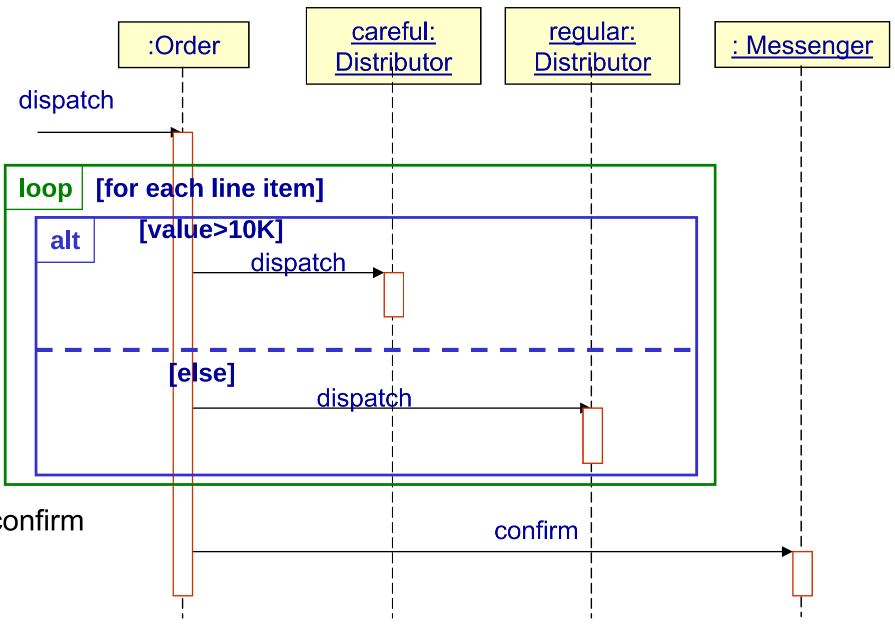


Loops and Conditionals (modeling control logic)



```
procedure dispatch
  foreach (lineitem)
      if (product.value > $10K)
           careful.dispatch
      else
           regular.dispatch
      endif
   endfor
  if (needsConfirmation) messenger.confirm
end
```

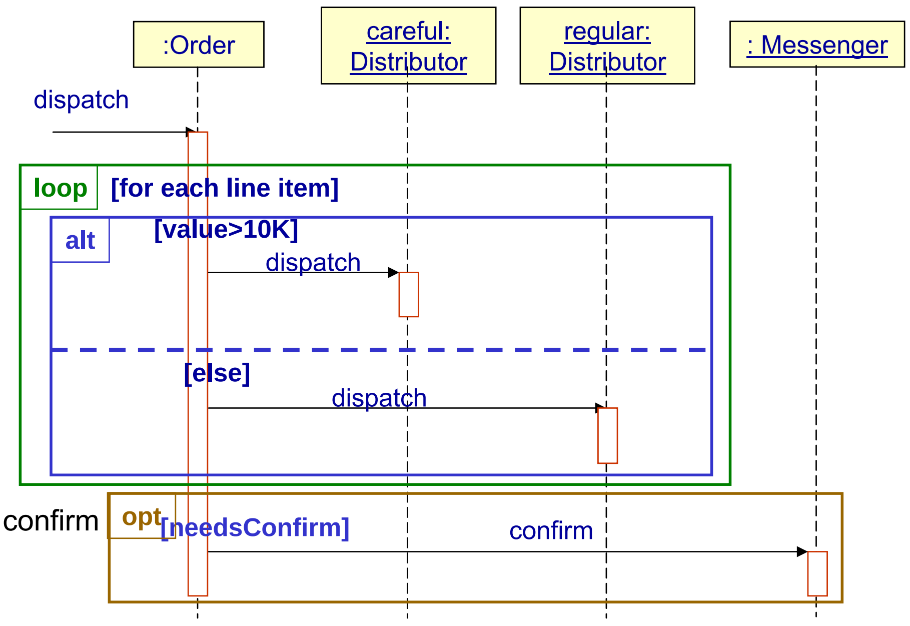
Loops and Conditionals (modeling control logic)



```
procedure dispatch
  foreach (lineitem(
      if (product.value > $10K)
           careful.dispatch
      else
           regular.dispatch
      endif
   endfor
  if (needsConfirmation) messenger.confirm
end
```

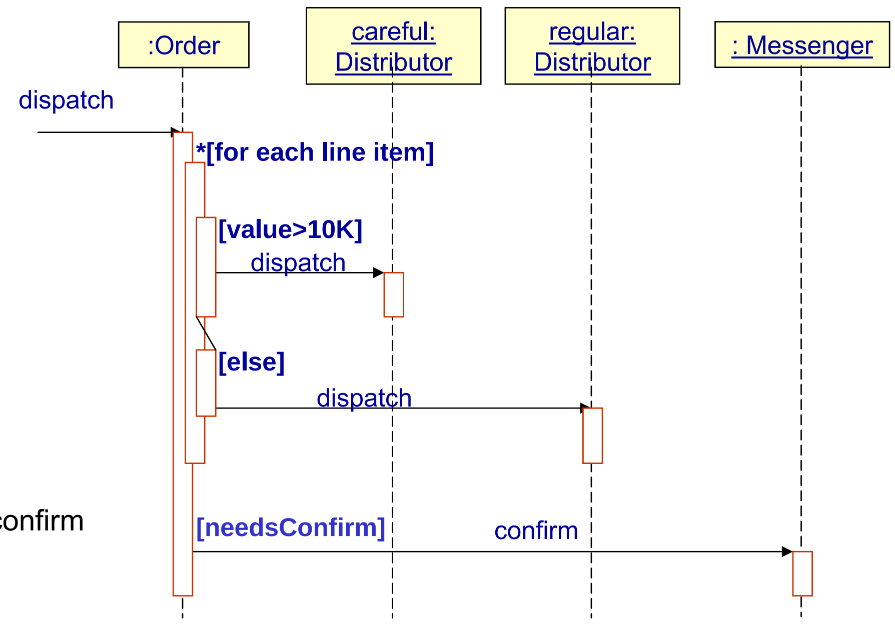
Loops and Conditionals (modeling control logic)

```
procedure dispatch
  foreach (lineitem(
      if (product.value > $10K)
           careful.dispatch
      else
           regular.dispatch
      endif
    endfor
  if (needsConfirmation) messenger.confirm
end
```



Loops and Conditionals (notations of UML 1)

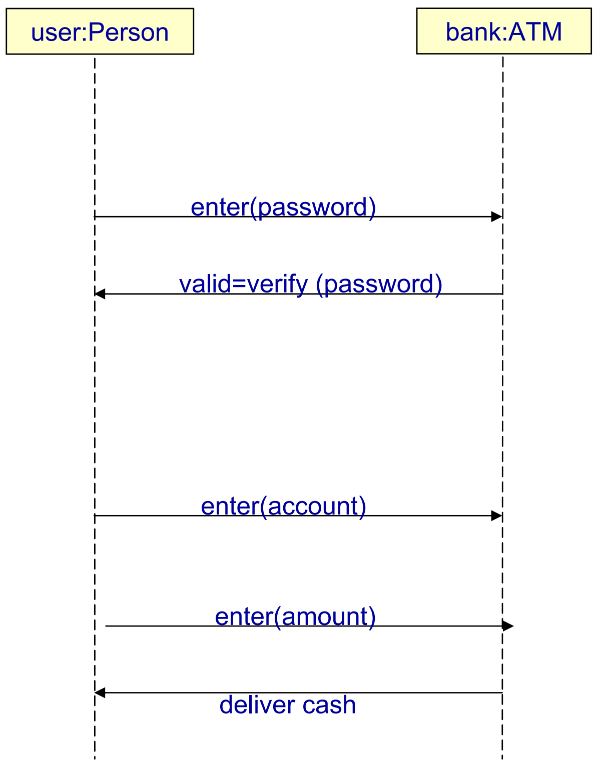
```
procedure dispatch
  foreach (lineitem(
      if (product.value > $10K)
           careful.dispatch
      else
           regular.dispatch
      endif
    endfor
  if (needsConfirmation) messenger.confirm
end
```



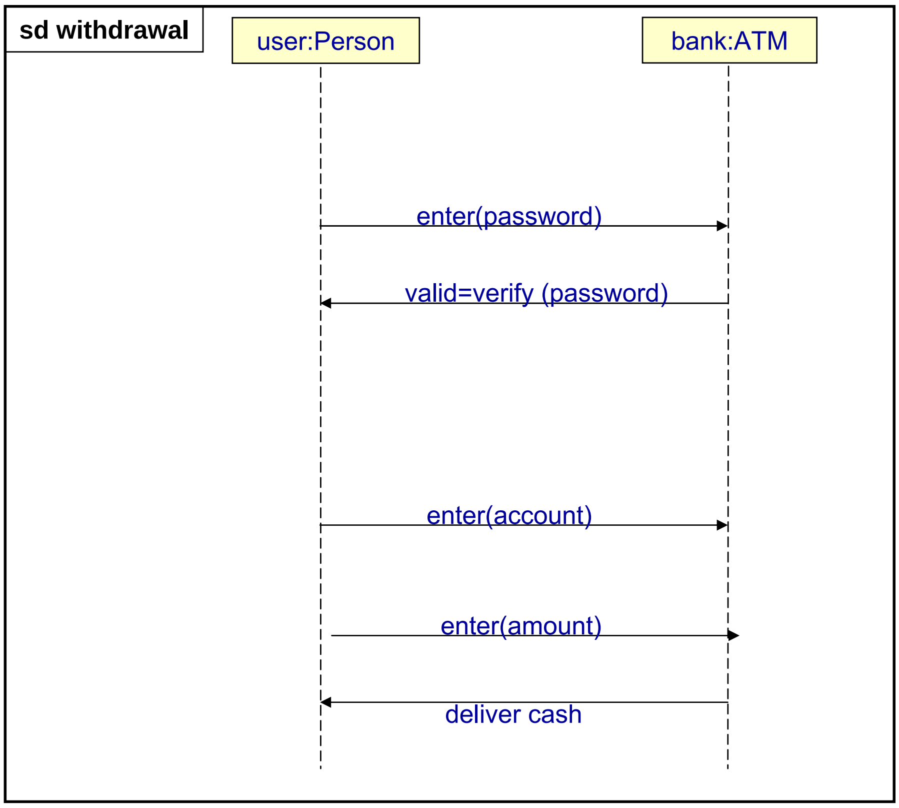
Operators for sequence diagrams

- **alt**: alternative multiple fragments; only the one whose condition is true will be executed
- **opt**: optional fragments; executed only if its condition is true (equiv to alt with one fragment)
- **par**: parallel execution of fragments
- **loop**: the fragments will be executed multiple times (based on the guard)
- **region**: critical region; the fragment can have only one thread executing it at once
- **neg**: the fragment shows an invalid interaction
- **ref**: reference: refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return a value.
- **sd**: sequence diagram; used to surround the entire diagram

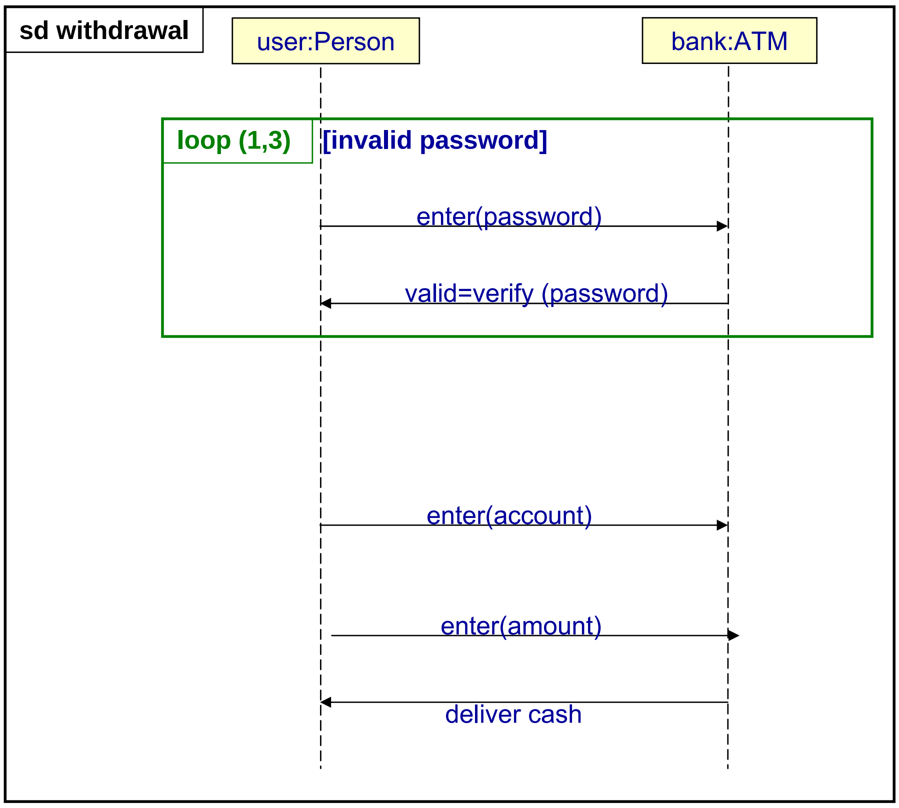
Withdraw cash from an ATM



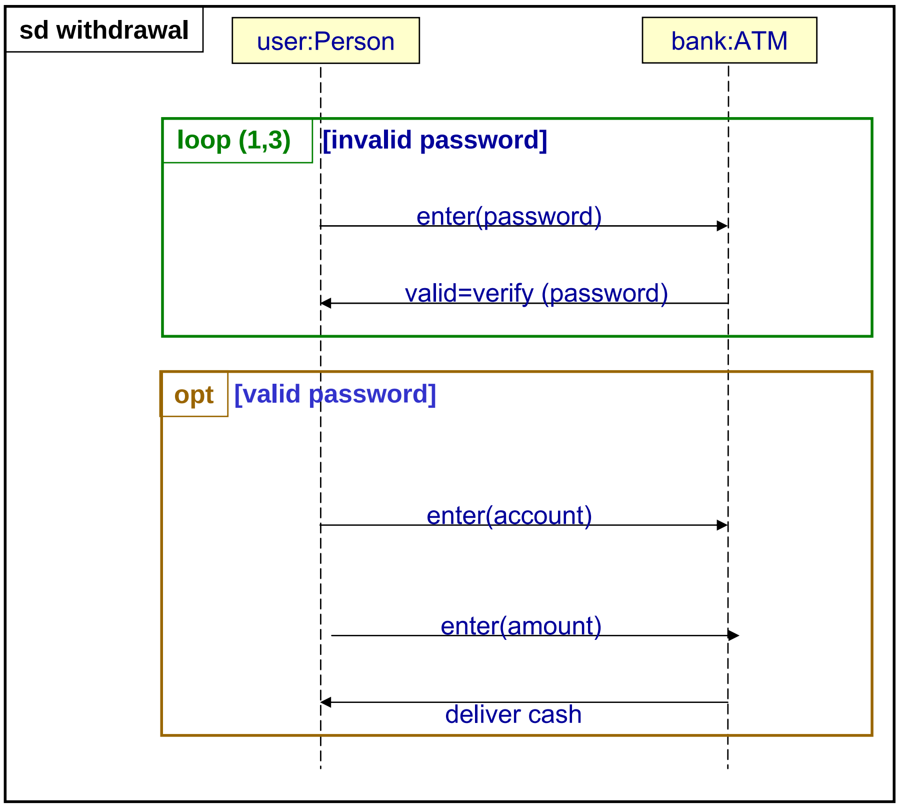
Withdraw cash from an ATM
Example of using operators sd



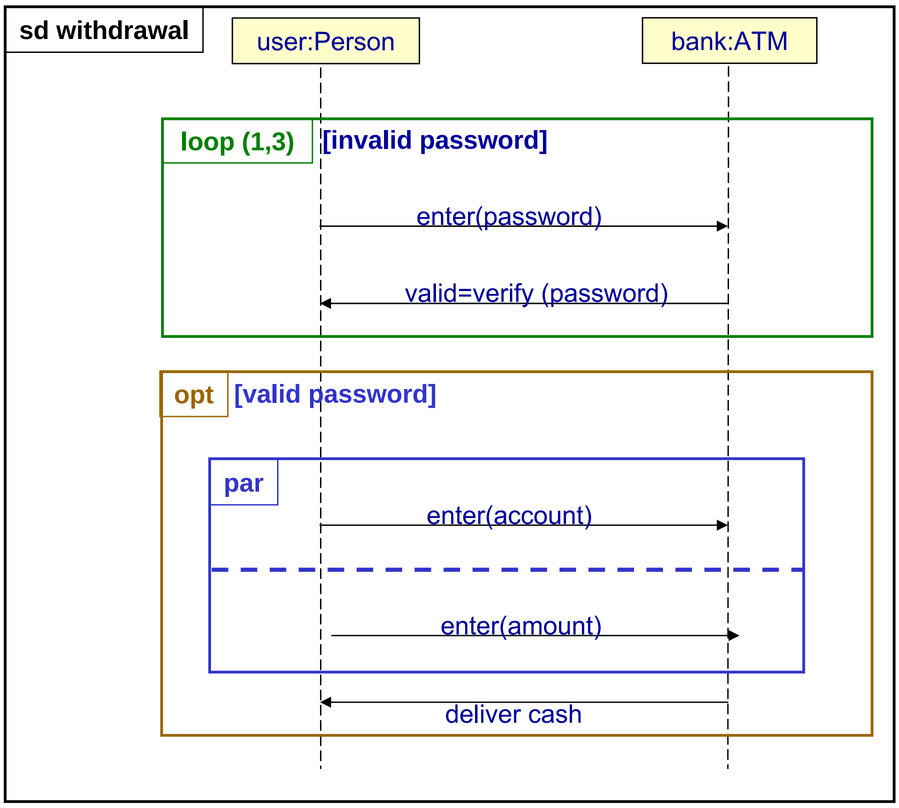
Withdraw cash from an ATM
Example of using operators sd, loop



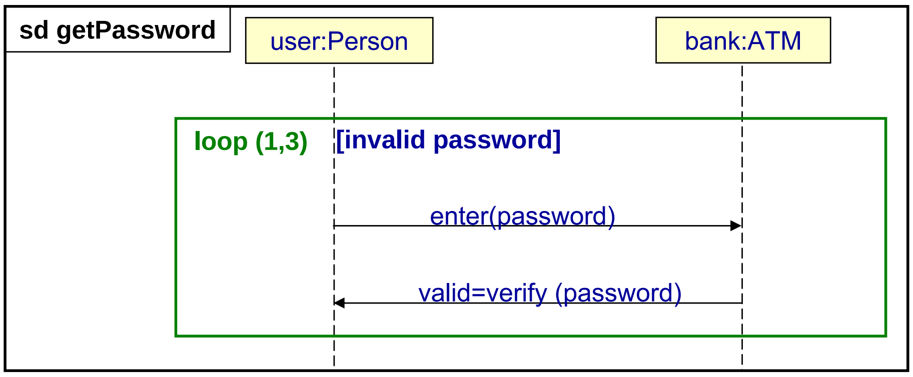
Withdraw cash from an ATM
Example of using operators sd, loop, opt

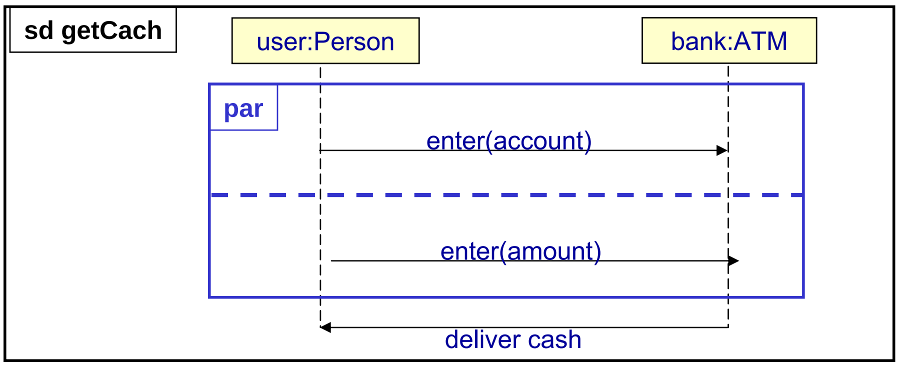


Withdraw cash from an ATM
Example of using operators sd, loop, opt, par

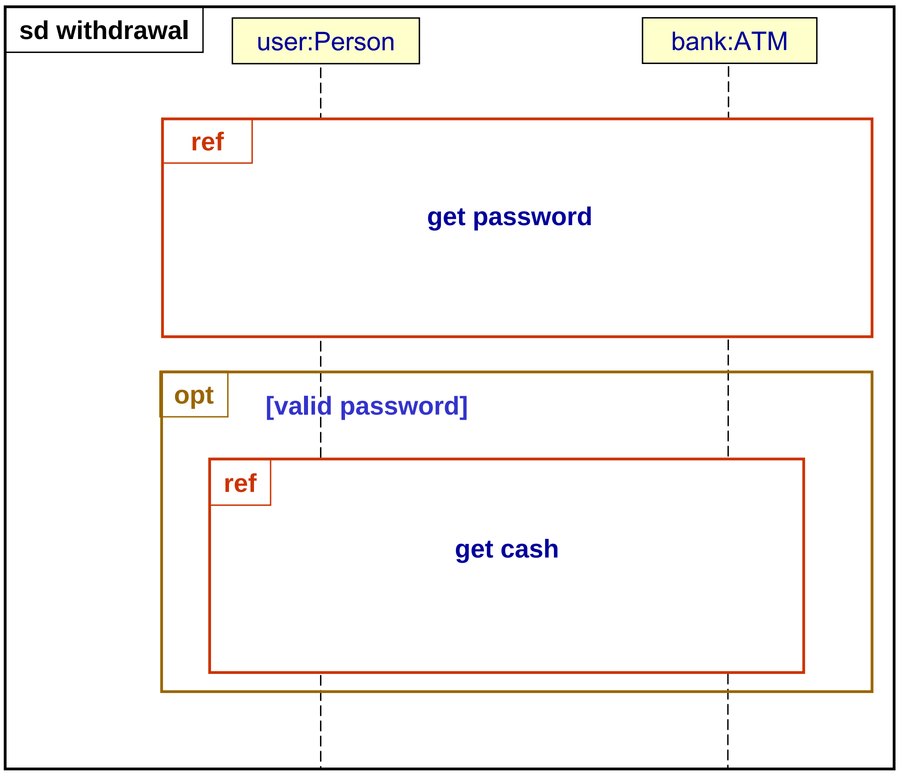


Suppose we had defined the following two sequence diagrams

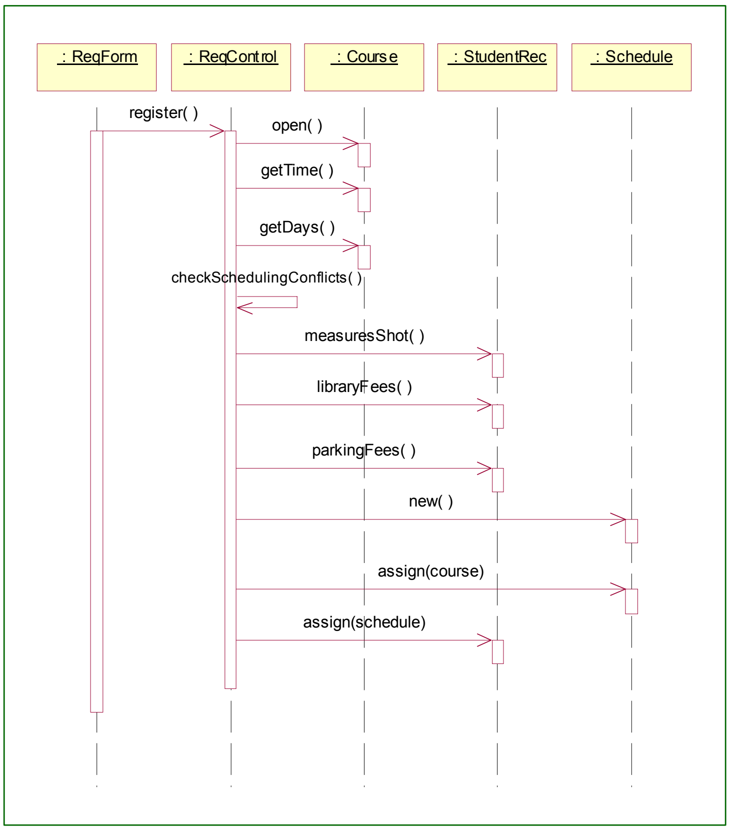




We can exploit them using the operator ref



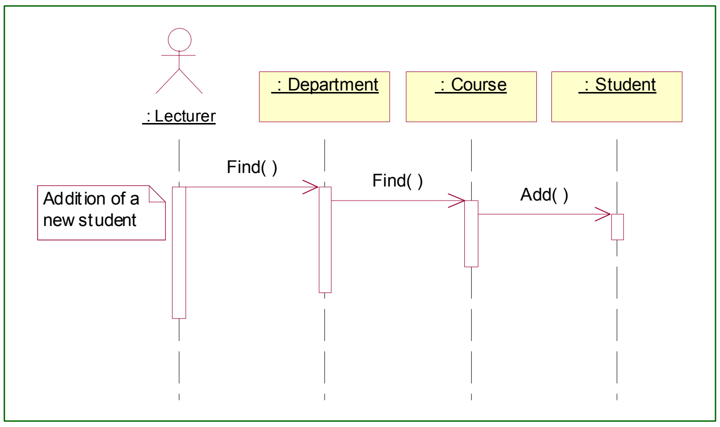
Example



Communication Diagrams (UML 2.0)
~ **Collaboration Diagrams** (UML V 1.3)

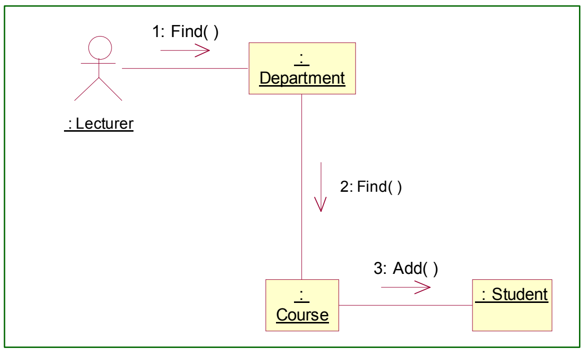
[B] Communication Diagrams (διαγράμματα επικοινωνίας) ≡ Collaboration Diagrams (v.1)

Sequence Diagram



≡

Communication Diagram



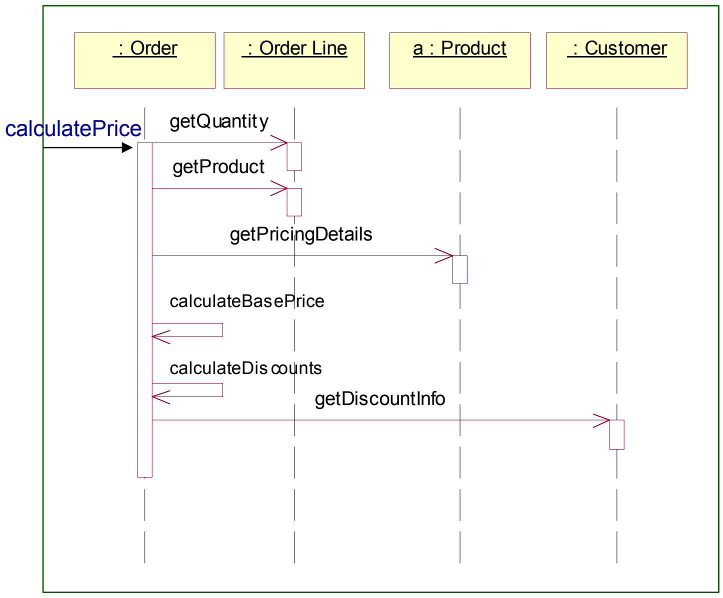
Here the sequence is indicated by numbering messages.

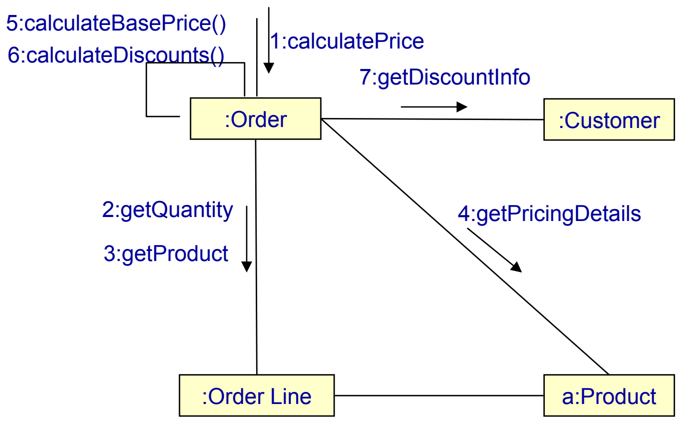
- Advantage: better exploits the drawing space (more compact)
- Weakness: makes it harder to see the sequence (comparing to sequence diagrams)

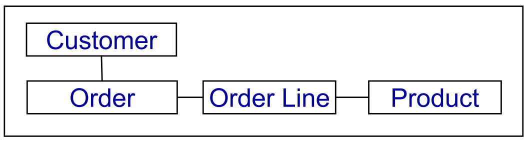
Sequence Diagram <=> Communication Diagram

- Automatic transformation is possible (e,g, F5 in Rational rose)

Sequence vs Collaboration Diagrams: Example

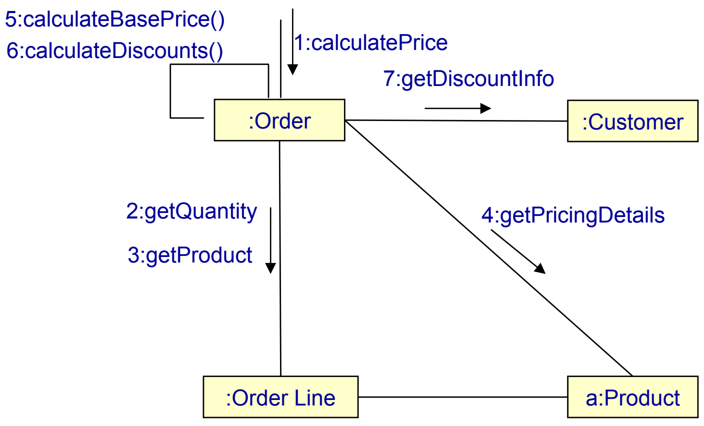


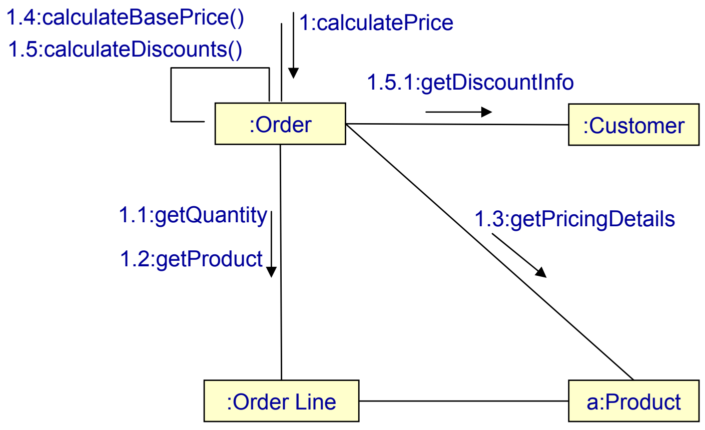




It is like an object diagram that shows message passing relationships instead of aggregation or generalization associations

Numbering Methods

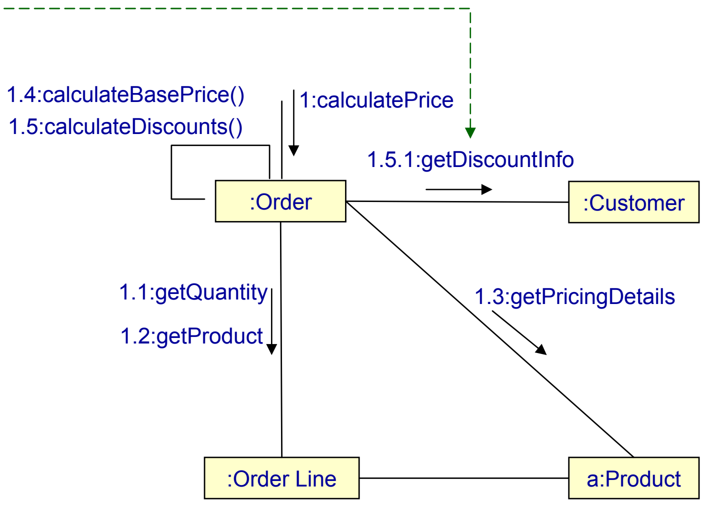




- Numbering methods
  - 1, 2, 3, …
  - 1, 1.1, 1.1.1, 1.1.2, 2.1 (Decimal numbering (used by UML))
- communication diagrams have not a precise notation for control logic
  - we could however use iteration markers and guards

Numbering Methods (II)

Why 1.5.1 and not 1.6?

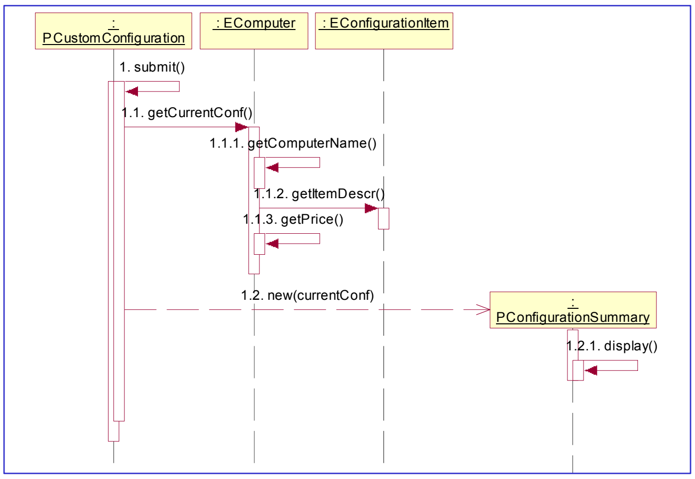


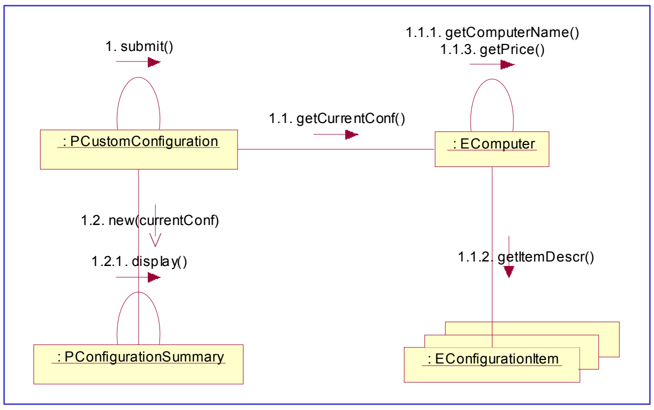
- Procedural (or nested) sequence
  - 1, 2, 2.1, 2.2
- Flat sequence
  - 1, 2, 3, 4

(2.1 and 2.2 are performed while the object of 2 is still active)

Sequence Diagrams vs Communication Diagrams

- Different developers have different preferences

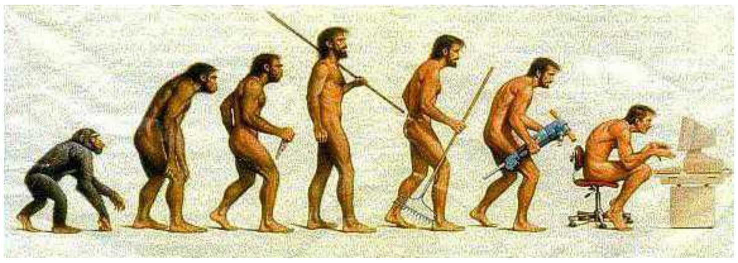




When to use Interaction Diagrams

- When to use Interaction Diagrams
  - To show the behaviour of several objects within a single Use Case
  - Tip: Focus on simplicity
    - If the control is complex split it to several interaction diagrams
- When use not Interaction Diagrams
  - If you want to look at the behaviour of a single object across multiple use cases, then use a **state diagram**
  - If you want to look at the behaviour across many use cases and many threads consider an **activity diagram**

State Diagrams



State Diagrams: Outline

- State Diagrams
- Concurrent State Diagrams

State Diagrams

- A state diagram describes all possible **states** that a particular object can get into and how the object's state **changes** as a result of events that reach the object.

- Usually are drawn for a single class

State Diagrams

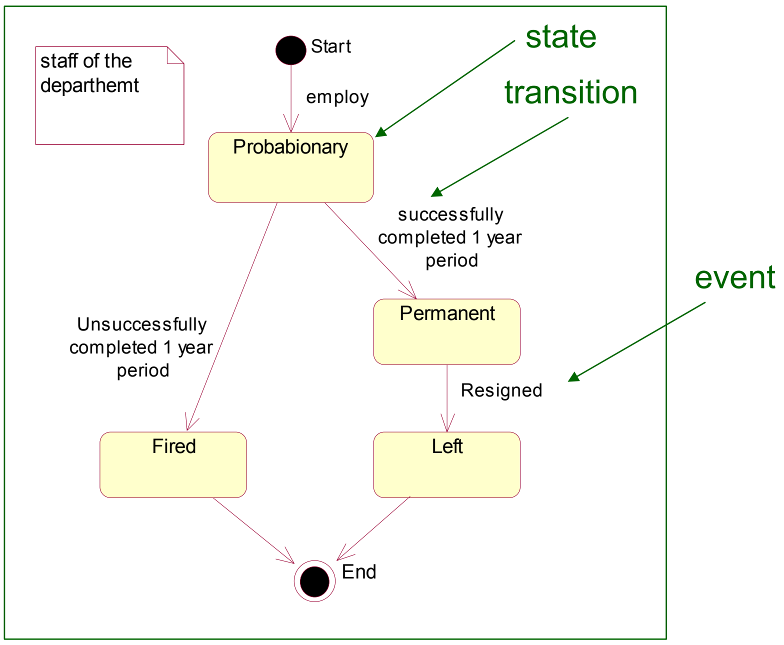
We can use them for various perspectives

Perspectives

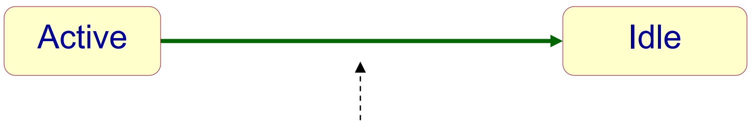
- **Conceptual**:
  - Business processes
    - ***what are the states of an order of a company ? Are cancellations possible ?***
- **Design**
  - ***states to be handled by the interfaces of the classes***
- **Implementation**
  - ***actual states of the implementation objects***

Basic Notions

- States
- Transitions
  - Events
- Activities



Transitions



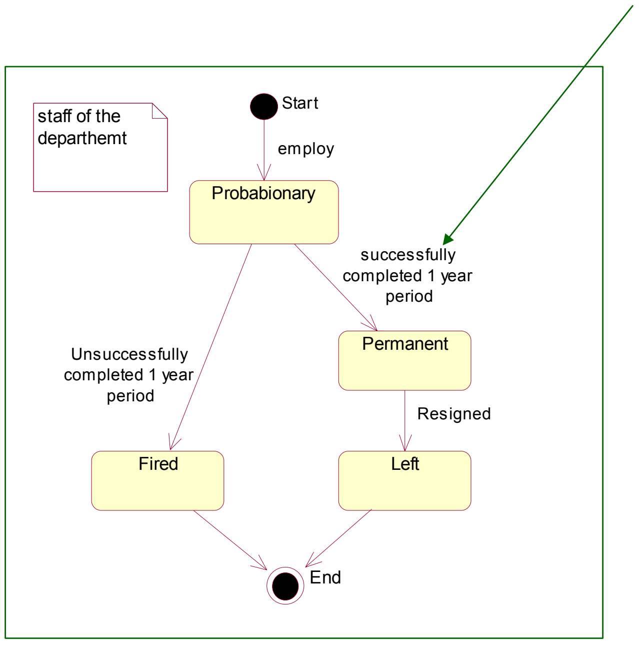
Transition labels: **Event[Condition]/Action**

- all three are optional

- Event:
  - if nil then when the task is completed we continue
- Condition
  - logical condition (transition occurs if its value is True)
  - the guards of transitions from a state must be <u>mutually exclusive</u> so that to have a <u>unique</u> next state
- Action
  - processes that occur quickly and are <u>not interruptible</u>

Example of a transition label of the form **Event[Condition]/Action**

After 1 year [successful so far]/inform the director of personnel



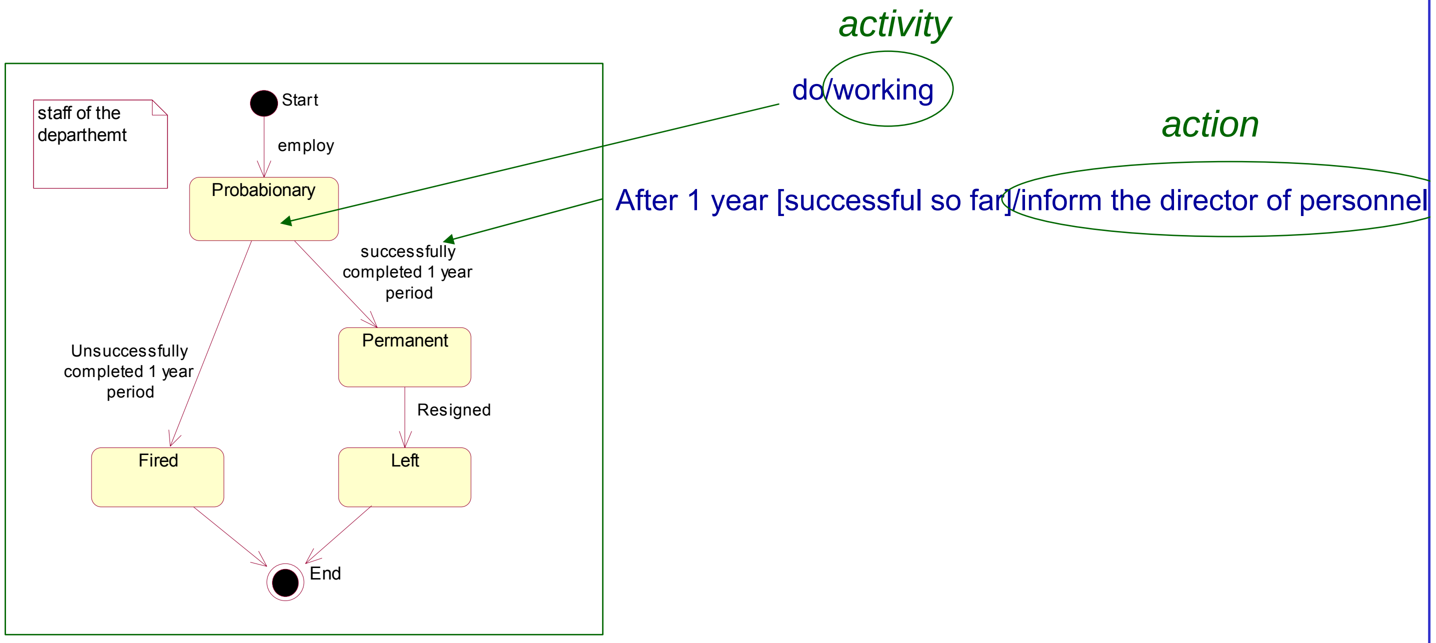
Kinds of Events

- Entry
  - any action related to entry event is executed whenever the given state is entered via a transition
- Exit
  - when we exit the transition
- After 20 minutes
  - example of event generated after a period of time
- When (temperature > 40)
  - example of event generated when a condition becomes true
- …

V 1. : Actions vs Activities
V 2.0: Internal vs External Activities

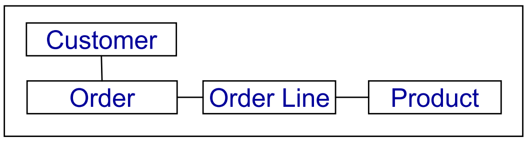
- Distinctions
  - Actions are associated with transitions (usually quick)
    - **not interruptible**
  - Activities are associated with states (can take longer)
    - **can be interrupted** by events
- Each state has an activity associated with it
  - syntax: **do/activity**

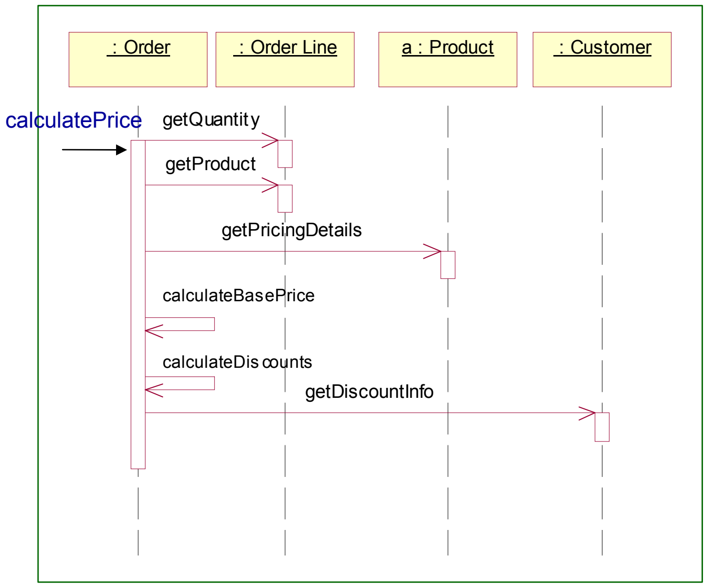
Actions vs Activities


activity
do/working
action
After 1 year [successful so far]/inform the director of personnel
staff of the departhemt
Start
employ
Probabionary
successfully completed 1 year period
Permanent
Unsuccessfully completed 1 year period
Resigned
Fired
Left
End

Example:

Recall the class diagram about Orders and Products and the interaction diagram about calculating the price of a product

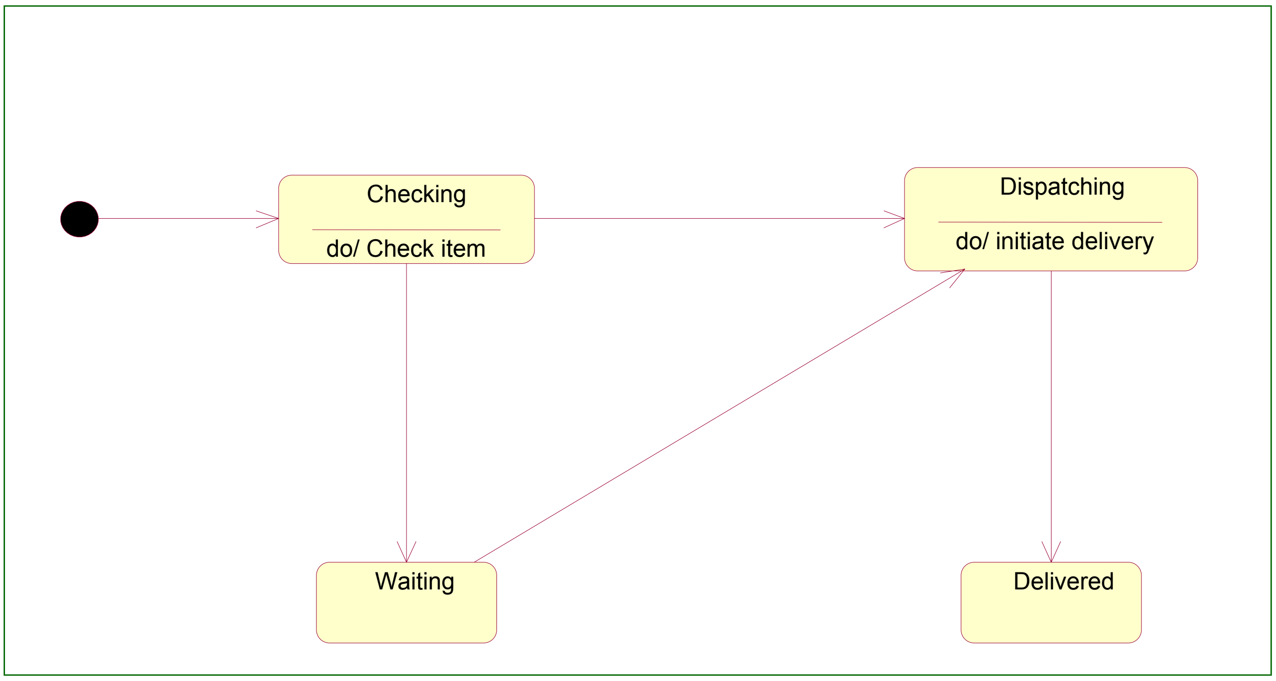




Which are the (important) states of an order object?

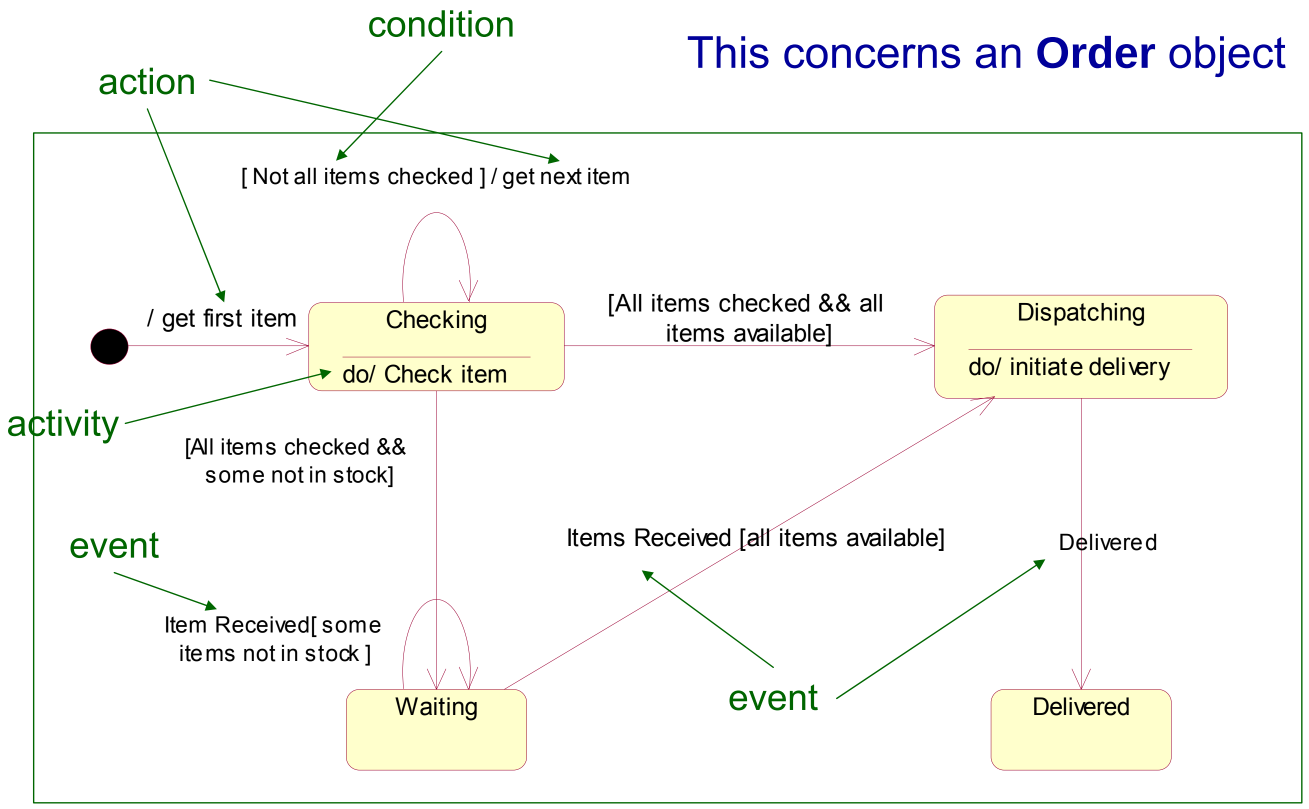
Example: The states of an Order object

This concerns an **Order** object



Example: The states of an Order object (II)

This concerns an **Order** object

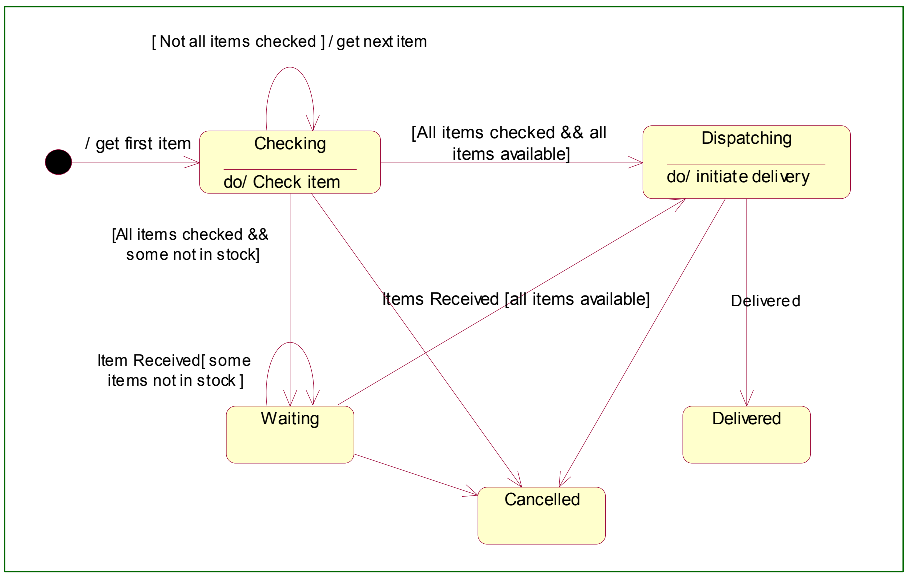


condition
action
[Not all items checked] / get next item
/ get first item
Checking
do/ Check item
[All items checked && all items available]
Dispatching
do/ initiate delivery
activity
[All items checked && some not in stock]
event
Items Received [all items available]
Delivered
Item Received[some items not in stock]
Waiting
event
Delivered

Example (cont)

Assume we want to be able to <u>cancel</u> at any point

Solution 1: add a cancel transition from each state

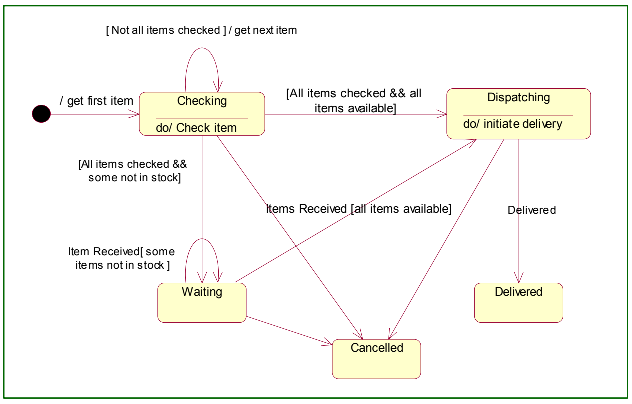


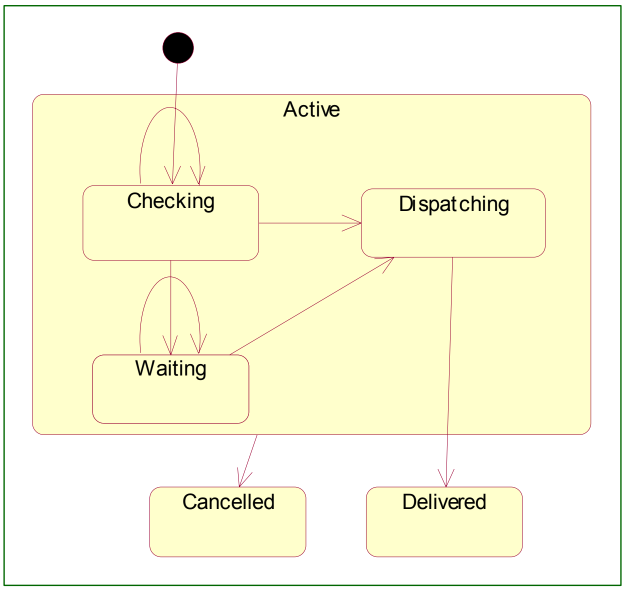
[Not all items checked] / get next item
/ get first item
Checking
do/ Check item
[All items checked && all items available]
Dispatching
do/ initiate delivery
[All items checked && some not in stock]
Items Received [all items available]
Delivered
Item Received[some items not in stock]
Waiting
Delivered
Cancelled

Example (cont): **Superstates**

Assume we want to able to <u>cancel</u> at any point

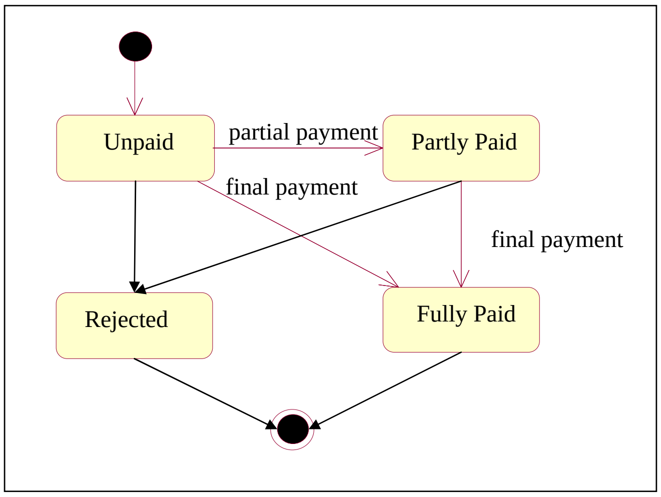
Solution 2: Define a ***superstate*** *and cancel only there (the substates inherit it)*


[Not all items checked] / get next item
/ get first item
Checking
do/ Check item
[All items checked && all items available]
Dispatching
do/ initiate delivery
[All items checked && some not in stock]
Items Received [all items available]
Delivered
Item Received[some items not in stock]
Waiting
Delivered
Cancelled



Example

The states of an Order object w.r.t. payment



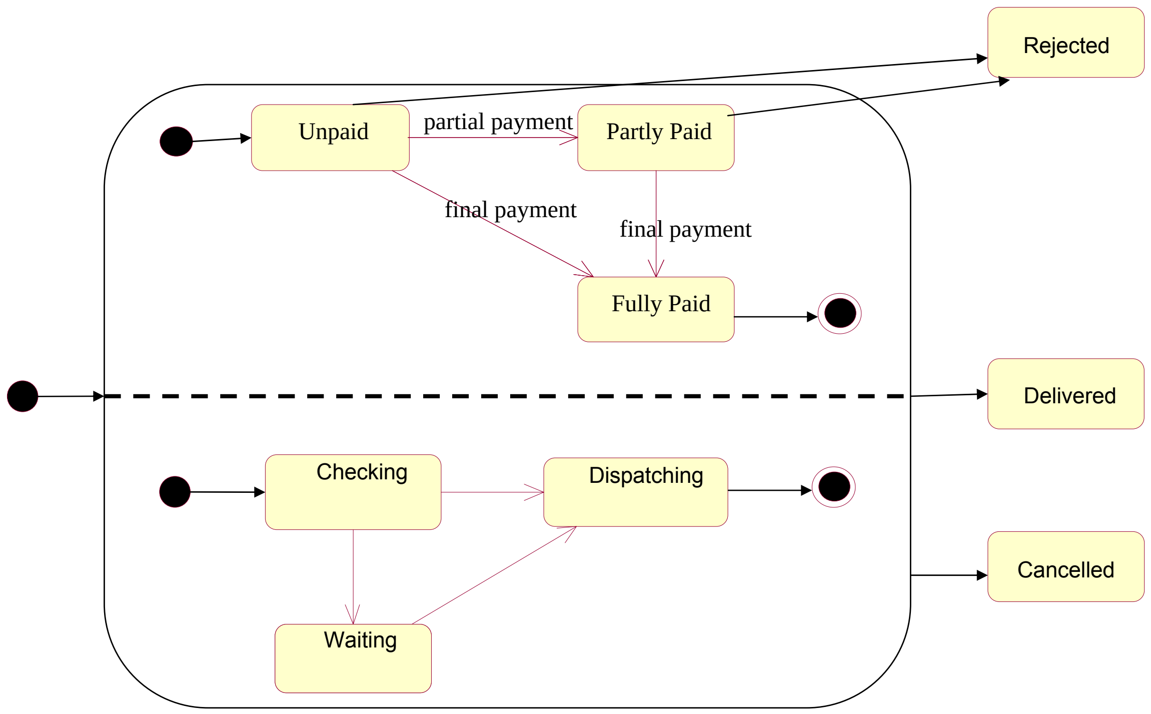
How to combine these states with the previous ones (i.e. checking, waiting, dispatching, delivered, etc) ?

Concurrent State Diagrams

Allow “parallel” execution

- multiple states are active concurrently
- when an object leaves the concurrent states, it is only in one state

Concurrent State Diagrams

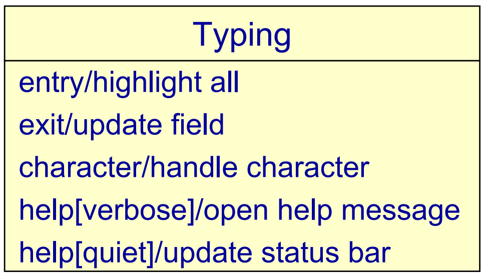


Recall fork and join from activity diagrams

Internal Activities (or self-transitions)

- States can react to events without transition, using <u>internal activities</u>
  - like self-transitions
  - putting the event[guard]/activity inside the state box

- Example of internal events of the typing state of a text field


Typing
entry/highlight all
exit/update field
character/handle character
help[verbose]/open help message
help[quiet]/update status bar

When to use State Diagrams

- To describe the behaviour of **an object** <u>across **several** use cases</u>
- not flexible if there are many collaborating objects
  - in this case it's better to use
    - interaction diagrams
    - activity diagrams

Classical cases for using state machine diagrams:

- Example applications
  - Cruise controls
  - vendor machines
- Formal methods
  - verification of network protocols

Summary

Sequence diagrams (and *Communication diagrams*)

- illustrate the classes that participate in a use case and the messages that pass between them.

State diagrams

- show the different states that a single class passes through in response to events.