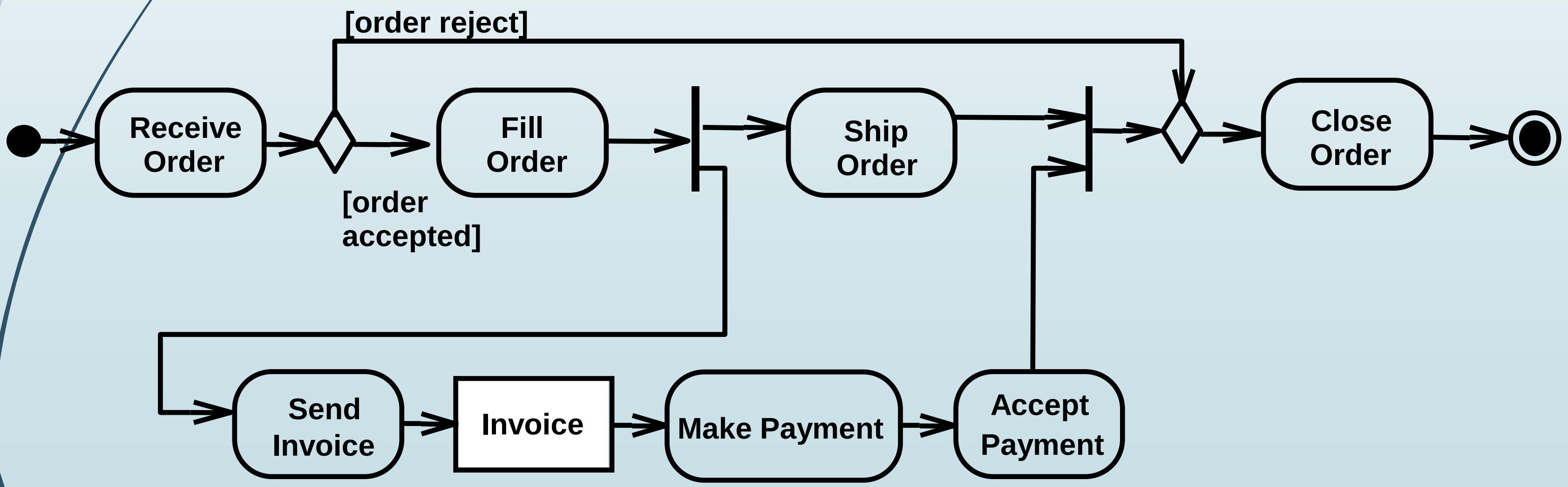
Activity diagrams

Activity diagrams

- Useful to specify software or hardware system behaviour
- Based on data flow models – a graphical representation (with a Directed Graph) of how data move around an information system

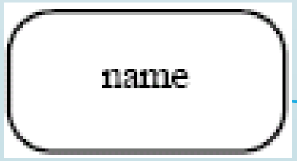


Some definitions

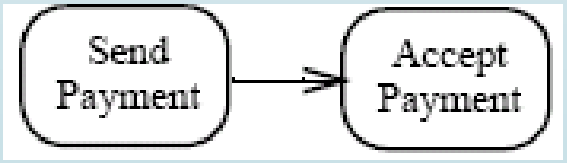
- ***Flow***: permits the interaction between two nodes of the activity diagram (represented by edges in activity diagram)
- ***State***: a condition or situation in the life of an object during which it satisfies some conditions, performs some activities, or waits for some events
- ***Type***: specifies a domain of objects together with the operations applicable to the objects (also none); includes primitive built-in types (such as integer and string) and enumeration types
- ***Token***: contains an object, datum, or locus of control, and is present in the activity diagram at a particular node; each token is distinct from any other, even if it contains the same value as another
- ***Value***: an element of a type domain

Actions

- The fundamental unit of executable functionality in an activity
- The execution of an action represents some transformations or processes in the modeled system (creating objects, setting attribute values, linking objects together, invoking user-defined behaviours, etc.)

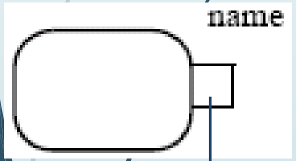


$\longrightarrow$ Description of the action behaviour

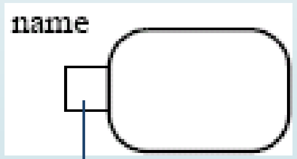


Pins

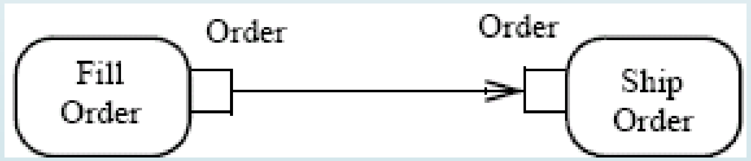
- Actions can have inputs and outputs, through the pins
- Hold inputs to actions until the action starts, and hold the outputs of actions before the values move downstream
- The name of a pin is not restricted: generally recalls the type of objects or data that flow through the pin

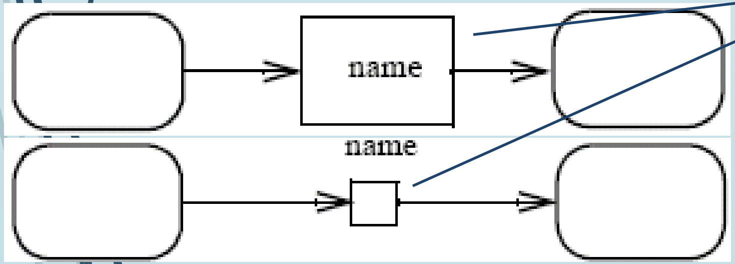


Output pins



Input pins

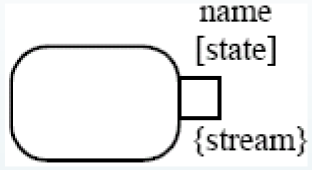


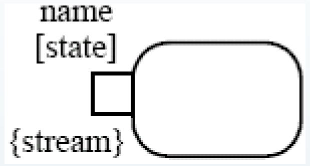


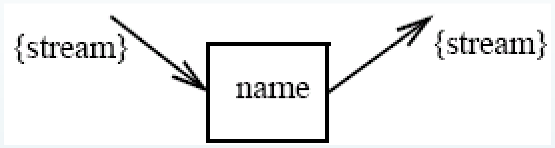
Standalone pin notations: the output pin and the input pin have the same name and same type

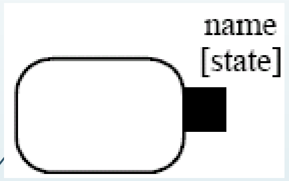
Special kind of pins

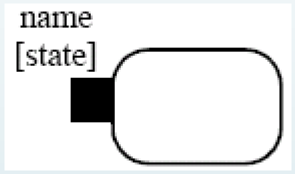
- **Streaming Parameters** (notated with {STREAM}): accept or provide one or more values while an action is executing

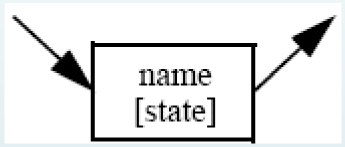


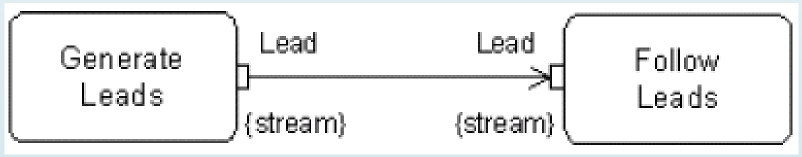




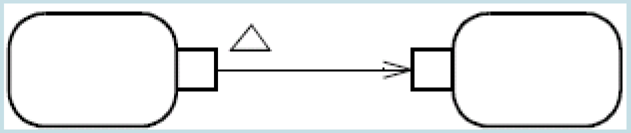
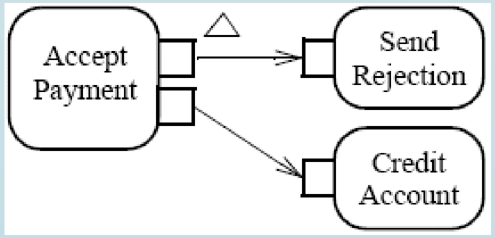






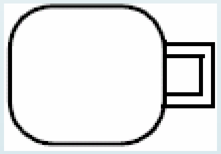


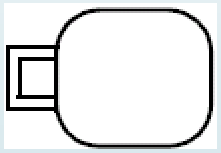
- **Exception Output Parameters** (notated with a triangle)
  - Provide values to the exclusion of any other output parameter or outgoing control of the action
  - The action *must immediately terminate*, and the output cannot quit

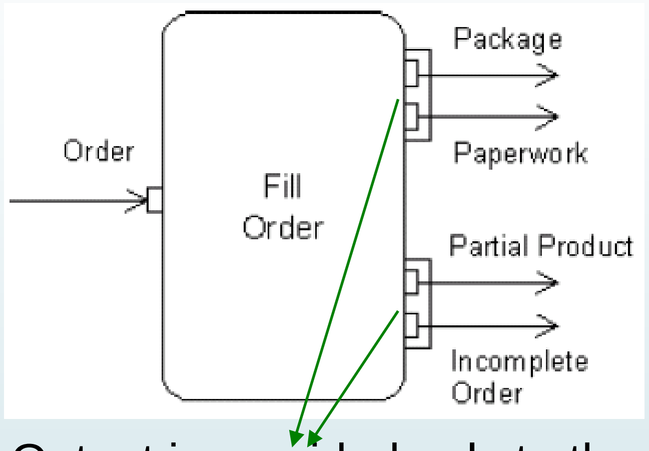



Special kind of pins

- **Parameter Sets**
  - group of parameters
  - the action can only accept inputs from the pins in one of the sets
  - the action can only provide outputs to the pins in one of the sets

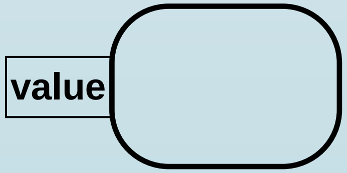


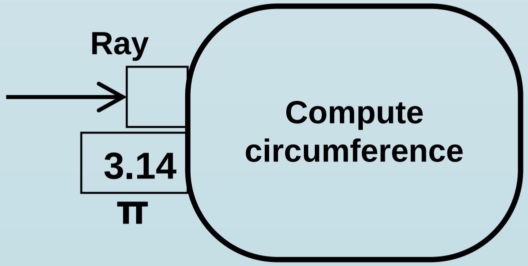




Output is provided only to the first or to the second set

- **ValuePin**: special kind of input pin defined to provide constant values (the type of specificated value must be compatible with the type of the value pin)



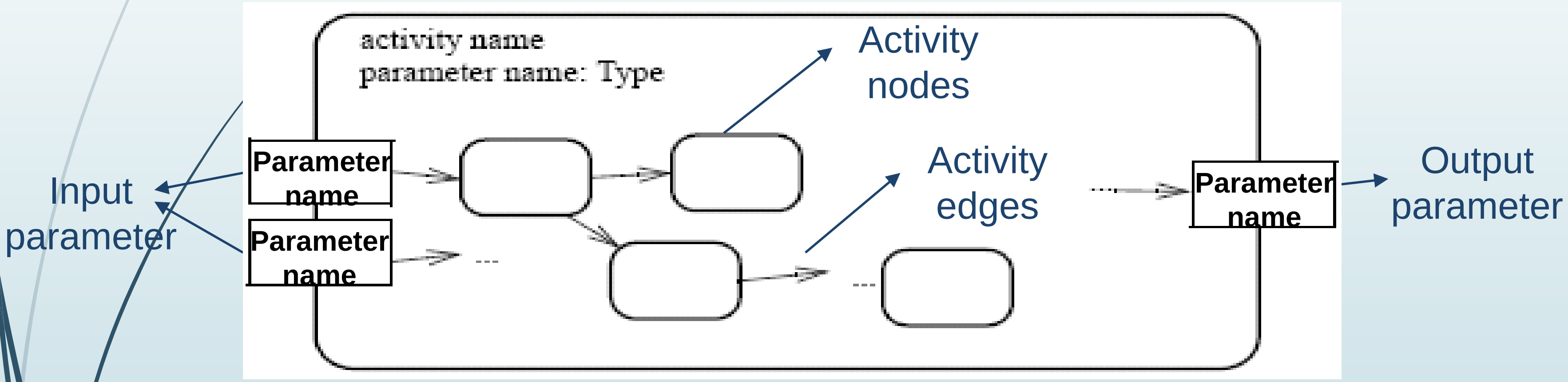


Conditions to start and end actions

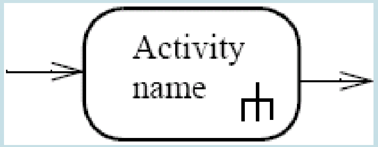
- An action can start only if:
  - all non-stream inputs have arrived (or at least one stream input if there are only stream inputs)
- The action can finish only if:
  - all inputs have arrived (streaming inputs included)
  - all non-stream and non-exception outputs (or an exception outputs) have been provided
- ***Prevent deadlock***: an input pin of an action cannot accept tokens until all the input pins of the action can accept them

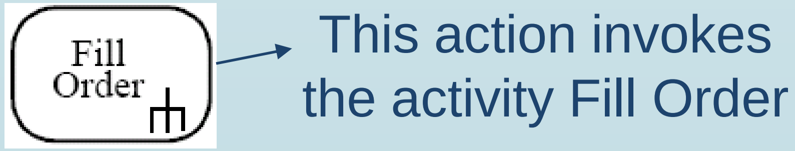
Activities

- An activity is the specification of parameterized behaviour as the coordinated sequencing of subordinate units whose individual elements are actions
- Uses parameters to receive and provide data to the invoker



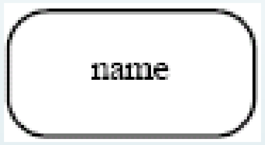
- An action can invoke an activity to describe its action more finely



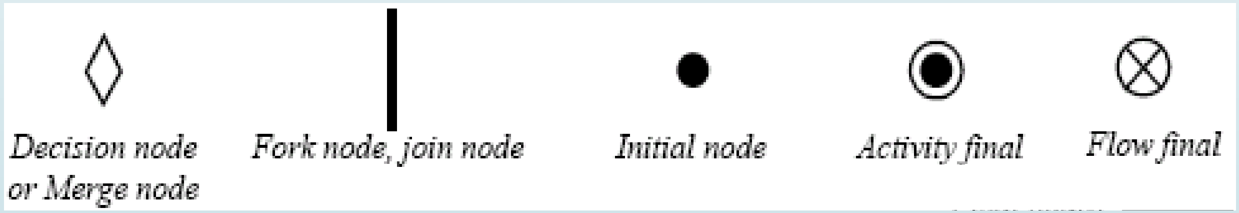


Activity nodes

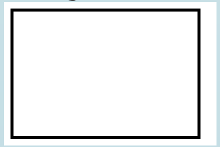
- Three type of activity nodes:
  - **Action nodes**: executable activity nodes; the execution of an action represents some transformations or processes in the modeled system (already seen)



  - **Control nodes**: coordinate flows in an activity diagram between other nodes

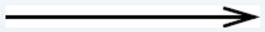


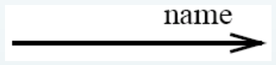
  - **Object nodes**: indicate an instance of a particular object, may be available at a particular point in the activity (i.e Pins are object nodes)

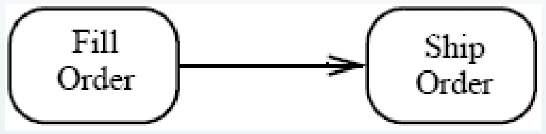


Activity edges

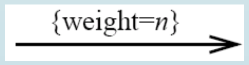
- Are directed connections
- They have a source and a target, along which tokens may flow

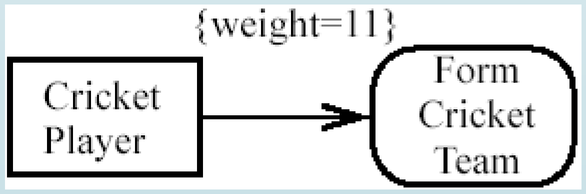


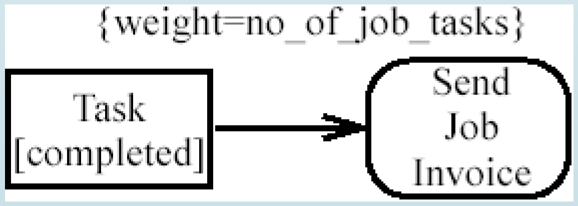




- Any number of tokens can pass along the edge, in groups at one time, or individually at different times
- ***Weight***: determines the minimum number of tokens that must traverse the edge at the same time



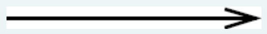


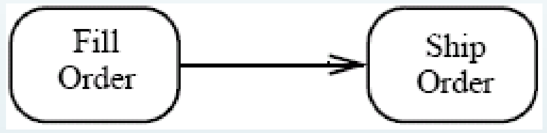


In this example we use a non-constant weight: an invoice for a particular job can only be sent when all of its tasks have been completed

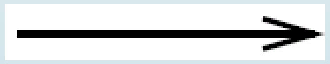
Activity edges

- Two kinds of edges:
  - **Control flow edge** - is an edge which starts an activity node after the completion of the previous one by passing a control token

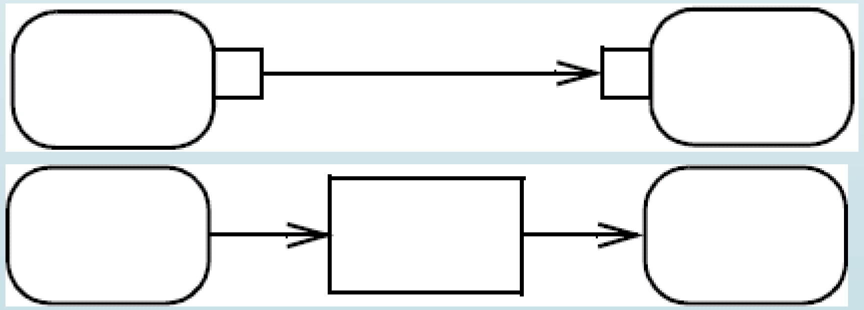




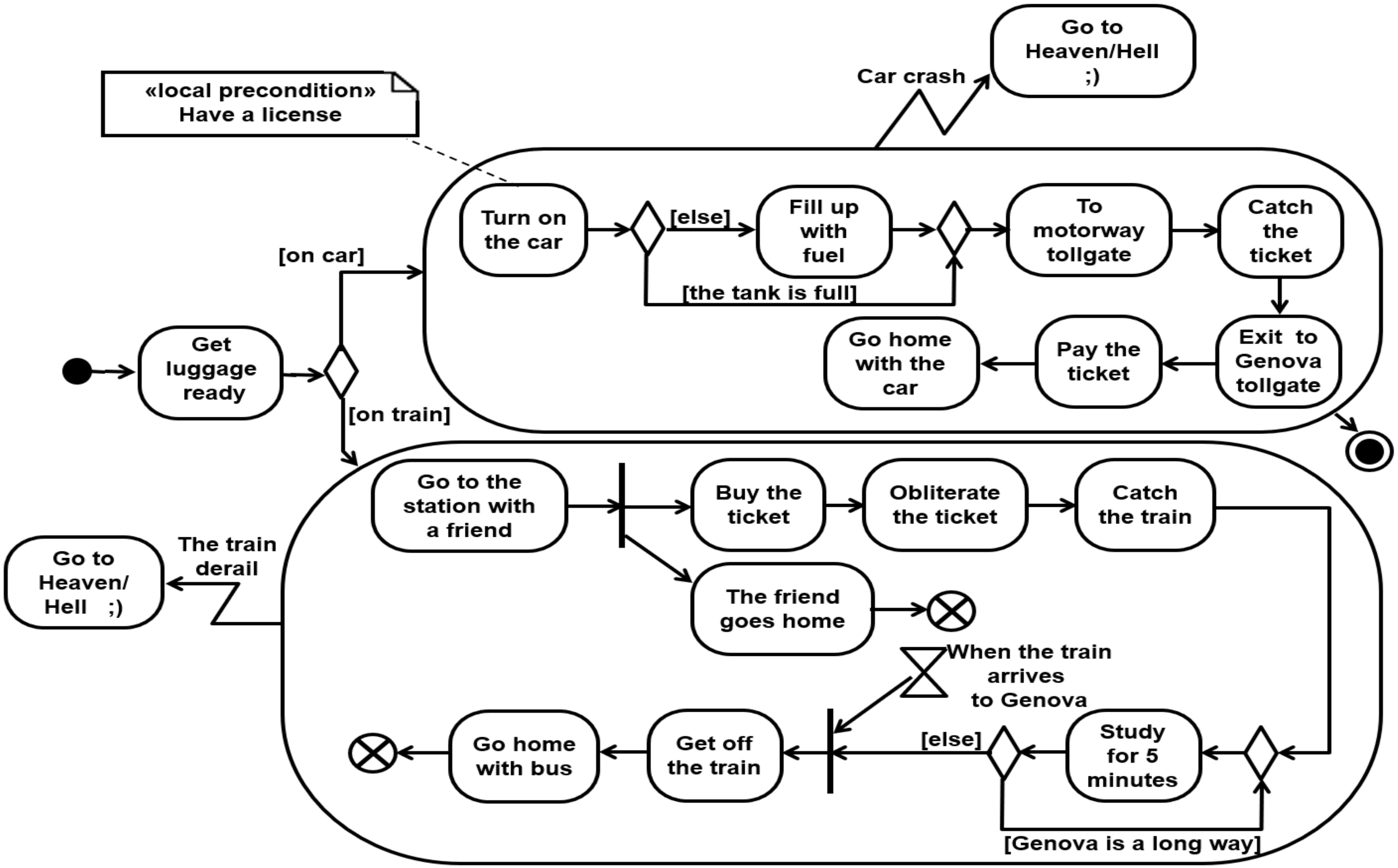
  - **Object flow edge** - models the flow of values to or from object nodes by passing object or data tokens



Notation without activity nodes

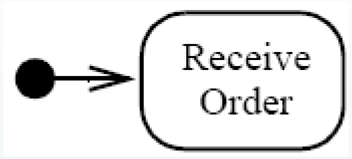


Example – go to Genova

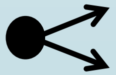


Control nodes – initial nodes

- In an *activity* the flow starts in initial nodes, that return the control immediately along their outgoing edges



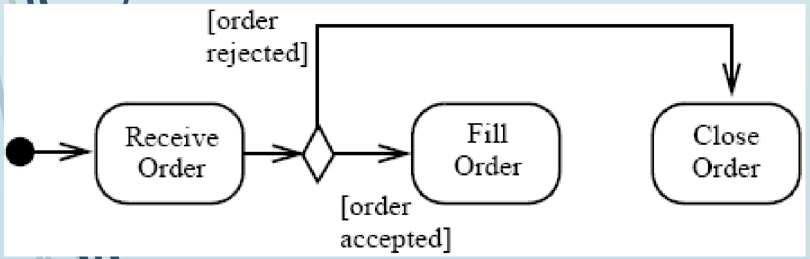
- If there are more than one initial node, a control token is placed in each initial node when the activity is started, initiating multiple flows
- If an initial node has more than one outgoing edge, only one of these edges will receive control, because initial nodes cannot duplicate tokens

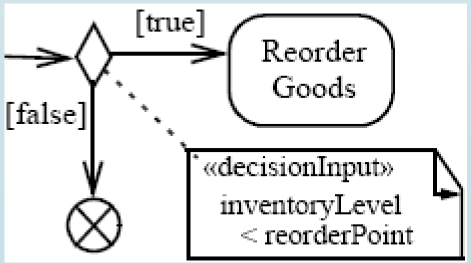


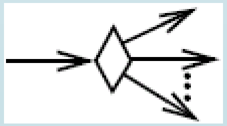
A or B ?

Control nodes – decision nodes

- Route the flow to one of the outgoing edges (tokens are not duplicated)
- Guards are specified on the outgoing edges or with the stereotype «decisionInput»
- There is also the predefined guard **[else]**, chosen only if the token is not accepted by all the other edges
- If all the guards fail, the token remains at the source object node until one of the guards accept it

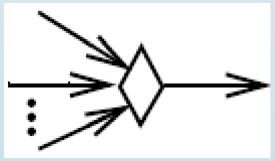


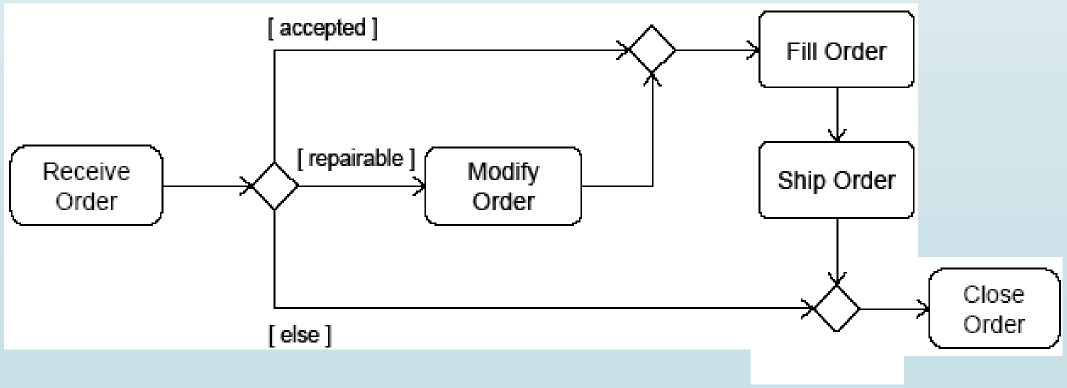




Control nodes – merge nodes

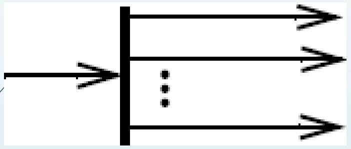
- Bring together multiple alternate flows
- All controls and data arriving at a merge node are immediately passed to the outgoing edge
- There is no synchronization of flows or joining of tokens

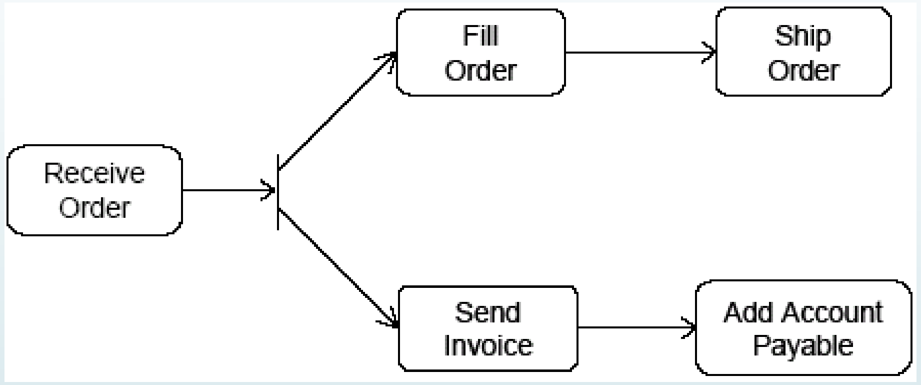




Control nodes – fork nodes

- Fork nodes split flows into multiple concurrent flows (tokens are duplicated)

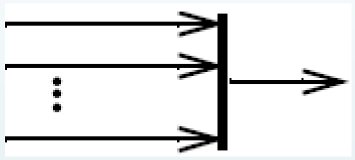


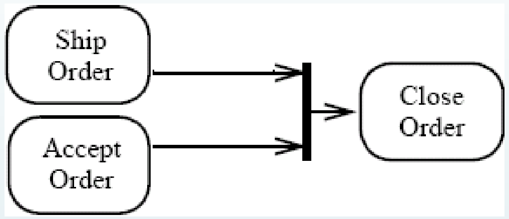


- State machine forks in UML 1.5 required synchronization between parallel flows through the state machine RTC step (it means that the first state in each branch is executed, then the second one, etc.)
- UML 2.0 activity forks model unrestricted parallelism

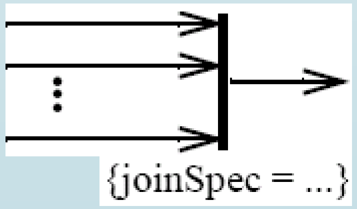
Control nodes – join nodes

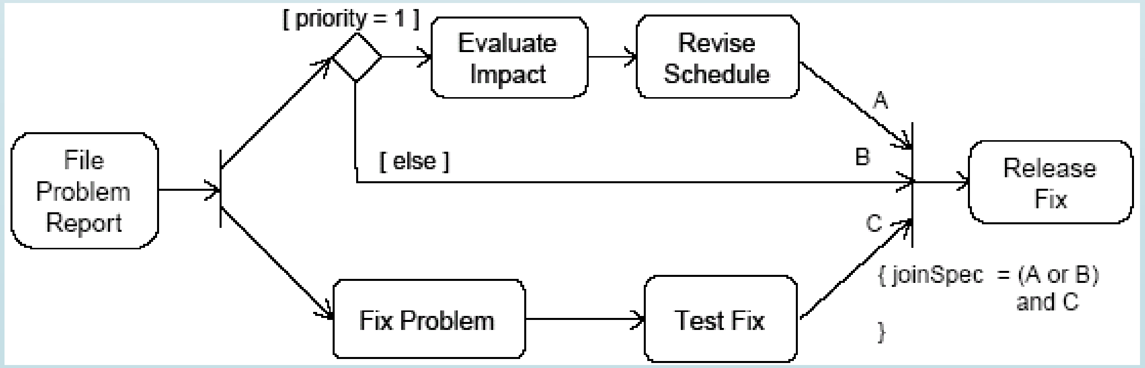
- Join nodes synchronize multiple flows





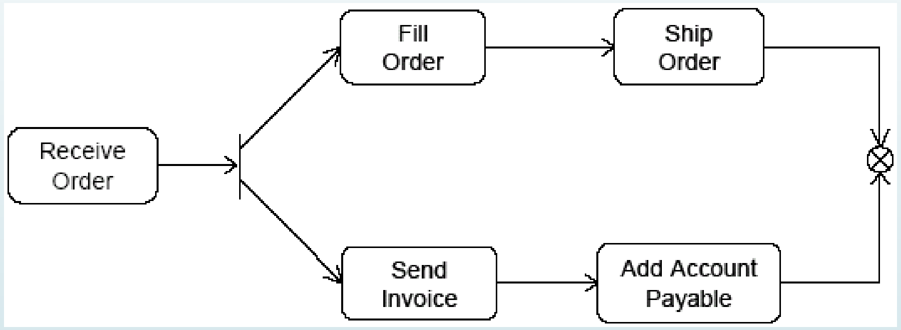
- Generally, controls or data must be available on every incoming edge in order to be passed to the outgoing edge, but user can specify different conditions under which a join accepts incoming controls and data using a *join specification*



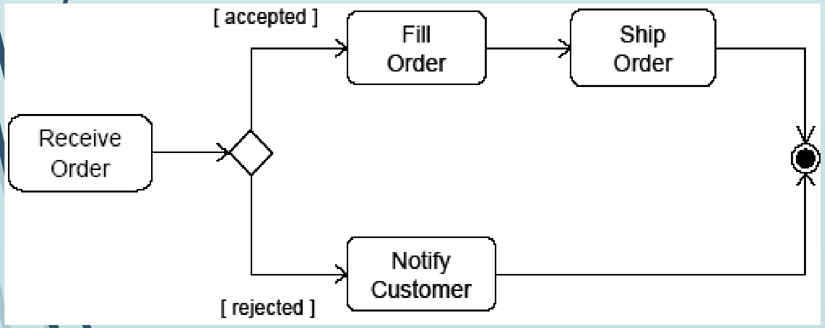


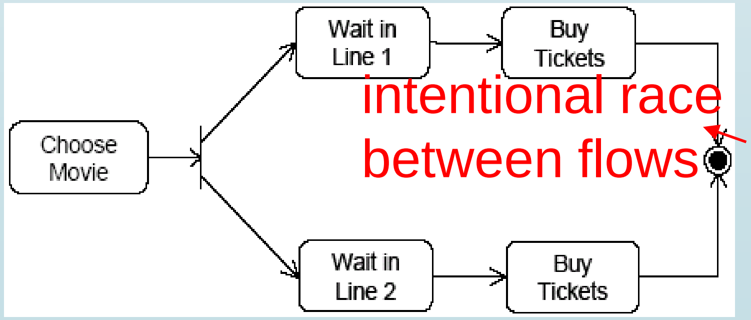
Control nodes – final nodes

- **Flow final:** ⊗
  - destroys the tokens that arrive into it
  - the activity is terminated when all tokens in the graph are destroyed



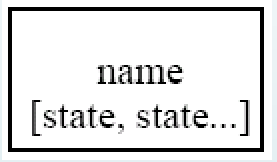
- **Final node:** ◉
  - the activity is terminated when the first token arrives



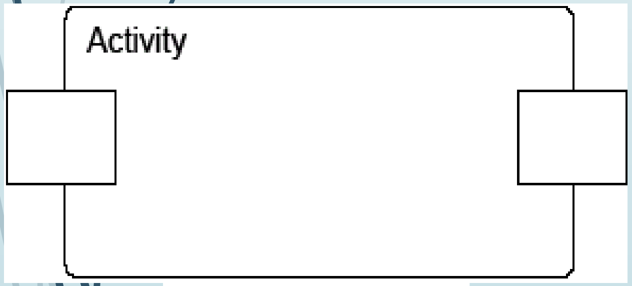


Object nodes

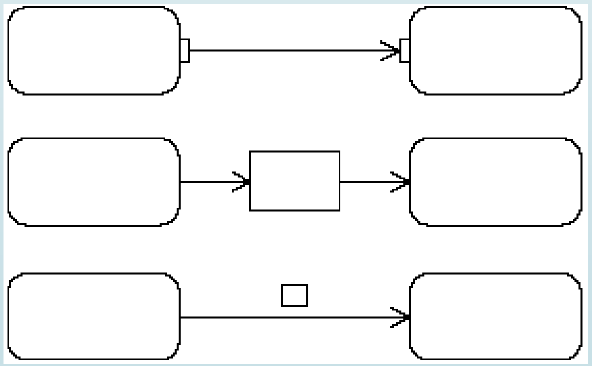
- Hold data temporarily while they wait to move through the graph
- Specify the type of values they can hold (if no type is specified, they can hold values of any type)
- Can also specify the state of the held objects



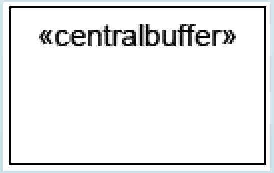
- There are four kinds of object nodes:



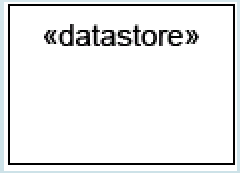
Activity Parameter Nodes



Pins
(three differents notations)



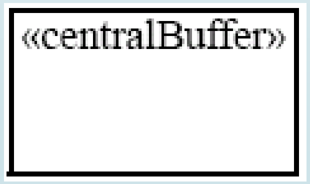
Central Buffer Nodes

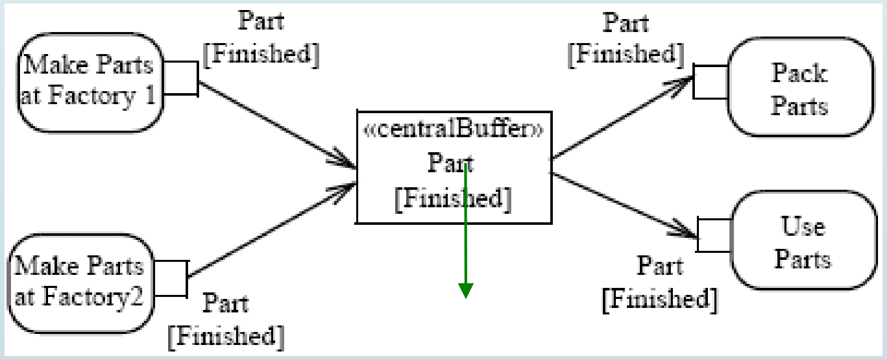


Data Store Nodes

Object nodes – centralBuffer

- A central buffer node is an object node that manages flows from multiple sources and destinations (as opposed to pins and parameters)
- Acts as a buffer for multiple input flows and output flows
- Is not tied to an action like pins, or to an activity like activity parameter nodes

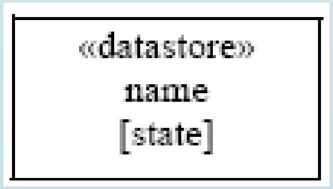


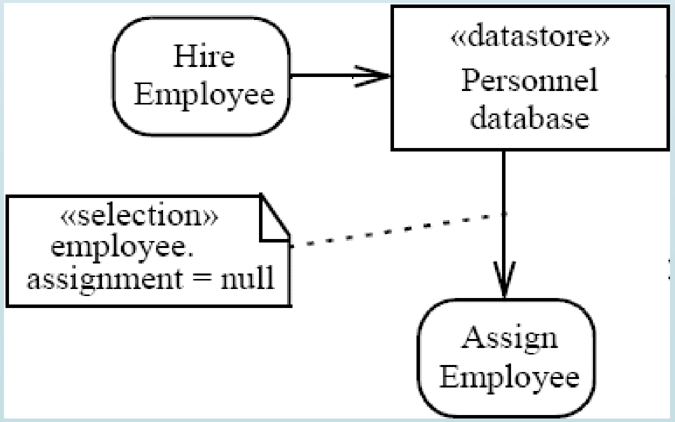


The centralBuffer node collects the object Parts, and each Part can be used or packet (but not both)

Object nodes – datastore

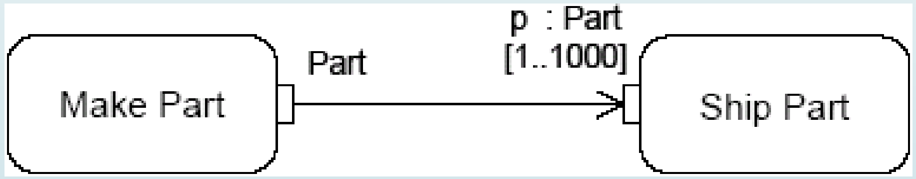
- Is a specific central buffer node which stores objects persistently
- Keeps all tokens that enter into it
- Tokens chosen to move downstream are copied so that tokens never leave the data store
- If arrives a token containing an object already present in the data store, this replaces the old one
- Tokens in a data store node cannot be removed (they are removed when the activity is terminated)



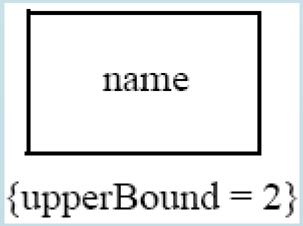


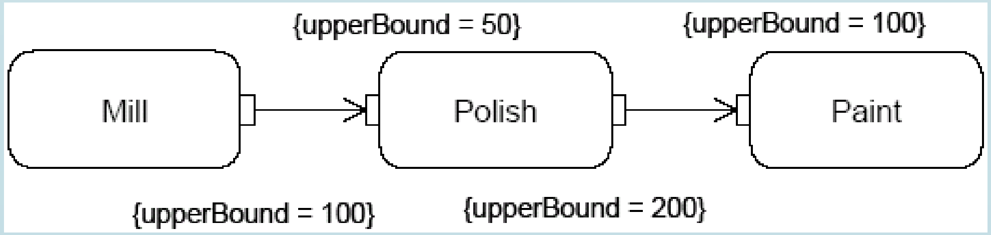
Object nodes - multiplicities and upperBound

- ***Multiplicities***: specify the minimum (≥0) and maximum number of values each pin accepts or provides at each invocation of the action:
  - when is available the minimum number of values, the action can start
  - if there is more values than the maximum, the action takes only the first maximum value



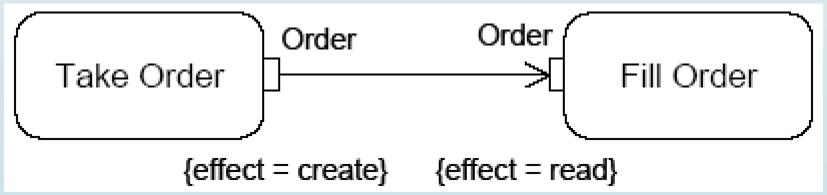
- ***UpperBound***: shows the maximum number of values that an object node can hold: at runtime, when the upper bound has been reached, the flow is stopped (buffering)

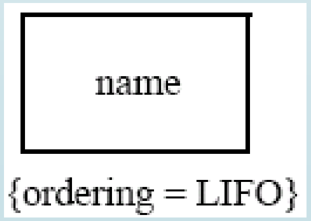


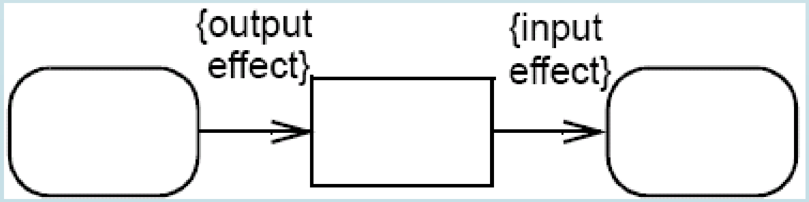


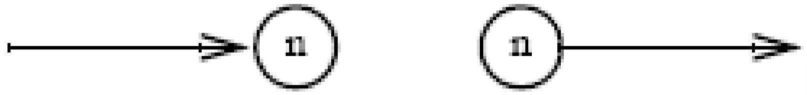
Object nodes – effect and ordering

- ***Effect***: pins can be notated with the effect that their actions have on objects that move through the pin
- The effects can be: 'create' (only on output pins), 'read', 'update' or 'delete' (only on input pins)
- ***Ordering***: specifies the order in which the tokens of an object node are offered to the outgoing edges (FIFO, LIFO or modeler-defined ordering)



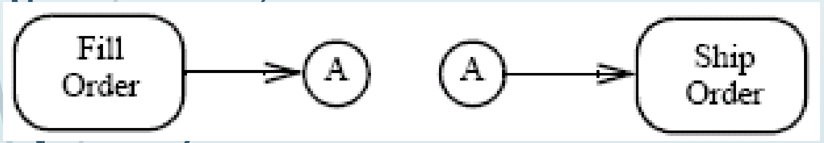




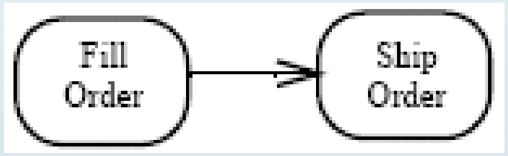


Activity edges – presentation options

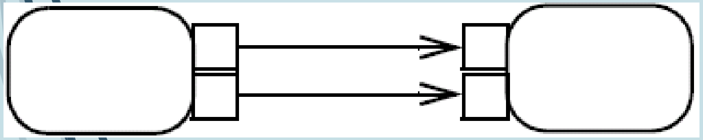
- An edge can also be notated using a connector
- Every connector with a given label must be paired with exactly one other with the same label on the same activity diagram



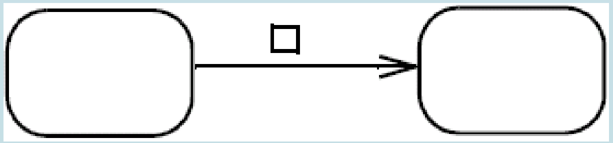
is equivalent to



- To reduce clutter in complex diagrams, object nodes may be elided

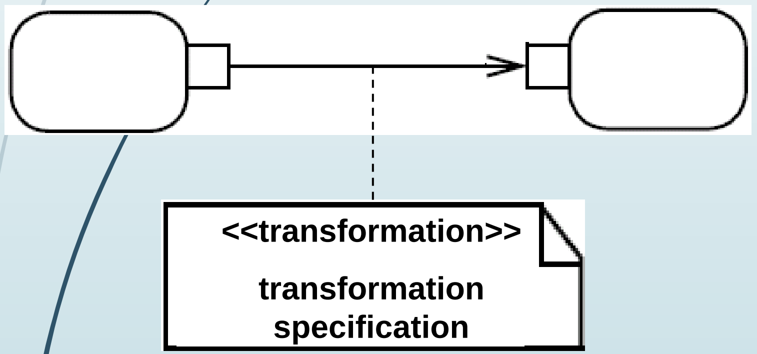


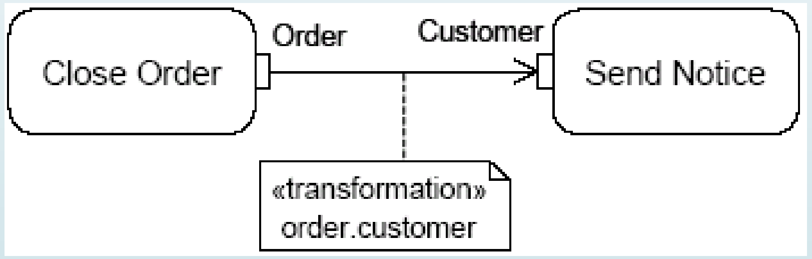
is equivalent to



Activity edges - transformation

- It is possible to apply a transformation of tokens as they move across an object flow edge (each order is passed to the transformation behaviour and replaced with the result)

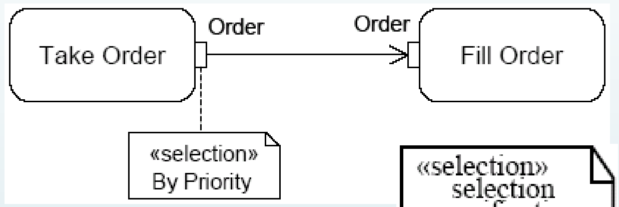


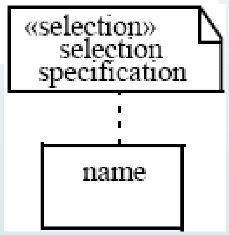


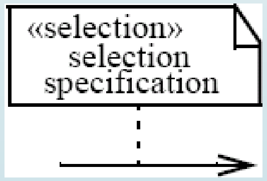
In this example, the transformation gets the value of the attribute Customer of the Order object

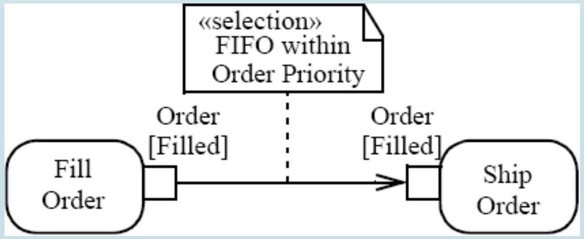
Selection

- Specifies the order (FIFO, LIFO or modeler-defined ordering) in which tokens in the node are offered to the outgoing edges
- Can be applied to:
  - ***Object node*** - specifies the object node ordering, choosing what token offers to the outgoing edge whenever it asks a token
  - ***Edge*** - chooses the order on which tokens are offered from the source object node to the edge (overrides any selection present on the object node, that is object node ordering)



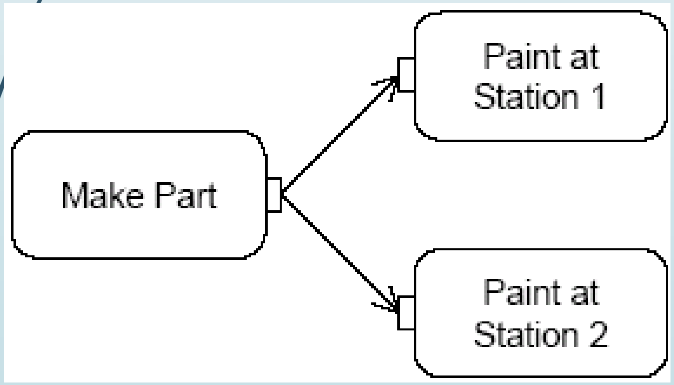






Token competition

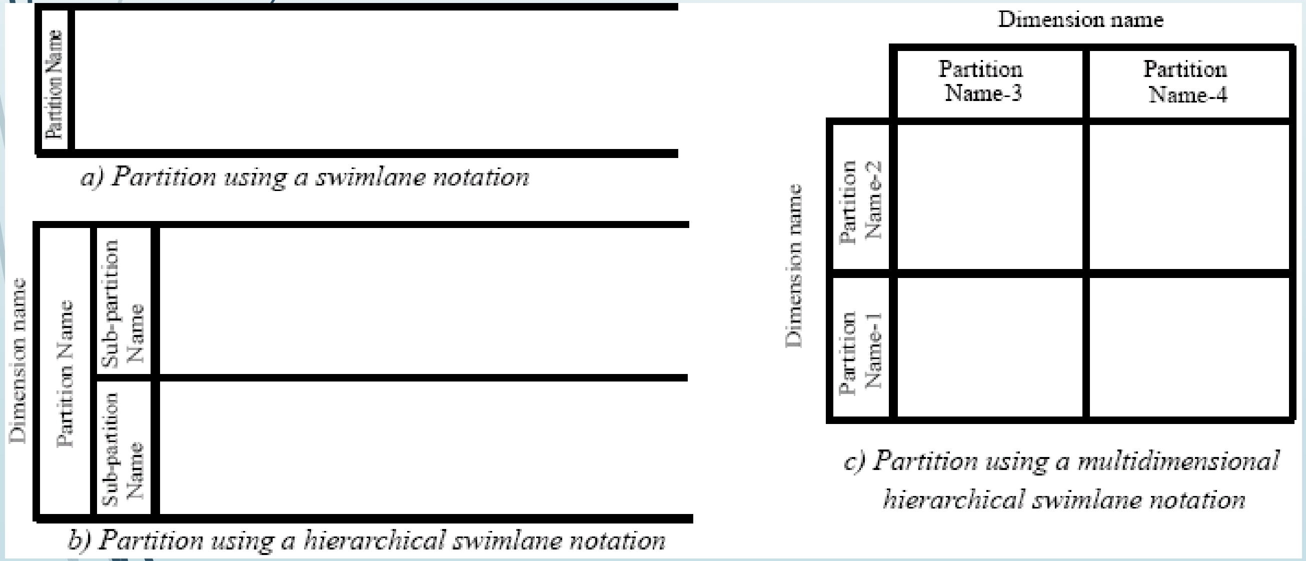
- A parameter node or pin may have multiple edges coming out of it, whereupon there will be competition for its tokens, because object nodes cannot duplicate tokens while forks can
- Then there is *indeterminacy* in the movement of data in the graph

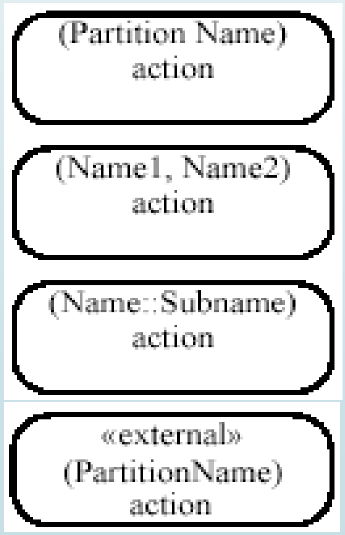


If the input pin of *Paint at Station 1* is full, the token remains at the output of *Make Part* until the traversal can be completed to one of the input pins

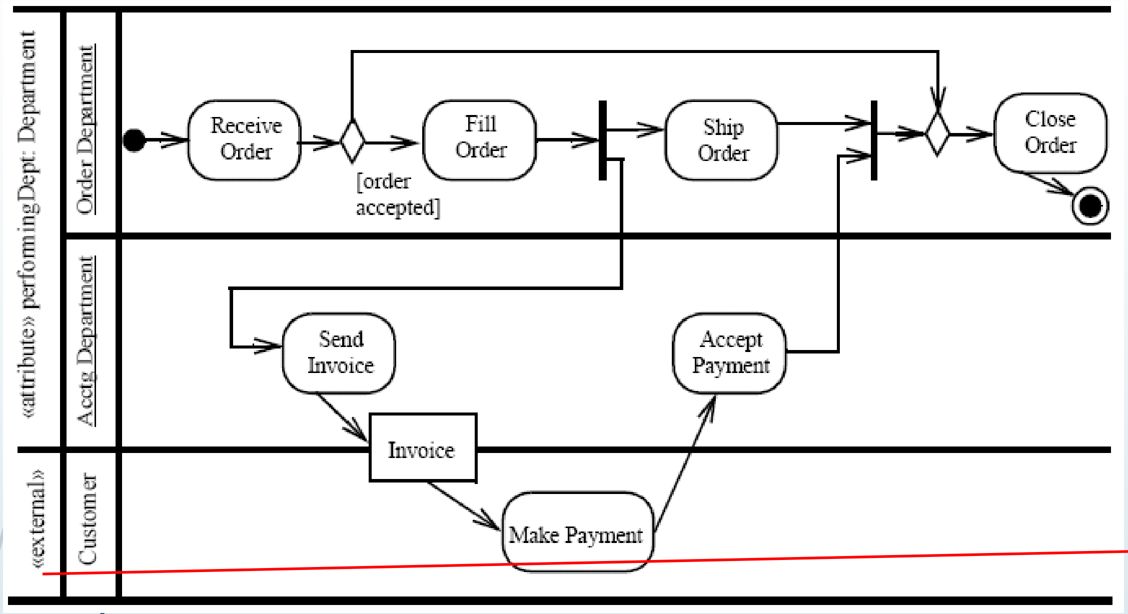
ActivityPartition

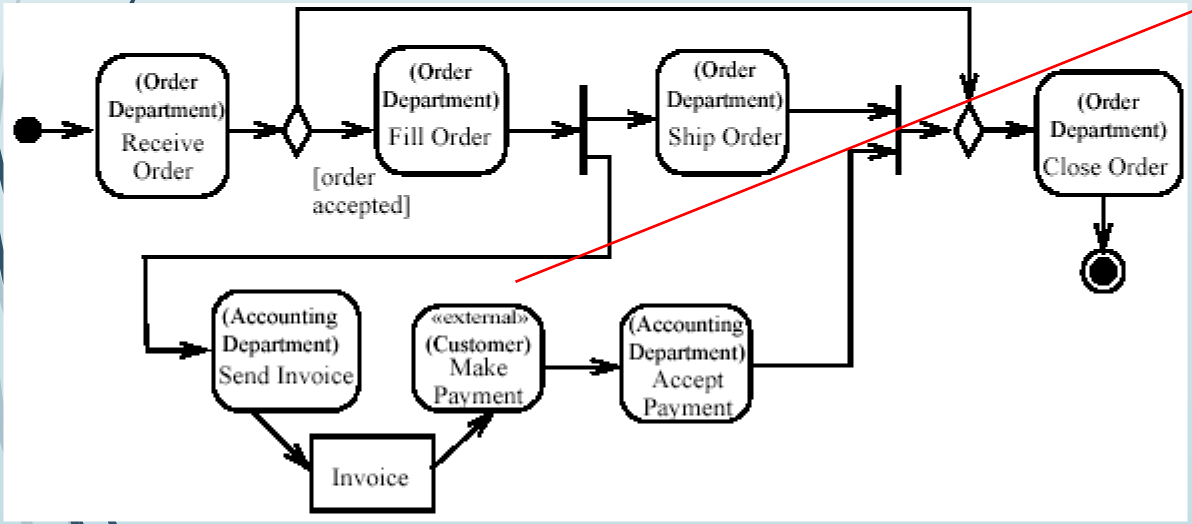
- Partitions divide the nodes and edges for identifying actions that have some characteristics in common
- They often correspond to organizational units in a business model
- Partitions can be hierarchical and multidimensional
- Additional notation is provided: placing the partition name in parenthesis above the activity name





ActivityPartition

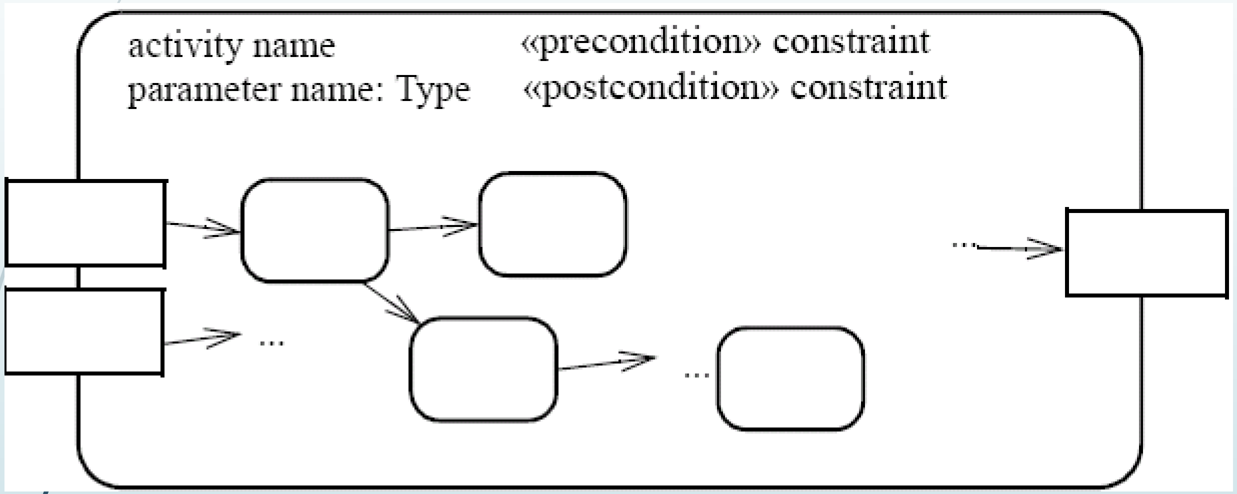


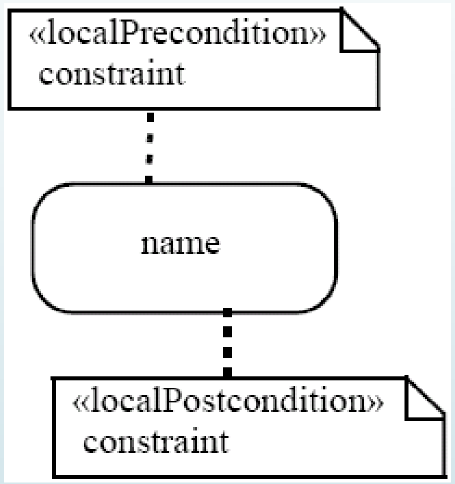

(Order Department) Receive Order
[order accepted]
(Order Department) Fill Order
(Order Department) Ship Order
(Order Department) Close Order
(Accounting Department) Send Invoice
Invoice
«external» (Customer) Make Payment
(Accounting Department) Accept Payment

Partition notated to occur outside the primary concern of the model

Pre & post condition

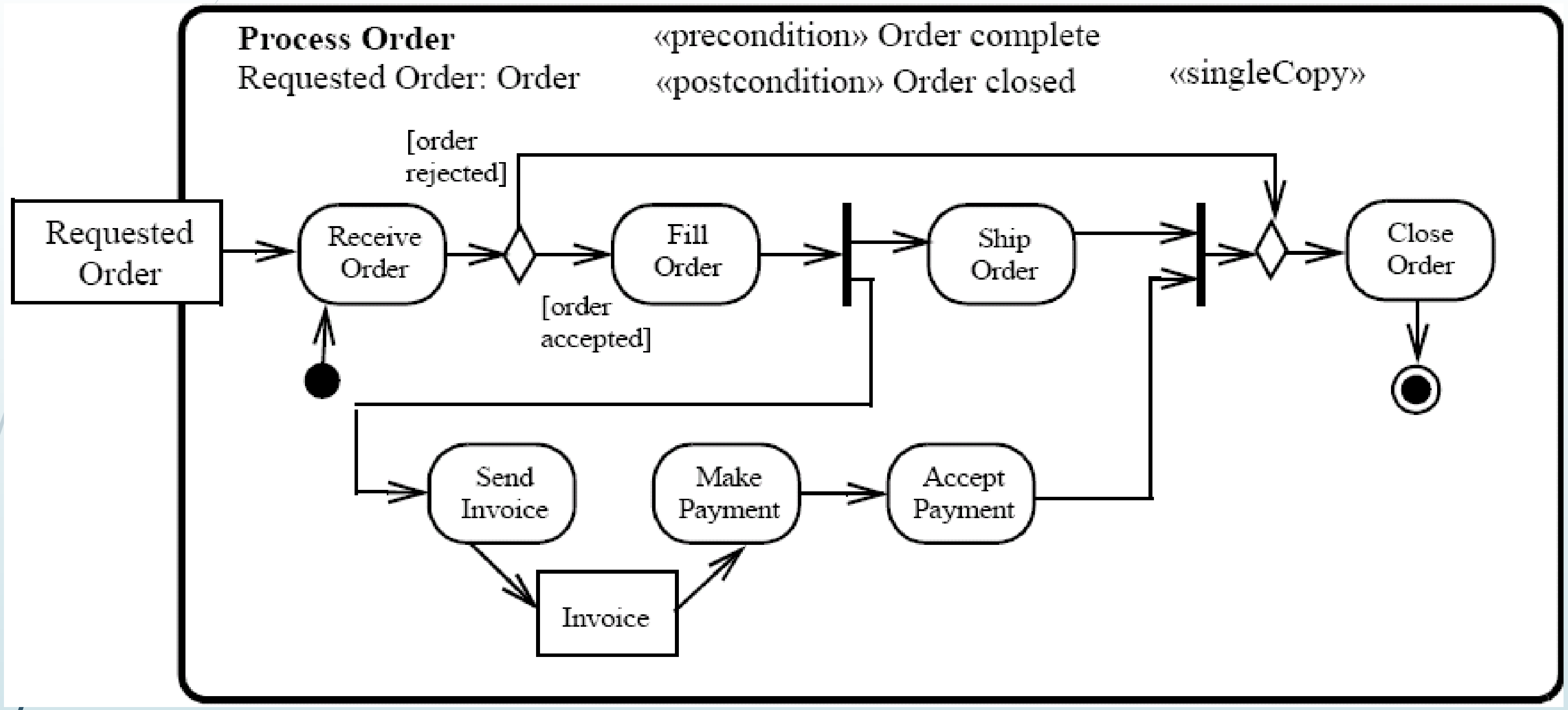
- Can be referred to an activity or to an action (local condition)

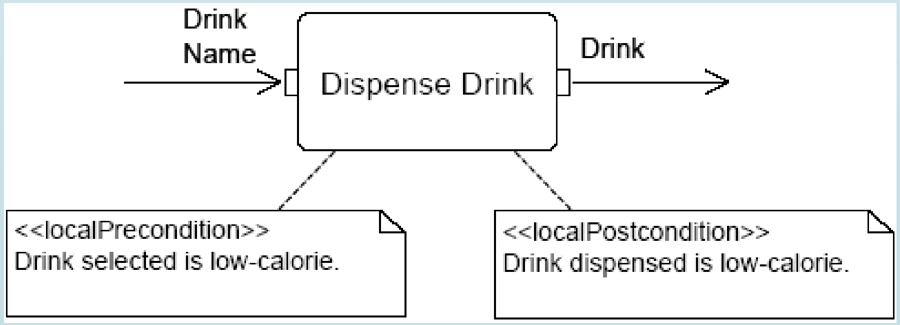



«localPrecondition» constraint
name
«localPostcondition» constraint

- UML intentionally does not specify when or whether pre/post conditions are tested (design time, runtime, etc.)
- UML also does not define what the runtime effect of a failed pre/post condition should be (error that stops execution, warning, no action)

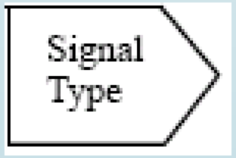
Pre & post condition

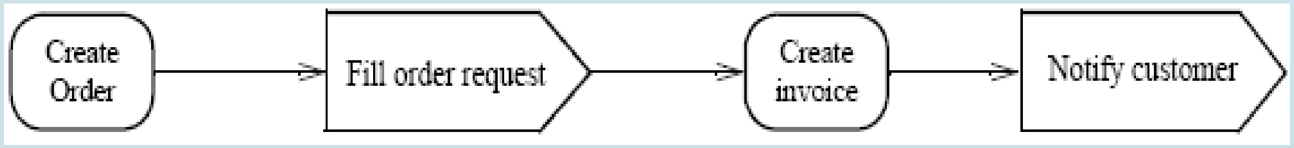




Send Signal Action

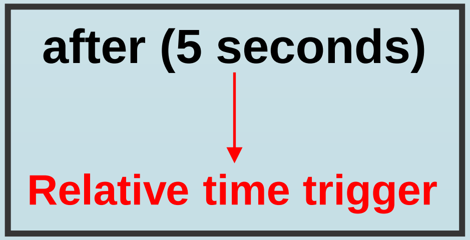
- Creates a signal instance from its inputs, and transmits it to the target object (local or remote)
- A signal is an asynchronous stimulus that triggers a reaction in the receiver in an asynchronous way and without a reply
- Any reply message is ignored

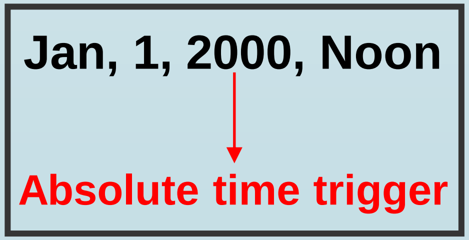




Time triggers and Time events

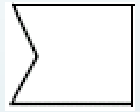
- A *Time trigger* is a trigger that specifies when a time event will be generated
- *Time events* occur at the instant when a specified point in time has transpired
- This time may be relative or absolute
  - ***Relative time trigger***: is specified with the keyword 'after' followed by an expression that evaluates to a time value
  - ***Absolute time trigger***: is specified as an expression that evaluates to a time value

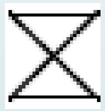


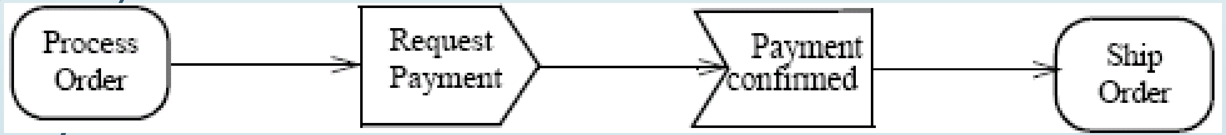


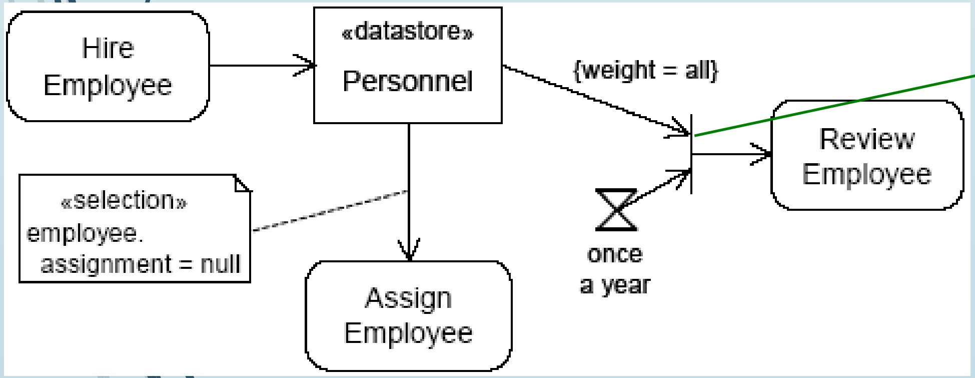
AcceptEventAction

- Waits for the occurrence of an event meeting specified conditions
- Two kinds of AcceptEventAction:
  - **Accept event action** – accepts signal events generated by a SendSignalAction
  - **Wait time action** – accepts time events

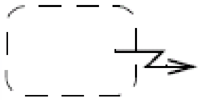






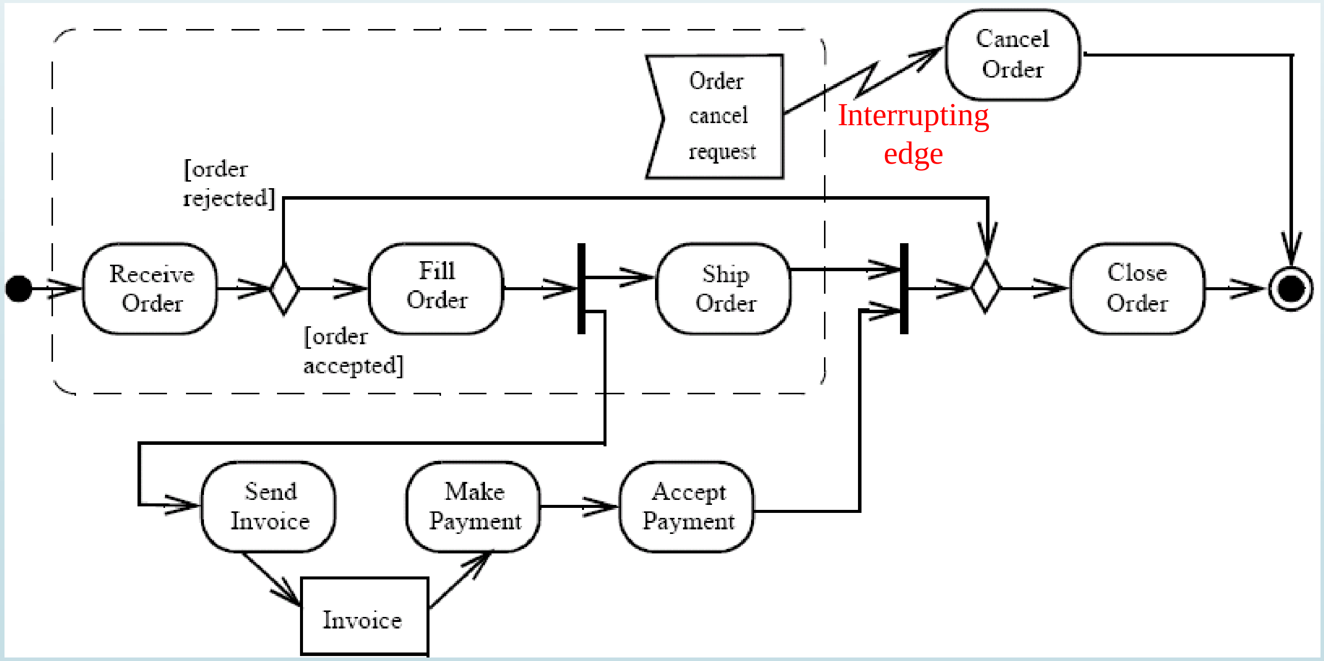


The objects stored in Personnel are only retrieved when the join succeeds (only once a year)



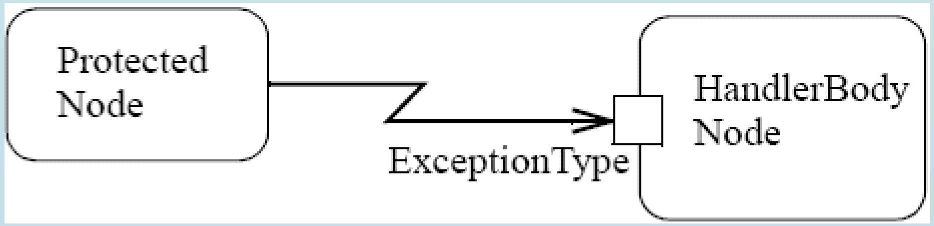
InterruptibleActivityRegion

- Is an activity group (sets of nodes and edges) that supports termination of tokens flowing into it
- When a token leaves an interruptible region via interrupting edges, all tokens and behaviours in the region are terminated
- Token transfer is still atomic: a token transition is never partial; it is either complete or it does not happen at all (also for internal stream)



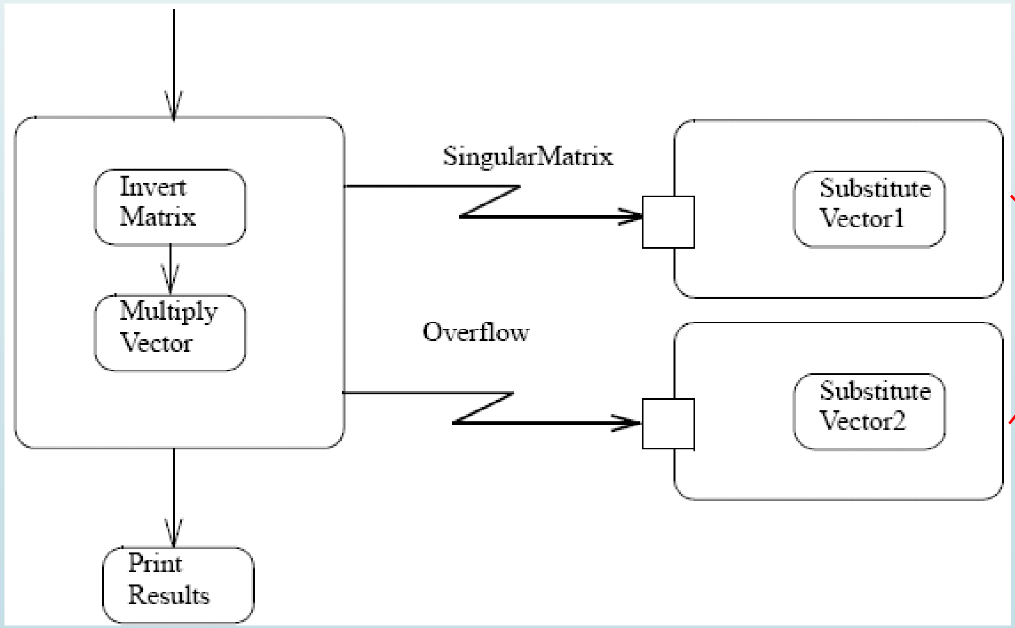
Exceptions

- Exception handler - specifies the code to be executed when the specified exception occurs during the execution of the protected node
- When an exception occurs the set of execution handlers on the action is examined to look for a handler that matches (*catches*) the exception
- If the exception is not caught, it is propagated to the enclosing protected node, if one exists
- If the exception propagates to the topmost level of the system and is not caught, the behaviour of the system is unspecified; profiles may specify what happens in such cases



Exceptions

- When an exception is caught, is executed the exception body instead of the protected node, and then the token is passed to all the edges that go out from that protected node
- The exception body has no explicit input or output edges
- Exception body can resolve the problems that have caused the exception or can abort the program
- We can put any activities nested in a protected node (in UML 2.0, nesting activities is allowed)



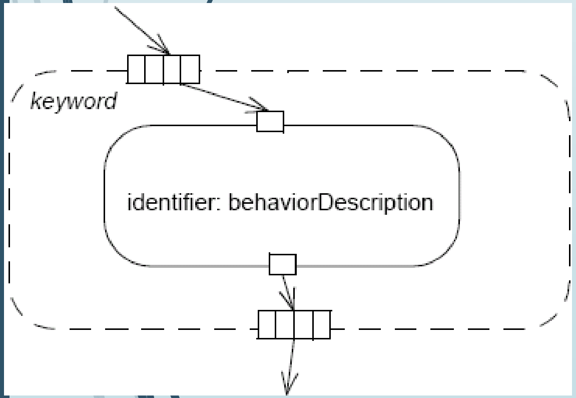
Protected node with two nested activities

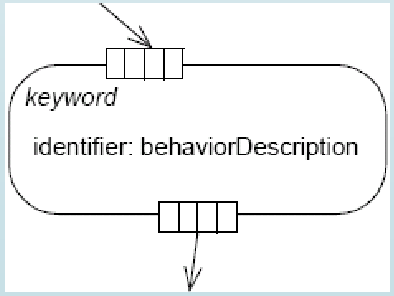
HandlerBody node

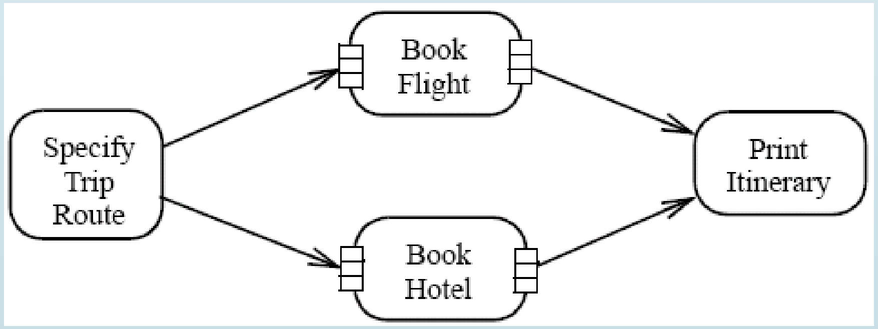
Successful end

ExpansionRegion

- Nested region of an activity in which each input is a collection of values
- The expansion region is executed once for each element (or position) in the input collection
- On each execution of the region, an output value from the region is inserted into an output collection at the same position as the input elements







ExpansionRegion

- There are three ways of interaction between the executions:
  - **Parallel** (concurrent): all the interactions are independent
  - **Iterative**: the interactions occur in the order of the elements (the executions of the region must happen in sequence, with one finishing before another can begin)
  - **Stream** (streaming): there is a single execution of the region, where the values in the input collection are extracted and placed into the execution of the expansion region as a stream (in order if the collection is ordered)

Component Diagram

Introduction

- UML component diagrams describe software components and their dependencies to each others
  - A component is an **autonomous** unit within a system
  - The components can be used to define software systems of arbitrary size and complexity
  - UML component diagrams enable to model the high-level software components, and the interfaces to those components
  - Important for component-based development (CBD)
  - Component and subsystems can be flexibly REUSED and REPLACED
  - A dependency exists between two elements if changes to the definition of one element may cause changes to the other
  - Component Diagrams are often referred to as “wiring diagrams”
  - The wiring of components can be represented on diagrams by means of components and dependencies between them

Introduction

- UML components diagrams are **structure diagrams**
  - A type of diagram that depicts the elements of a specification that are irrespective of time. This includes class, composite structure, component, deployment
- UML components diagrams are **implementation diagrams**
  - Implementation diagrams describe the different elements required for implementing a system

Component in UML 2.0

- Modular unit with well-defined interfaces that is replaceable within its environment
- **Autonomous** unit within a system
  - Has one or more provided and required interfaces
  - Its internals are hidden and inaccessible
  - A component is encapsulated
  - Its dependencies are designed such that it can be treated as independently as possible

Component ELEMENTS

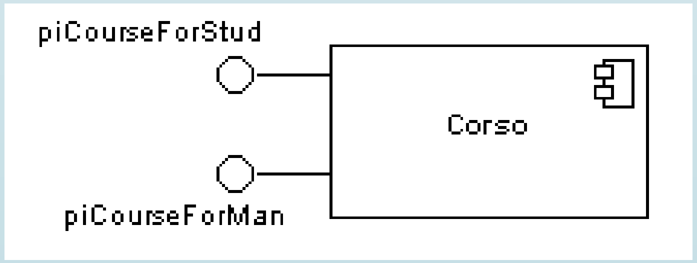
- Interfaces
  - An interface represents a declaration of a set of operations and obligations
- Usage dependencies
  - A usage dependency is relationship which one element requires another element for its full implementation
- Ports
  - Port represents an interaction point between a component and its environment
- Connectors
  - Connect two components
  - Connect the external contract of a component to the internal structure
  - Two kinds of connectors : Delegation and Assembly

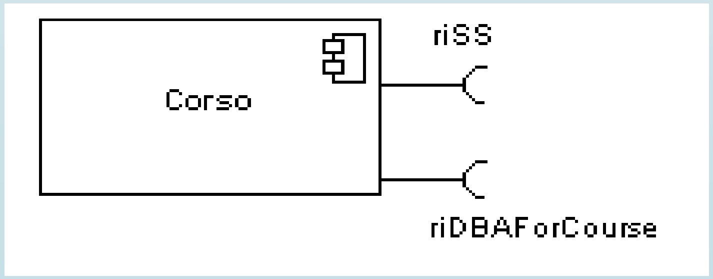
Interface

- A component defines its behaviour in terms of provided and required interfaces
- An interface
  - Is the definition of a collection of one or more operations
  - Provides only the operations but not the implementation
  - Implementation is normally provided by a class/ component
  - In complex systems, the physical implementation is provided by a group of classes rather than a single class

Interface

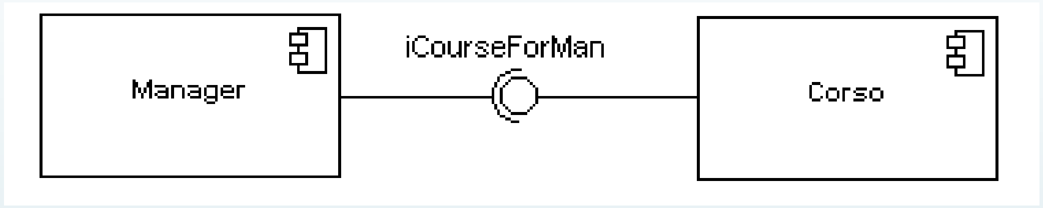
- A **provided** interface
  - Characterize services that the component offers to its environment
  - Is modeled using a ball, labelled with the name, attached by a solid line to the component
- A **required** interface
  - Characterize services that the component expects from its environment
  - Is modeled using a socket, labelled with the name, attached by a solid line to the component



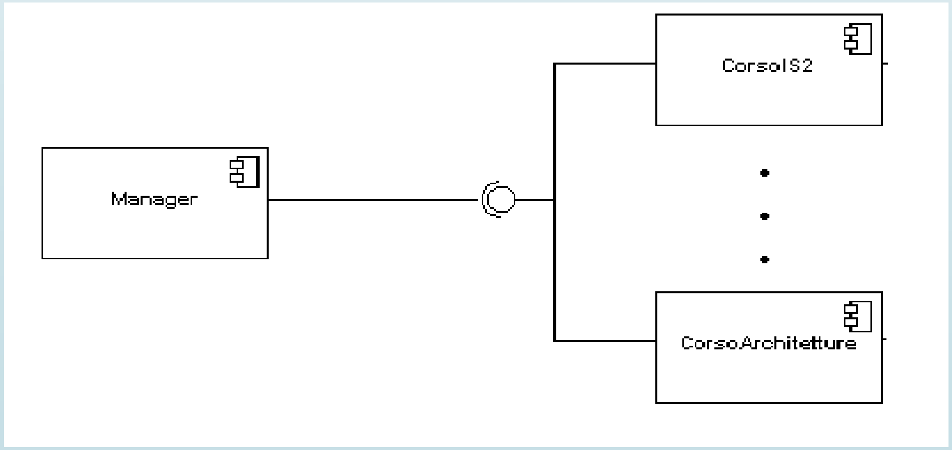


Interface

- Where two components/classes provide and require the same interface, these two notations may be combined

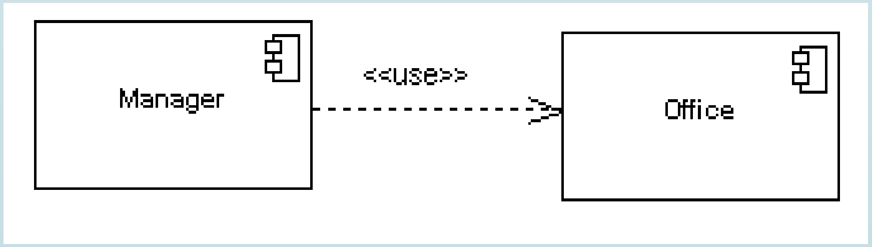


- In a system context where there are multiple components that require or provide a particular interface, a notation abstraction can be used that combines by joining the interfaces

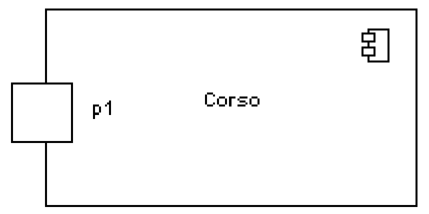


Dependencies

- Usage Dependency
  - A usage dependency is relationship which one element requires another element for its full implementation
  - Is a dependency in which the client requires the presence of the supplier
  - Is shown as dashed arrow with a <<use>> keyword
  - The arrowhead point from the dependent component to the one of which it is dependent



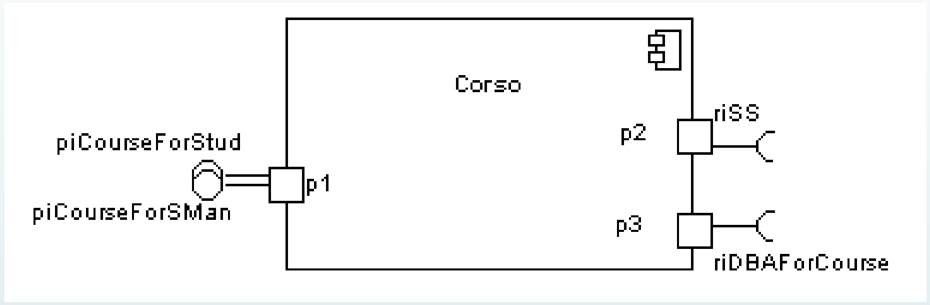
Port



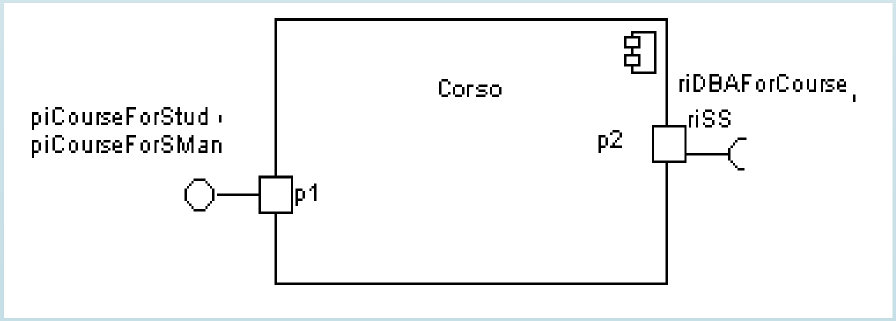
- Specifies a distinct interaction point
  - Between that component and its environment
  - Between that component and its internal parts
- Is shown as a small square symbol
- Ports can be named, and the name is placed near the square symbol
- Is associated with the interfaces that specify the nature of the interactions that may occur over a port
- All interactions of a component with its environment are achieved through a port
- The internals are fully isolated from the environment
- This allows such a component to be used in any context that satisfies the constraints specified by its ports

Port

- Ports can support unidirectional communication or bi-directional communication.

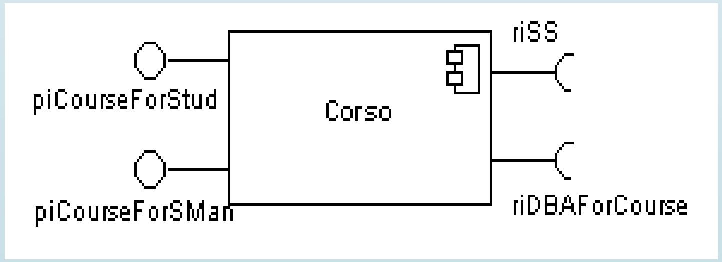


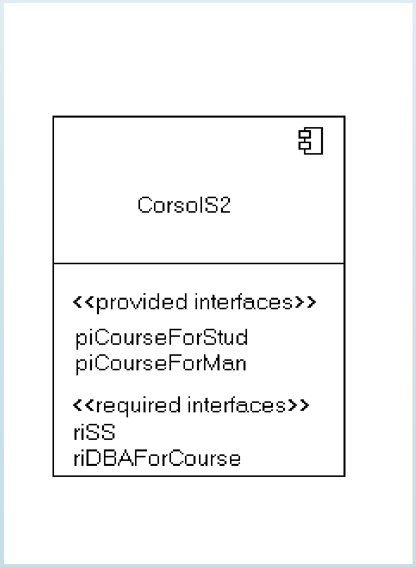
- If there are multiple interfaces associated with a port, these interfaces may be listed with the interface icon, separated by a commas



External view

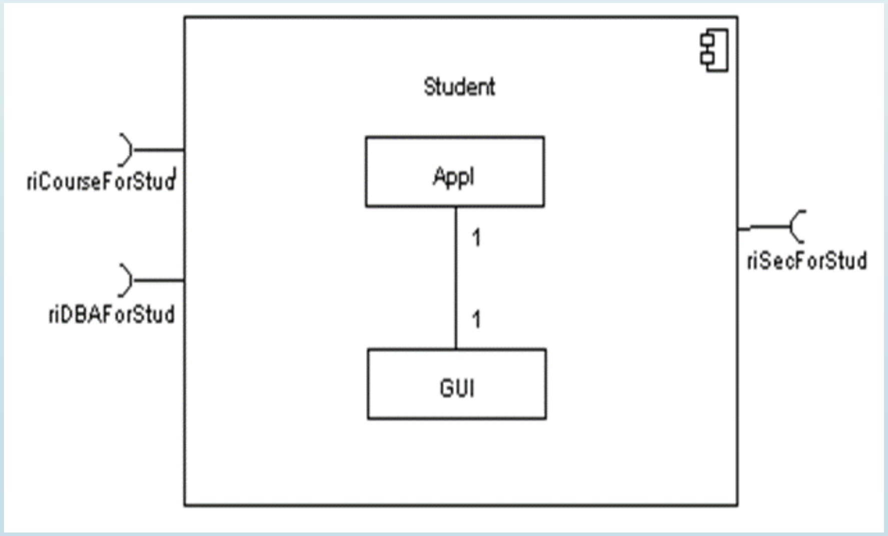
- A component have an external view and an internal view
- An external view (or black box view) shows publicly visible properties and operations
- An external view of a component is by means of interface symbols sticking out of the component box
- The interface can be listed in the compartment of a component box





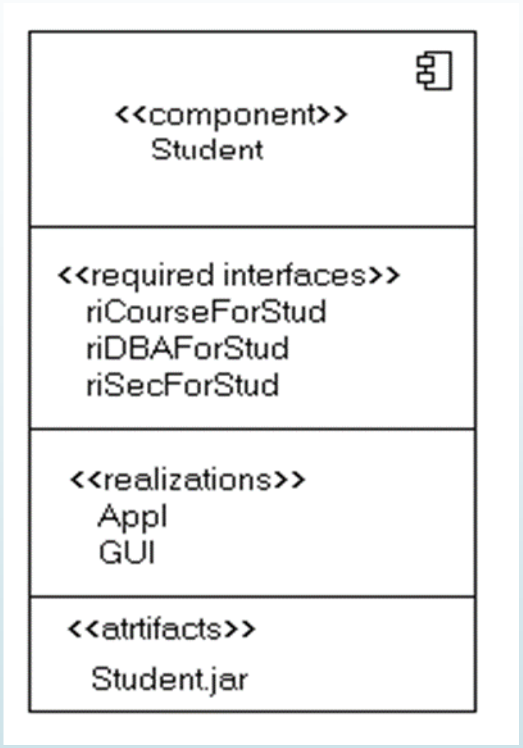
Internal view

- An internal, or white box view of a component is where the realizing classes/components are nested within the component shape
- Realization is a relationship between two set of model elements
  - One represents a specification
  - The other represent an implementation of the latter



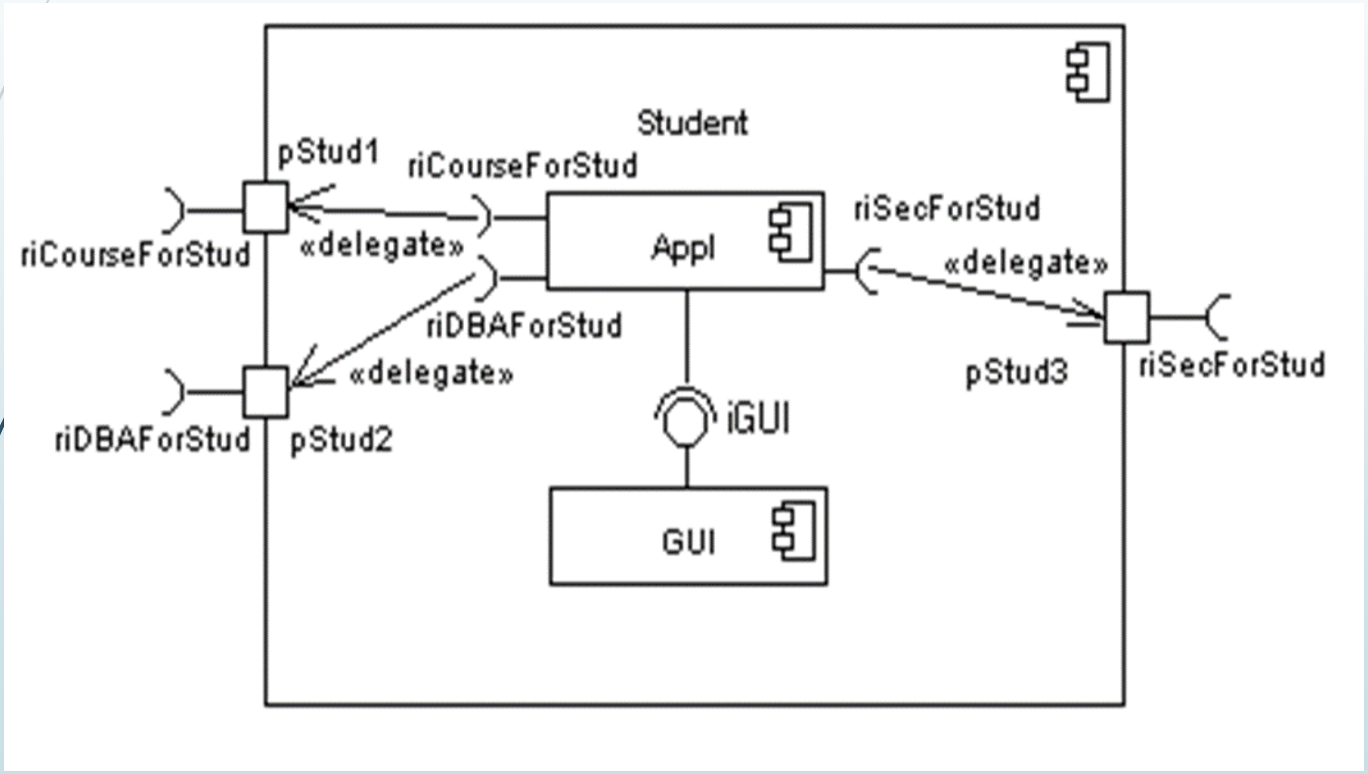
Internal view

- The internal class that realize the behavior of a component may be displayed in an additional compartment
- Compartments can also be used to display parts, connectors or implementation artifacts
- An artifact is the specification of a phisycal piece of information



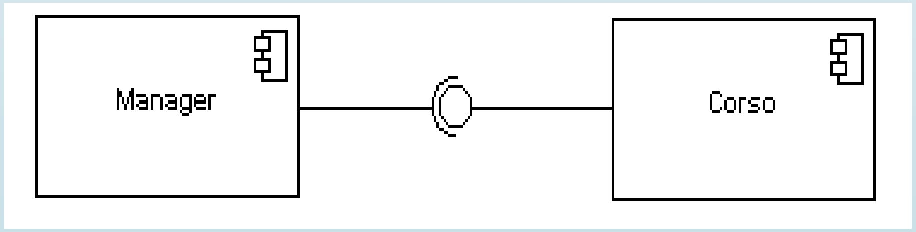
Internal view

- Components can be built recursively



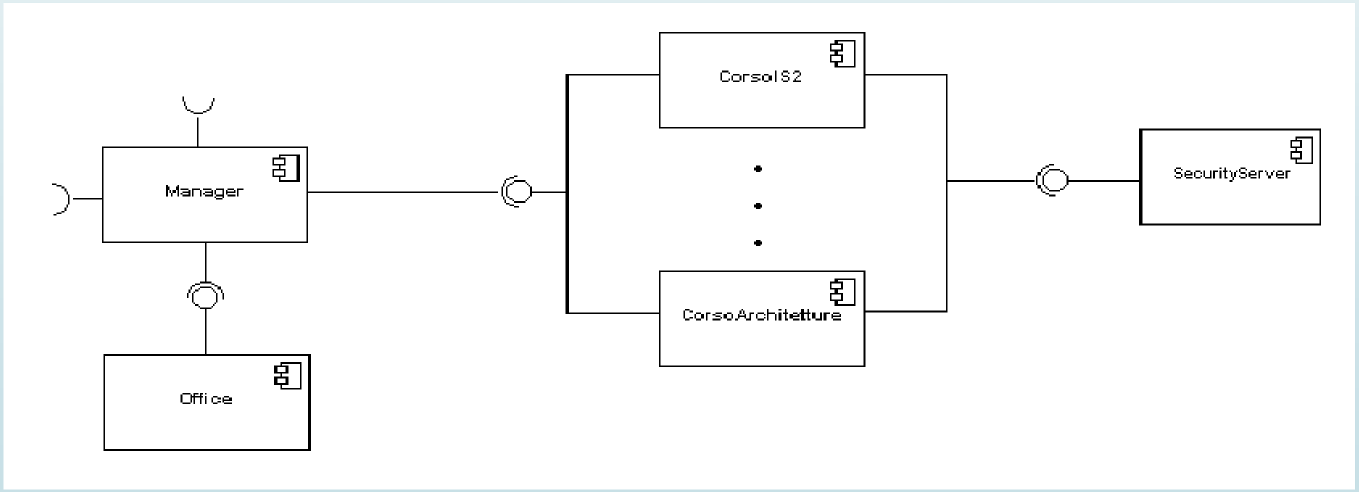
Assembly Connectors

- A connector between 2 components defines that one component provides the services that another component requires
- He must only be defined from a required interface to a provided interface
- An assembly connector is notated by a “ball-and-socket” connection



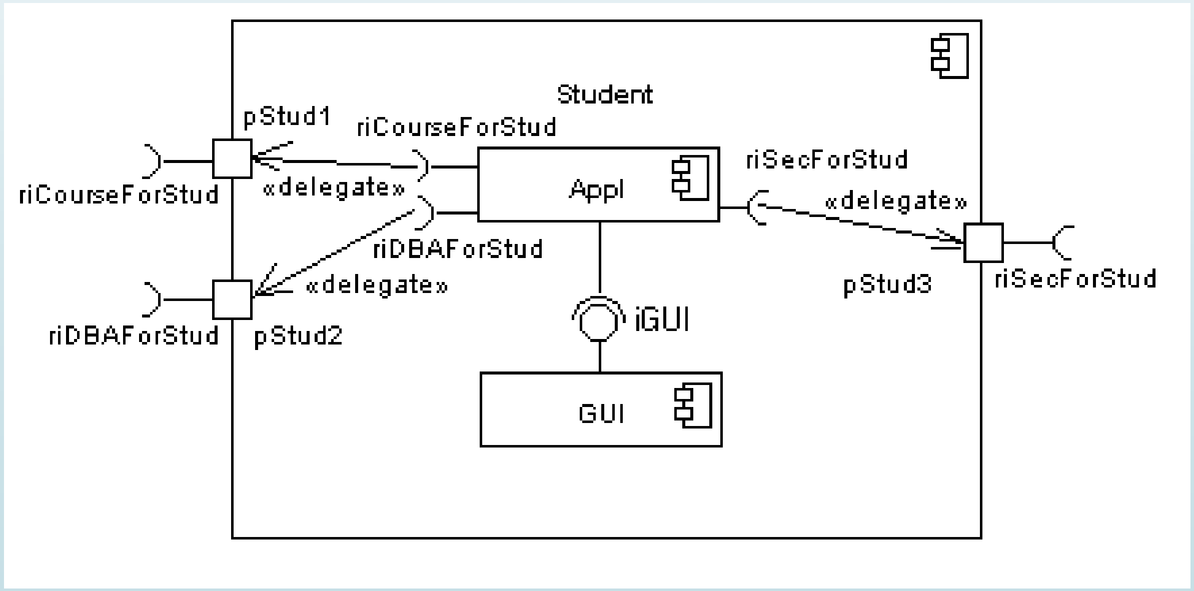
Assembly Connectors

- The semantics for an assembly connector
  - Are that signals travel along an instance of a connector originating in a required port and delivered to a provided port
  - The interfaces provided and required must be compatible
  - The interface compatibility between provided and required ports that are connected enables an existing component in a system to be replaced



Delegation Connectors

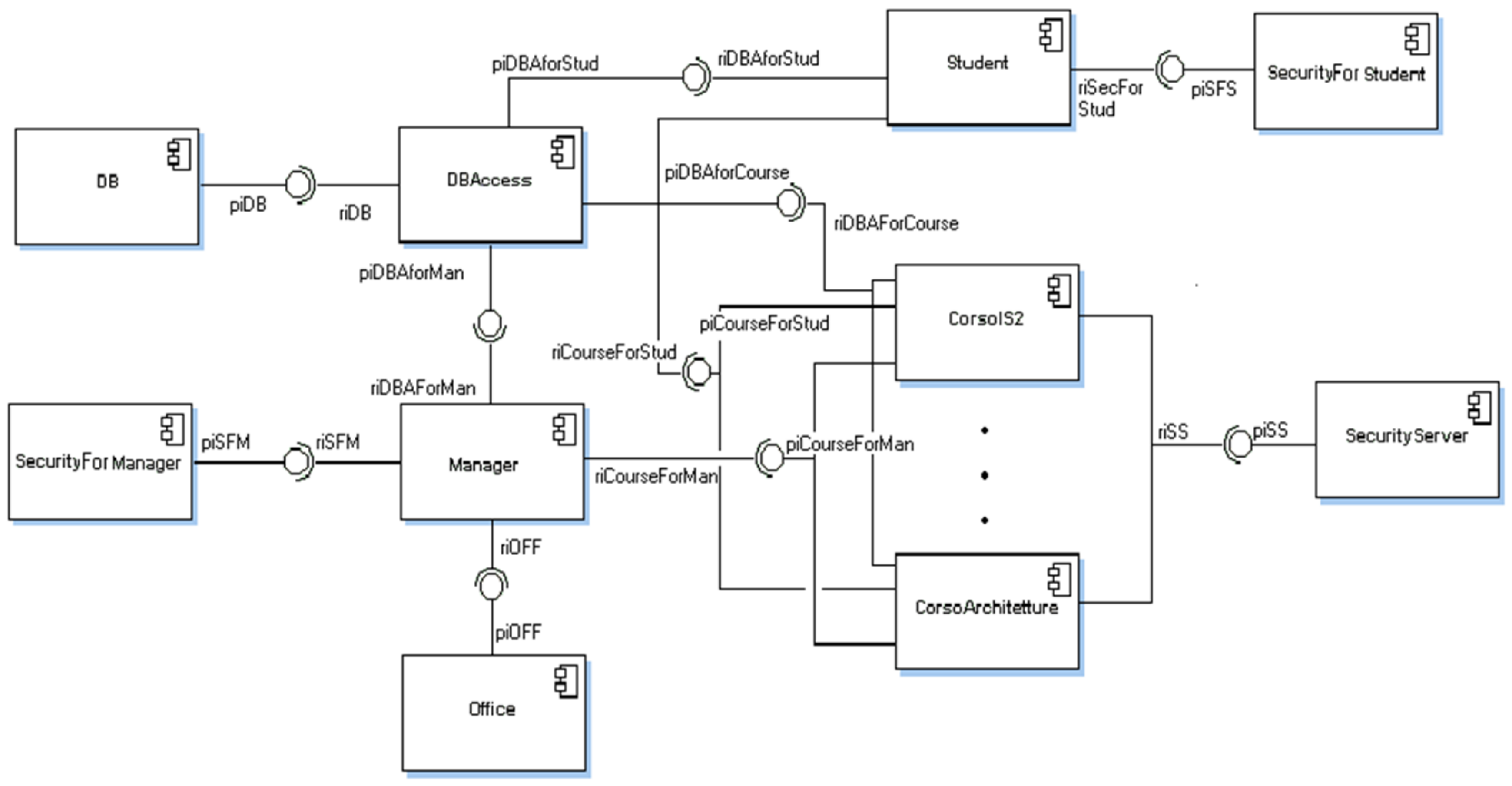
- Links the external contract of a component to the internal realization
- Represents the forwarding of signals
- He must only be defined between used interfaces or ports of the same kind



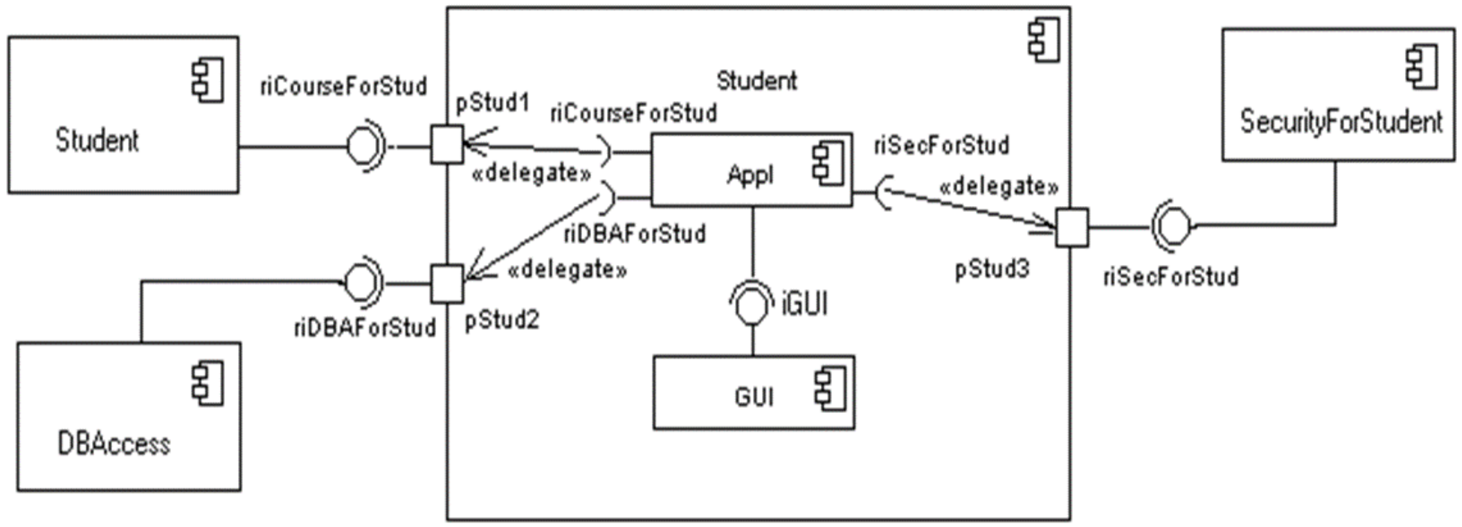
Case Study

- Development of an application collecting students' opinions about courses
- A student can
  - Read
  - Insert
  - Update
  - Make data permanent about the courses in its schedule
- A professor can only see statistic elaboration of the data
- The student application must be installed in pc client   (sw1, sw2)
- The manager application must be installed in pc client (in the manager's office)
- There is one or more servers with DataBase and components for courses management

Case Study


Student
SecurityFor Student
piDBAforStud
riDBAforStud
riSecFor Stud
piSFS
DB
DBAccess
piDB
riDB
piDBAforCourse
riDBAForCourse
piDBAforMan
CorsoIS2
piCourseForStud
riCourseForStud
riDBAForMan
SecurityFor Manager
piSFM
riSFM
Manager
riCourseForMan
piCourseForMan
riSS
piSS
SecurityServer
riOFF
piOFF
CorsoArchitetture
Office

Case Study


Student
riCourseForStud
pStud1
riCourseForStud
Student
Appl
«delegate»
riDBAForStud
«delegate»
riSecForStud
«delegate»
pStud3
SecurityForStudent
riSecForStud
iGUI
GUI
riDBAForStud
pStud2
DBAccess

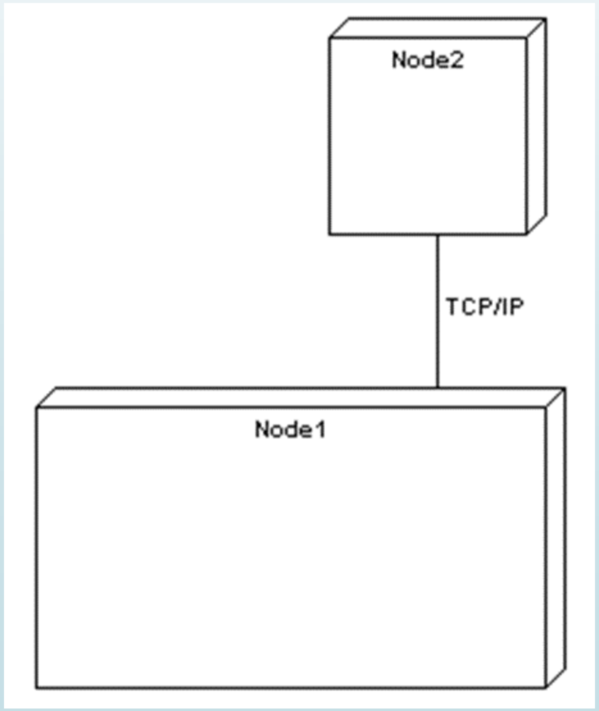
Deployment Diagrams

Deployment Diagrams

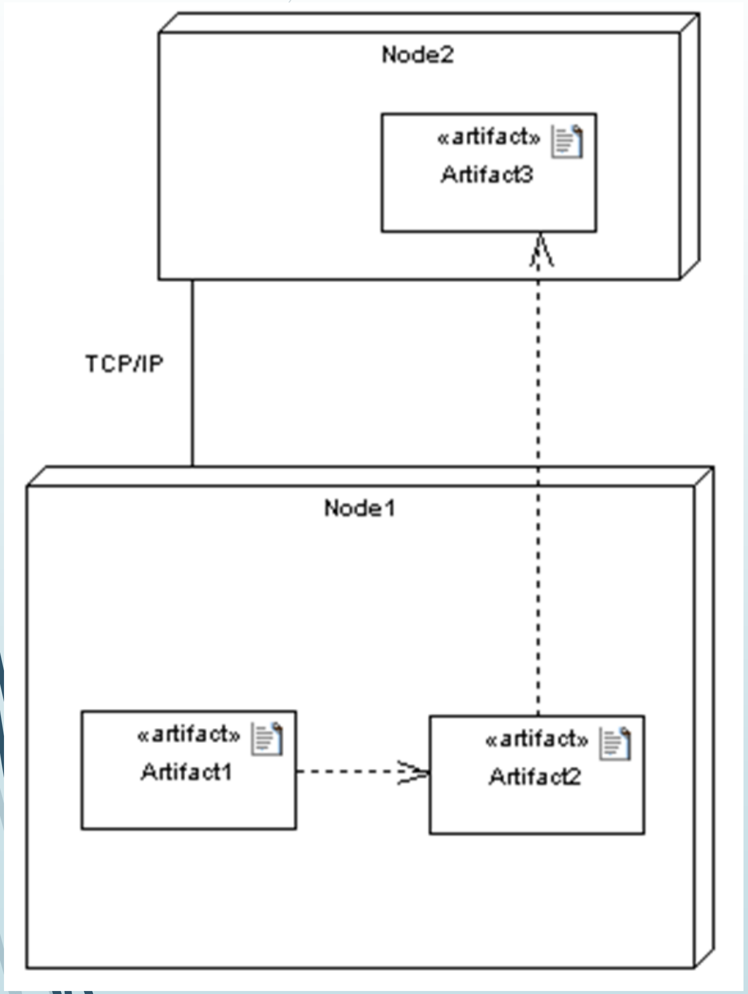
- There is a strong link between components diagrams and deployment diagrams
- Deployment diagrams
  - Show the physical relationship between hardware and software in a system
  - Hardware elements:
    - Computers (clients, servers)
    - Embedded processors
    - Devices (sensors, peripherals)
  - Are used to show the nodes where software components reside in the run-time system

Deployment Diagrams

- Deployment diagram
  - Contains nodes and connections
  - A node usually represent a piece of hardware in the system
  - A connection depicts the communication path used by the hardware to communicate
  - Usually indicates the method such as TCP/IP



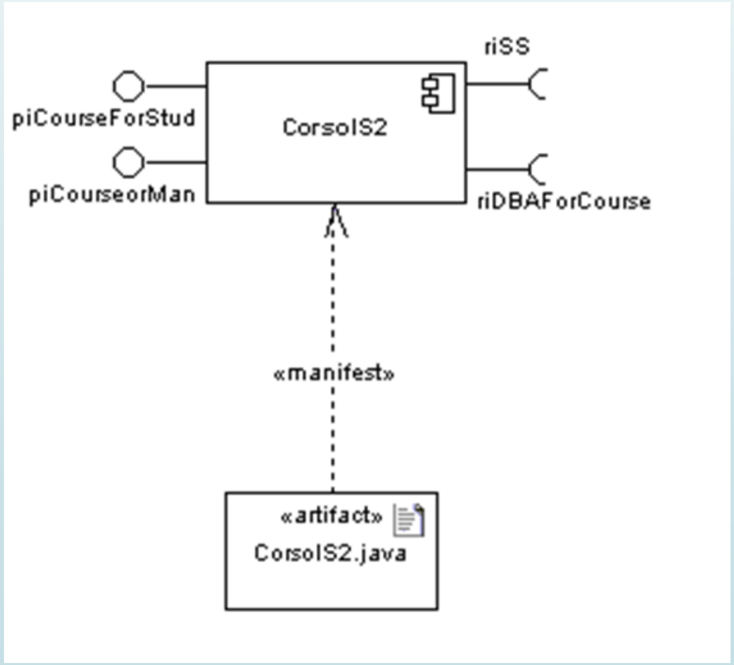
Deployment Diagrams



- Deployment diagrams contain artifact
- An artifact
  - Is the specification of a phisycal piece of information
  - Ex: source files, binary executable files, table in a database system,….
  - An artifact defined by the user represents a concrete element in the physical world

Deployment Diagrams

- An artifact manifest one or more model elements
- A <<manifestation>> is the concrete physical of one or more model elements by an artifact
- This model element often is a component
- A manifestation is notated as a dashed line with an open arrow-head labeled with the keyword <<manifest>>



Sequence Diagrams

Interaction Diagrams

- A series of diagrams describing the *dynamic behavior* of an object-oriented system.
  - A set of messages exchanged among a set of objects within a context to accomplish a purpose.
- Often used to model the way a use case is realized through a sequence of messages between objects.
- The purpose of Interaction diagrams is to:
  - Model interactions between objects
  - Assist in understanding how a system (a use case) actually works
  - Verify that a use case description can be supported by the existing classes
  - Identify responsibilities/operations and assign them to classes

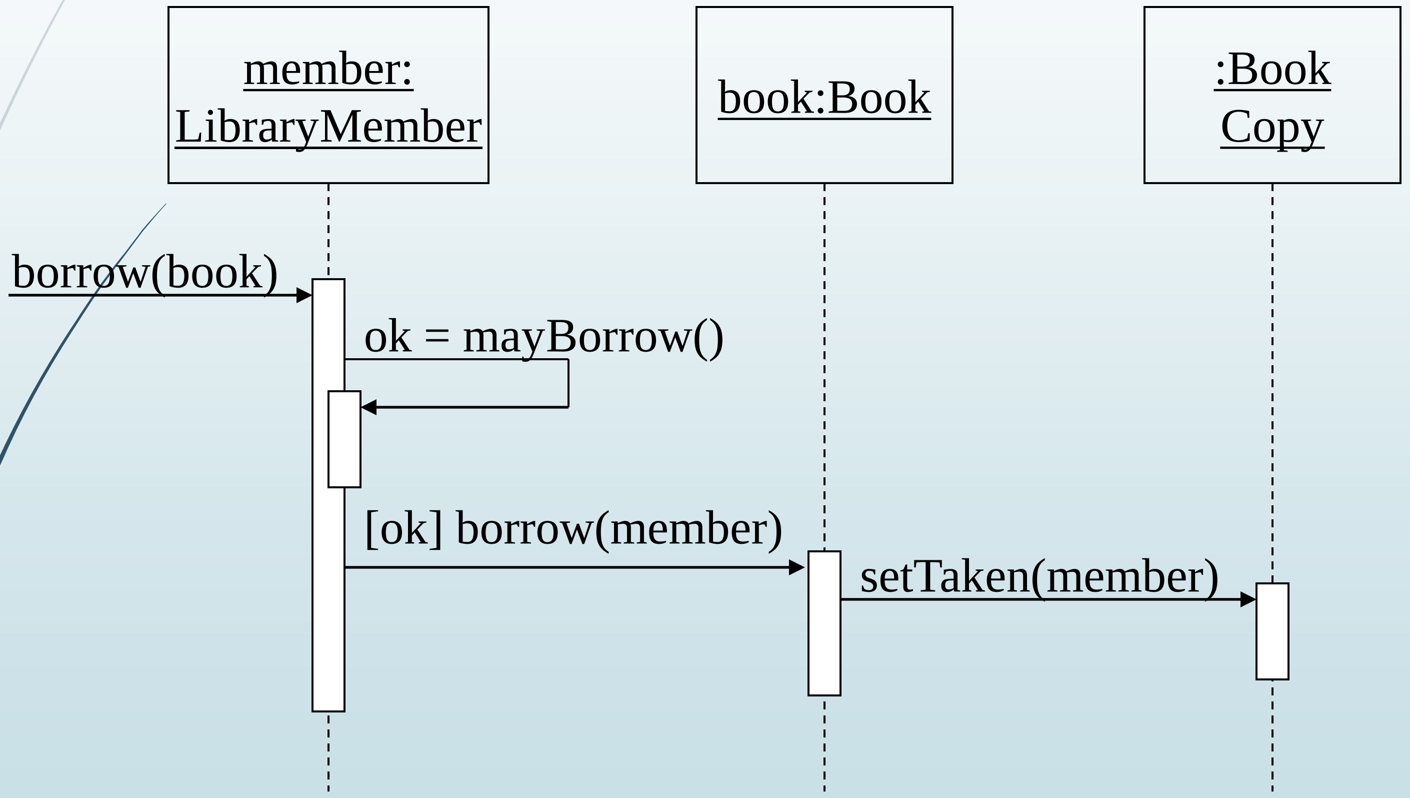
Interaction Diagrams

- UML
  - Collaboration Diagrams
    - Emphasizes structural relations between objects
  - Sequence Diagram
    - The subject of this tutorial

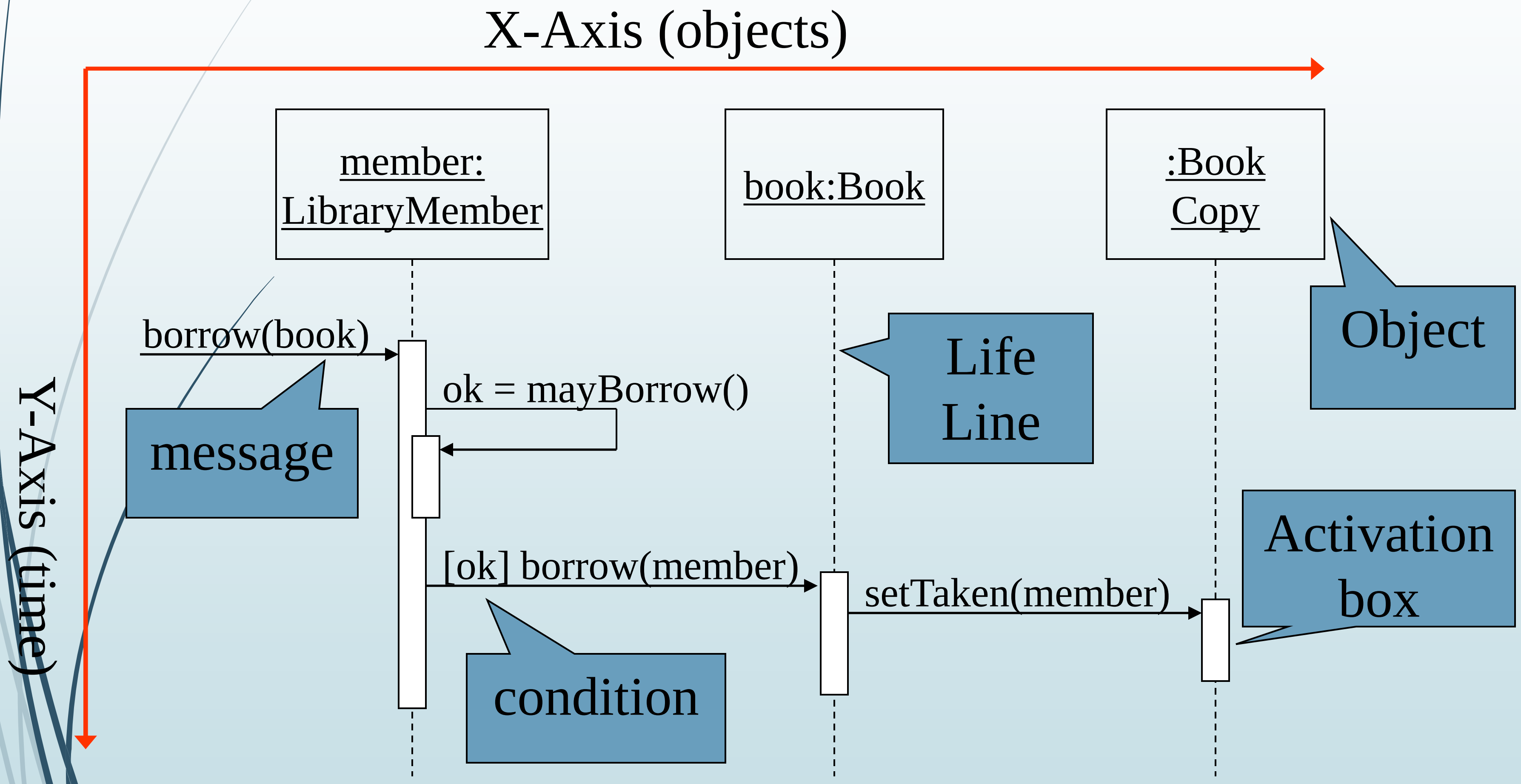
A First Look at Sequence Diagrams

- Illustrates how objects interacts with each other.
- Emphasizes time ordering of messages.
- Can model simple sequential flow, branching, iteration, recursion and concurrency.

A Sequence Diagram

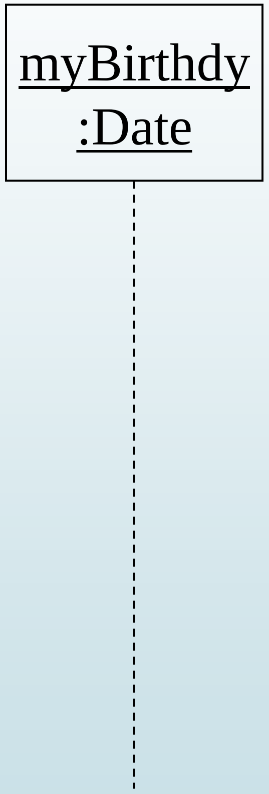


A Sequence Diagram



Object

- Object naming:
  - syntax: *[instanceName][:className]*
  - Name classes consistently with your class diagram (same classes).
  - Include instance names when objects are referred to in messages or when several objects of the same type exist in the diagram.
- The *Life-Line* represents the object's life during the interaction

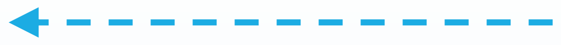


Messages

- An interaction between two objects is performed as a message sent from one object to another (simple operation call, Signaling, RPC)
- If object obj_1 sends a message to another object obj_2 some link must exist between those two objects (dependency, same objects)

- A message is represented by an arrow between the life lines of two objects.
  - Self calls are also allowed
  - The time required by the receiver object to process the message is denoted by an *activation-box.*
- A message is labeled at minimum with the message name.
  - Arguments and control information (conditions, iteration) may be included.

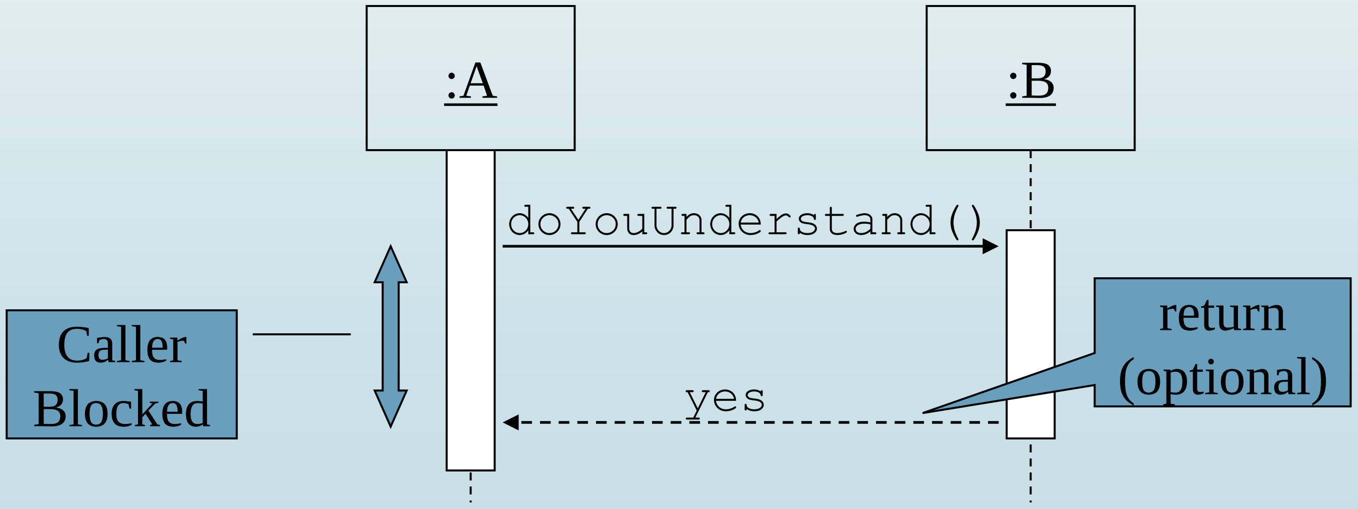
Return Values



- Optionally indicated using a dashed arrow with a label indicating the return value.
  - Don’t model a return value when it is obvious what is being returned, e.g. getTotal()
  - Model a return value only when you need to refer to it elsewhere, e.g. as a parameter passed in another message.
  - Prefer modeling return values as part of a method invocation, e.g. **`ok = isValid()`**

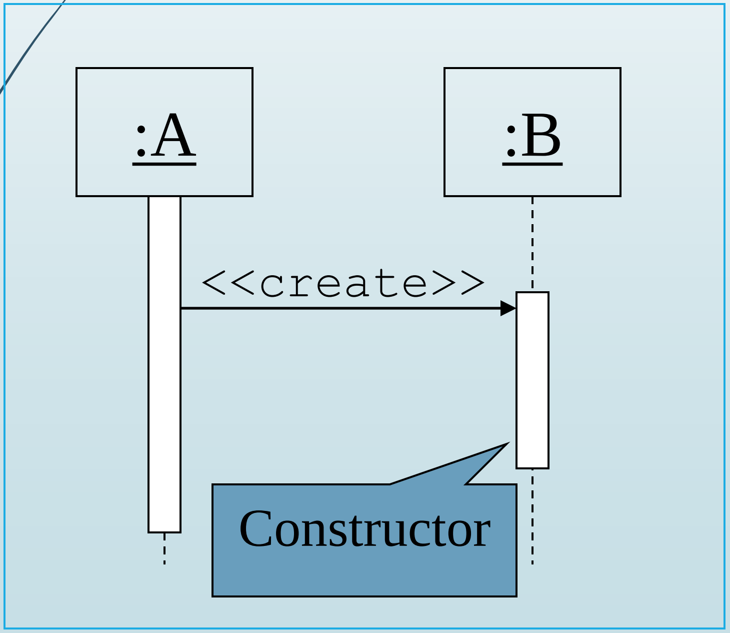
Synchronous Messages

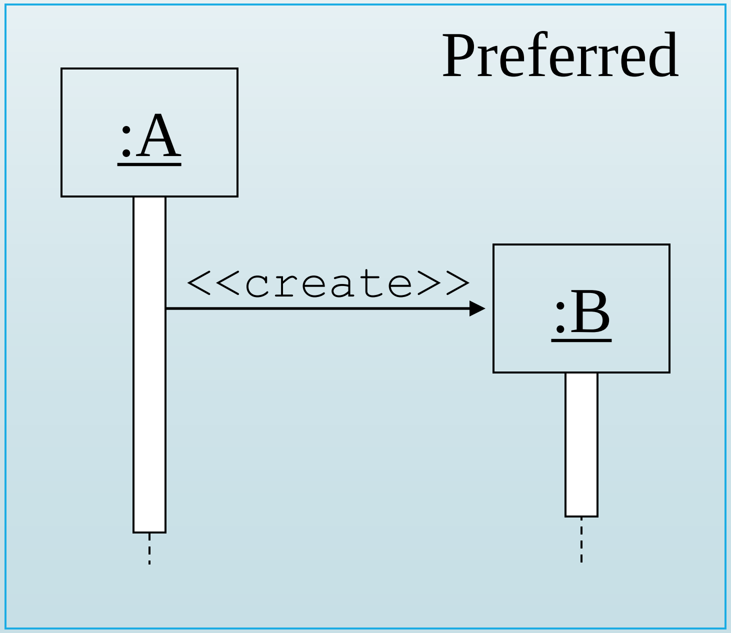
- Nested flow of control, typically implemented as an operation call.
  - The routine that handles the message is completed before the caller resumes execution.



Object Creation

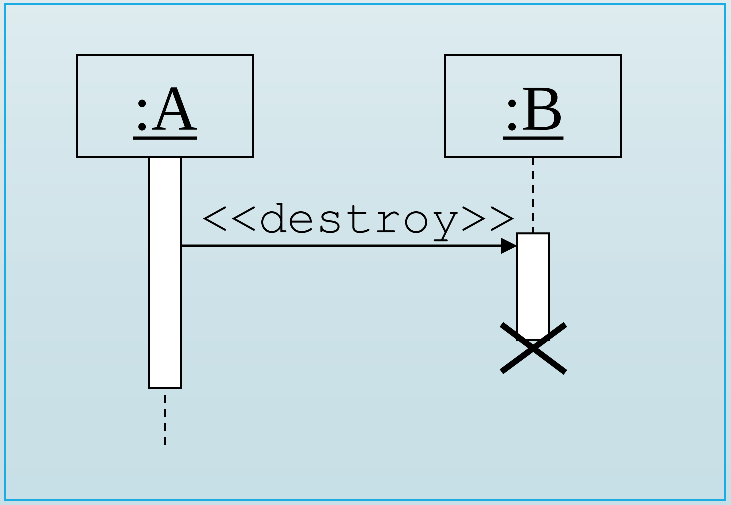
- An object may create another object via a **`<<create>>`** message.





Object Destruction

- An object may destroy another object via a **`<<destroy>>`** message.
  - An object may destroy itself.
  - Avoid modeling object destruction unless memory management is critical.

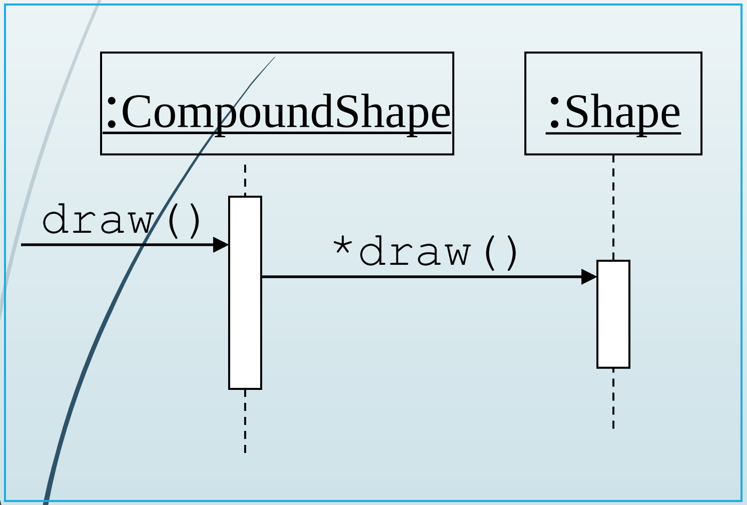


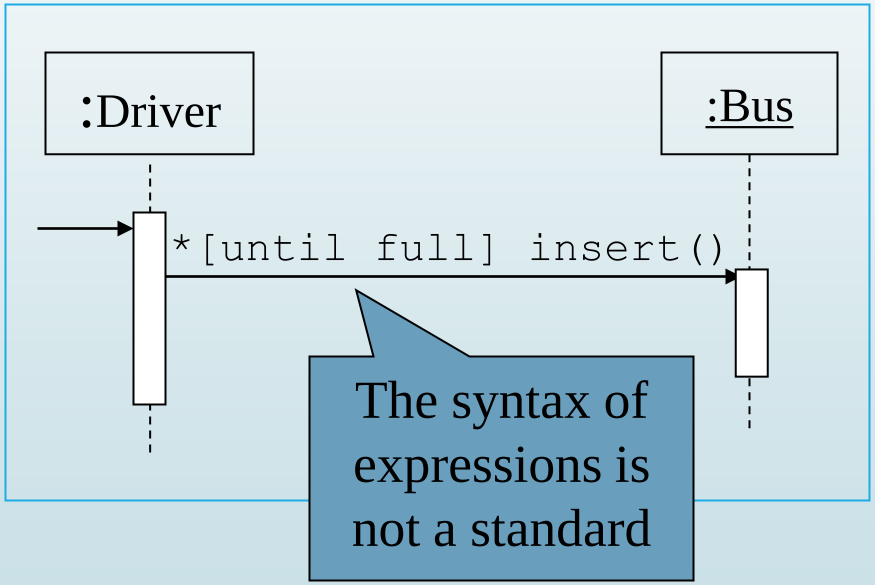
Control information

- Condition
  - syntax: ‘[‘ expression ’]’ message-label
  - The message is sent only if the condition is true
  - example: $\xrightarrow{\text{[ok] borrow(member)}}$
- Iteration
  - syntax: * [‘[‘ expression ‘]’] message-label
  - The message is sent many times to possibly multiple receiver objects.
- The control mechanisms of sequence diagrams suffice only for modeling simple alternatives.
  - Consider drawing several diagrams for modeling complex scenarios.
  - Don’t use sequence diagrams for detailed modeling of algorithms (this is better done using *activity diagrams, pseudo-code* or *state-charts*).

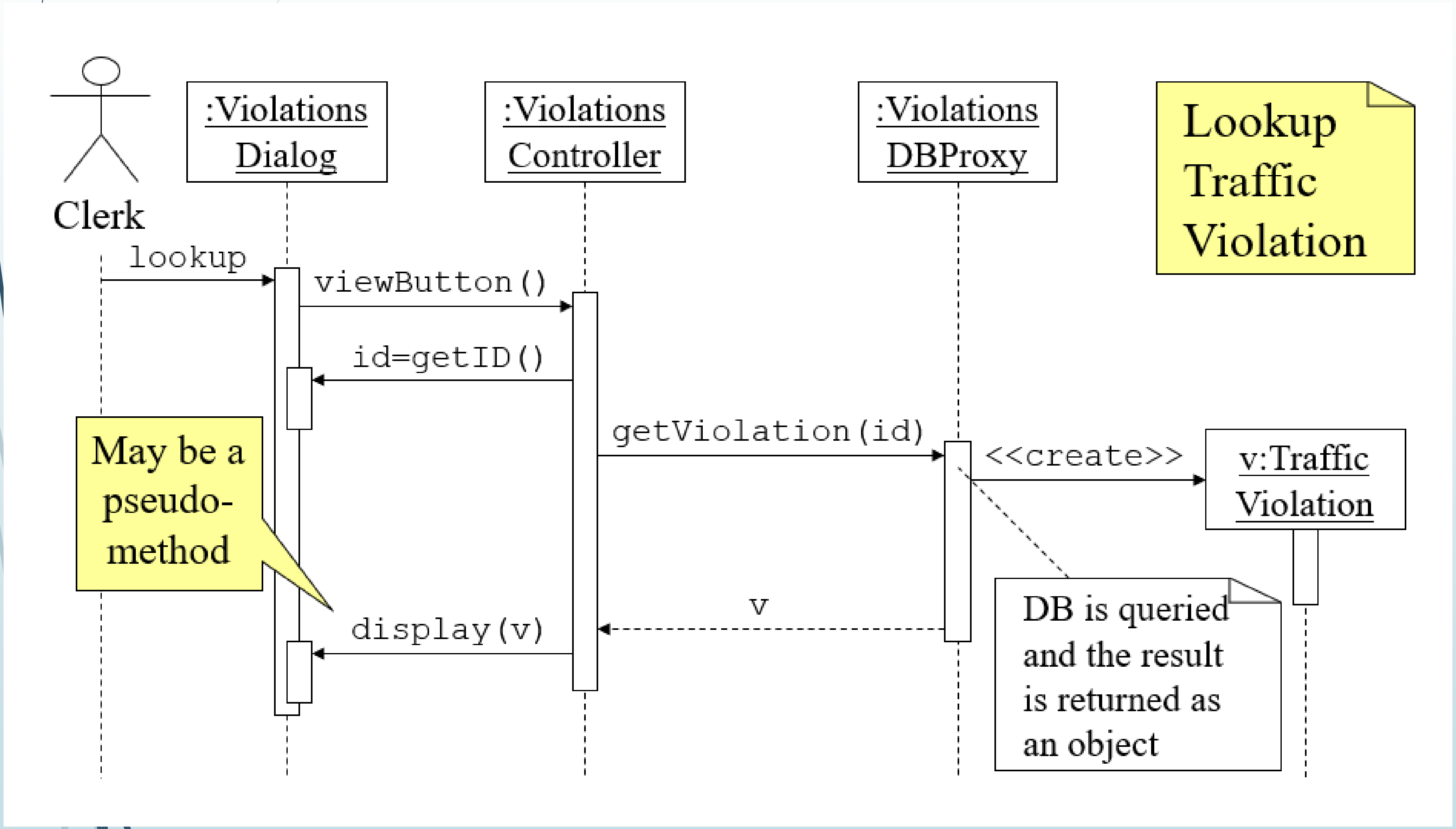
Control Information

- Iteration examples:

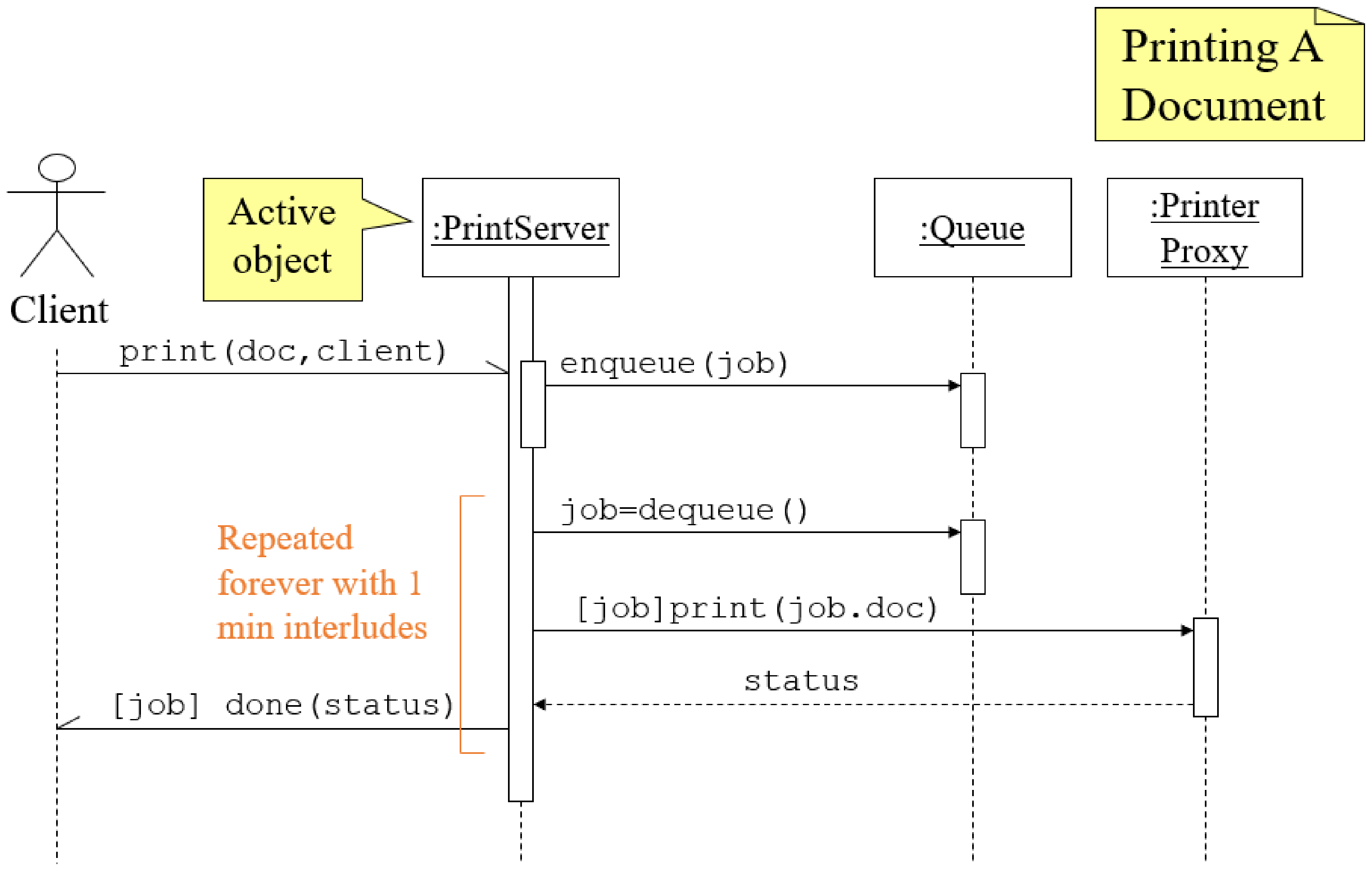


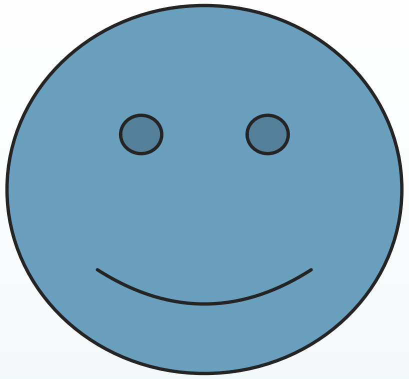


Example 1


Clerk
:Violations Dialog
:Violations Controller
:Violations DBProxy
Lookup Traffic Violation
lookup
viewButton()
id=getID()
getViolation(id)
<<create>>
v:Traffic Violation
May be a pseudo-method
v
display(v)
DB is queried and the result is returned as an object

Example 2


Printing A Document
Client
Active object
:PrintServer
:Queue
:Printer Proxy
print(doc,client)
enqueue(job)
job=dequeue()
[job]print(job.doc)
status
[job] done(status)
Repeated forever with 1 min interludes



Thank you for your attention