UML (Unified Modeling Language)

8 State Machine Diagrams
Activity Diagrams

Software Institute of
Nanjing University

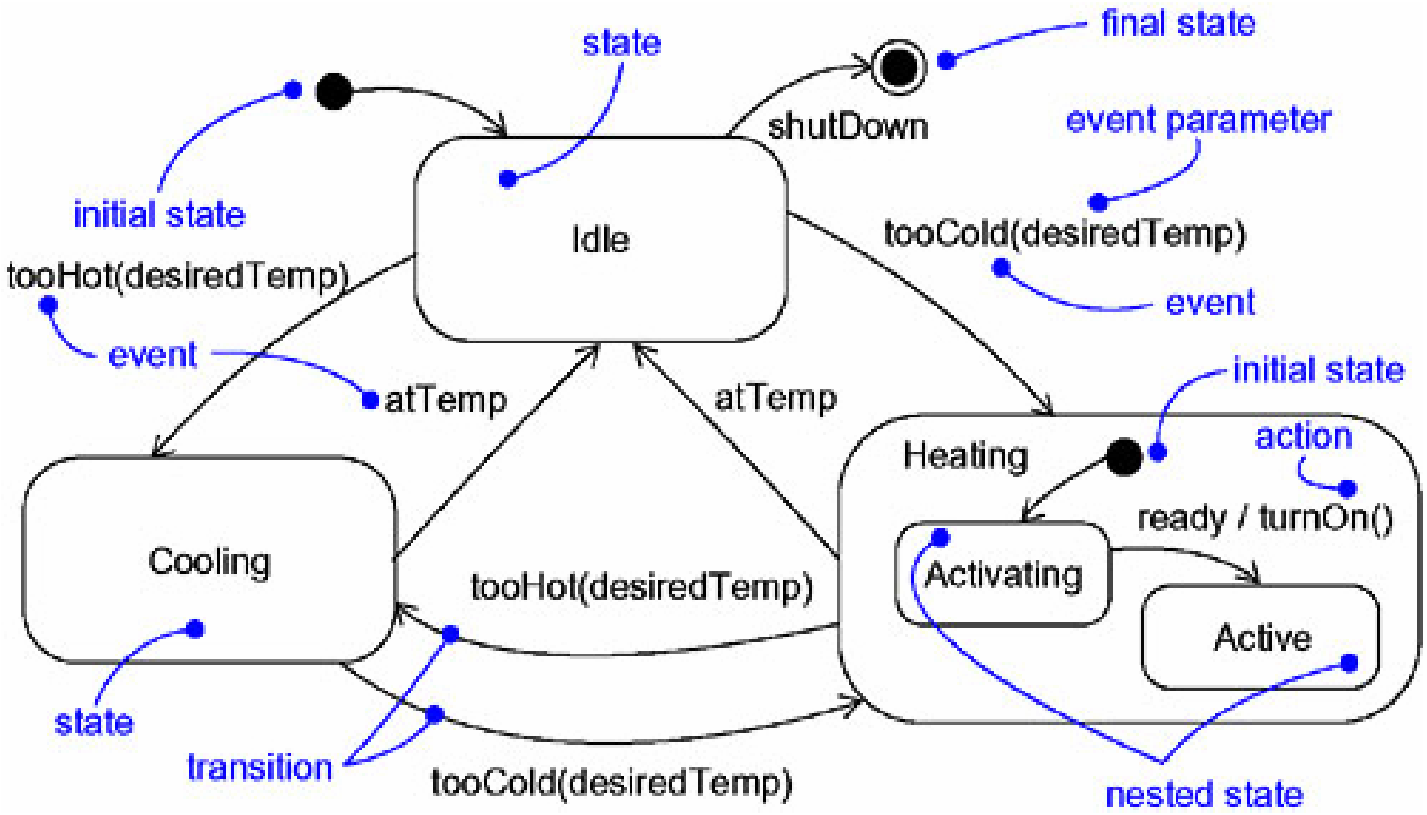
1. State Machine Diagrams

- A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- A *state* is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

1. State Machine Diagrams (cont')

- A *transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line.

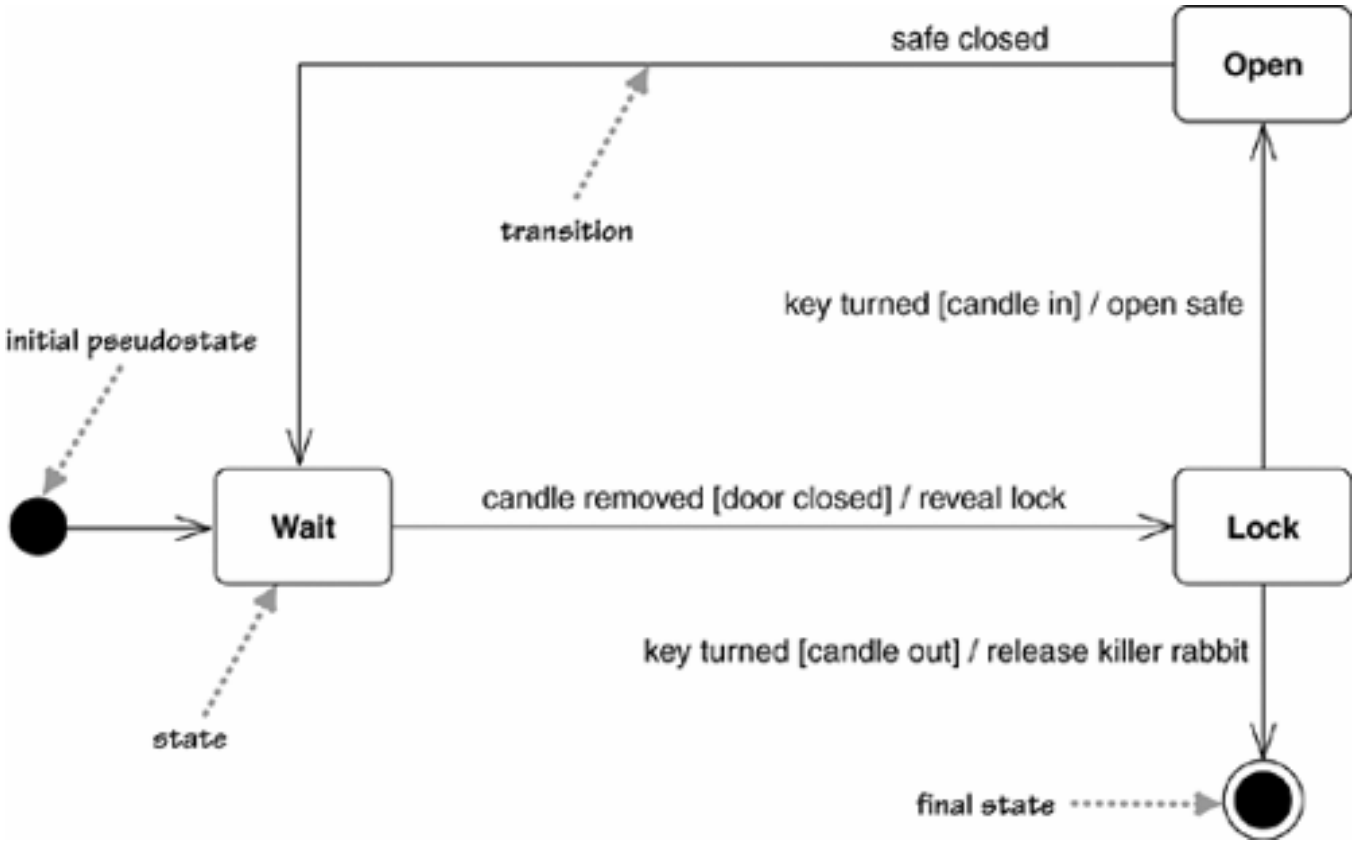
1. State Machine Diagrams (cont')



1.1 A simple state machine diagram

- A controller for a secret panel
  - In the castle, I want to keep my valuables in a safe that's hard to find.
  - So to reveal the lock to the safe, I have to remove a strategic candle from its holder, but this will reveal the lock only while the door is closed.
  - Once I can see the lock, I can insert my key to open the safe.
  - For extra safety, I make sure that I can open the safe only if I replace the candle first. If a thief neglects this precaution, I'll unleash a nasty monster to devour him.

1.1 A simple state machine diagram



1.1 A simple state machine diagram

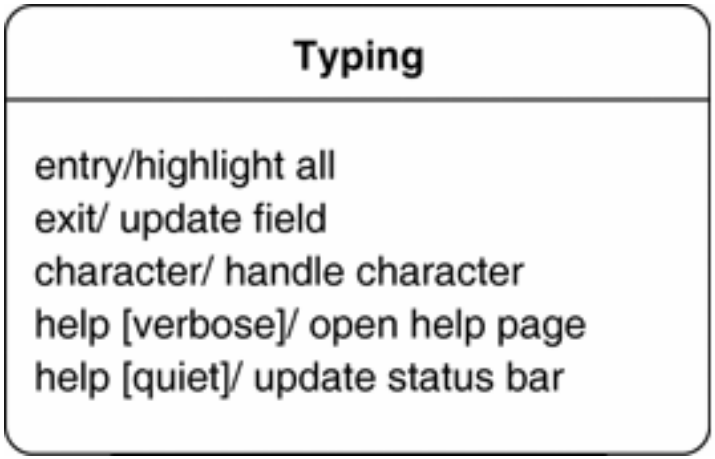
- Each transition has a label that comes in three parts:
  - trigger-signature [guard]/activity.
  - The trigger-signature is usually a single event that triggers a potential change of state.
  - The guard, if present, is a Boolean condition that must be true for the transition to be taken.
  - The activity is some behavior that's executed during the transition. It may be any behavioral expression.

1.1 A simple state machine diagram

- All three parts to a transition are optional.
  - A missing *activity* indicates that you don't do anything during the transition.
  - A missing *guard* indicates that you always take the transition if the event occurs.
  - A missing *trigger-signature* is rare but does occur. It indicates that you take the transition immediately, which you see mostly with *activity states*

1.2 Internal Activities

- States can react to events without transition, using *internal activities*: putting the event, guard, and activity inside the state box itself.

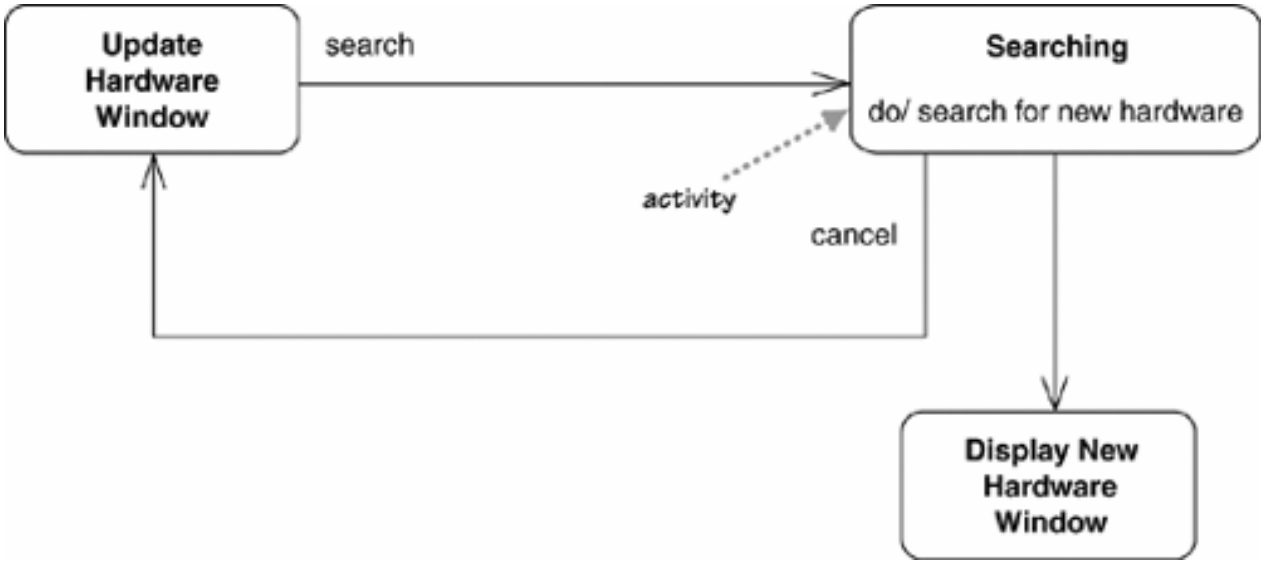


1.2 Internal Activities (cont')

- An internal activity is similar to a self-transition: a transition that loops back to the same state.
- Two special activities: the entry and exit activities.
  - The entry activity is executed whenever you enter a state;
  - The exit activity, whenever you leave.
- However, internal activities do not trigger the entry and exit activities; that is the difference between internal activities and self-transitions.

1.3 Activity States

- You can have states in which the object is doing some ongoing work.
- The ongoing activity is marked with the *do/;* hence the term do-activity.

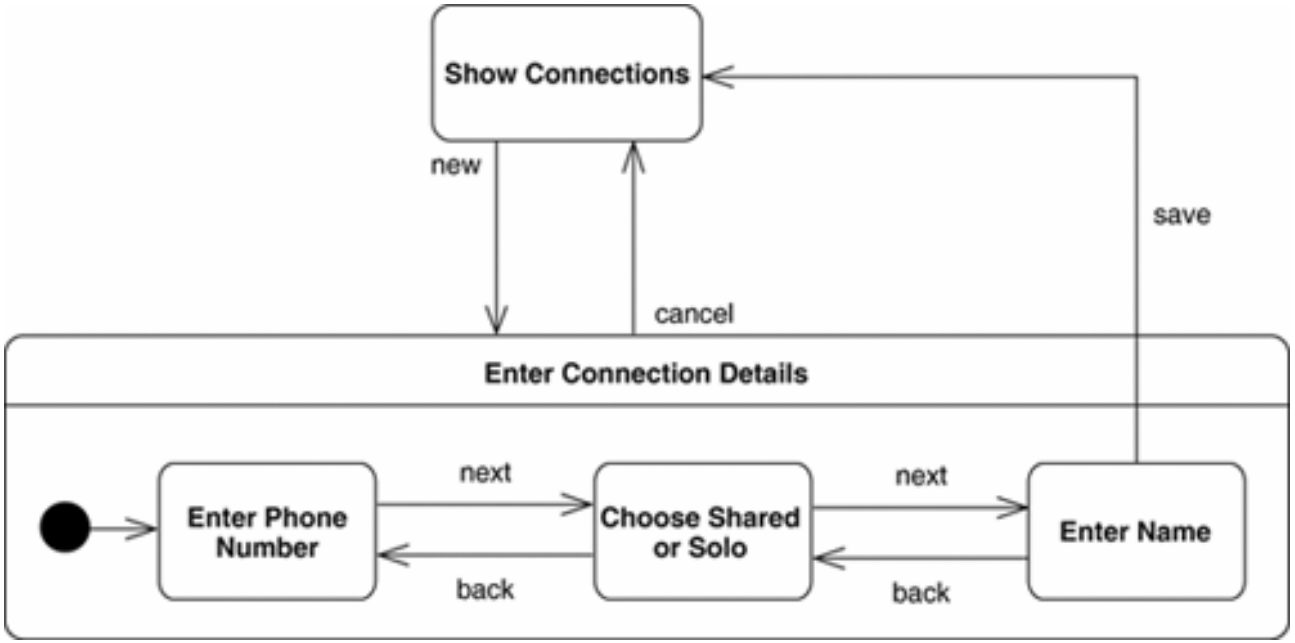


1.3 Activity States (cont’)

- Both *do-activities* and *regular activities* represent carrying out some behavior. The critical difference between the two is that *regular activities* occur "instantaneously" and cannot be interrupted by regular events, while *do-activities* can take finite time and can be interrupted.
- *UML 1 used the term action for regular activities and used activity only for do-activities.*

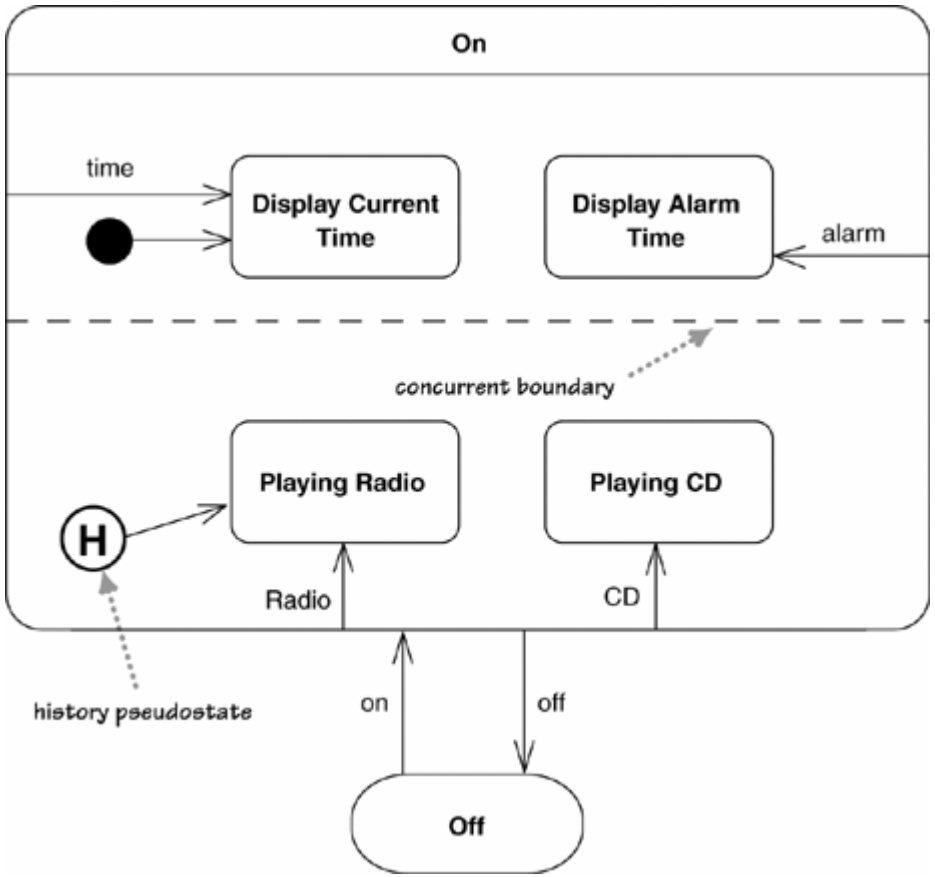
1.4 Superstates

- Often, you'll find that several states share common transitions and internal activities. In these cases, you can make them substates and move the shared behavior into a superstate



1.5 Concurrent States

- States can be broken into several orthogonal state diagrams that run concurrently.



1.6 Implementing State Diagrams

- A state diagram can be implemented in three main ways:
  - nested switch,
  - the State pattern,
  - and state tables.

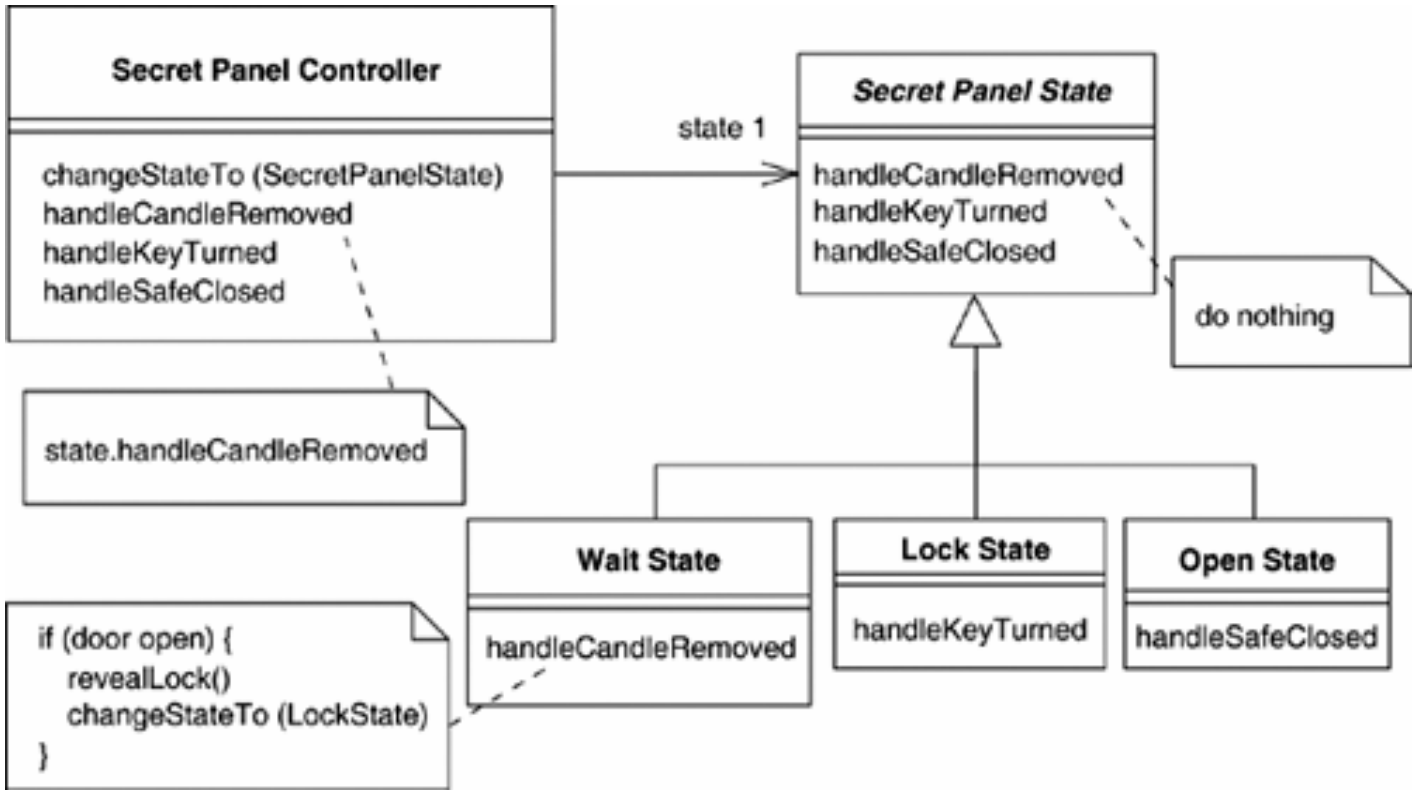
1.6.1 nested switch

- The most direct approach to handling a state diagram is a nested switch statement, such as *nested switch sample*.
- Although it's direct, it's long-winded, even for this simple case. It's also very easy for this approach to get out of control .

1.6.2 state pattern

- The State pattern [Gang of Four] creates a hierarchy of state classes to handle behavior of the states. Each state in the diagram has one state subclass. The controller has methods for each event, which simply forwards to the state class.
- The top of the hierarchy is an abstract class that implements all the event-handling methods to do nothing. For each concrete state, you simply override the specific event methods for which that state has transitions.

1.6.2 state pattern (cont')



1.6.3 state table

- The state table approach captures the state diagram information as data.

Source State	Target State	Event	Guard	Procedure
Wait	Lock	Candle removed	Door closed	Reveal lock
Lock	Open	Key turned	Candle in	Open safe
Lock	Final	Key turned	Candle out	Release killer rabbit
Open	Wait	Safe closed		

1.7 When to Use State Diagrams

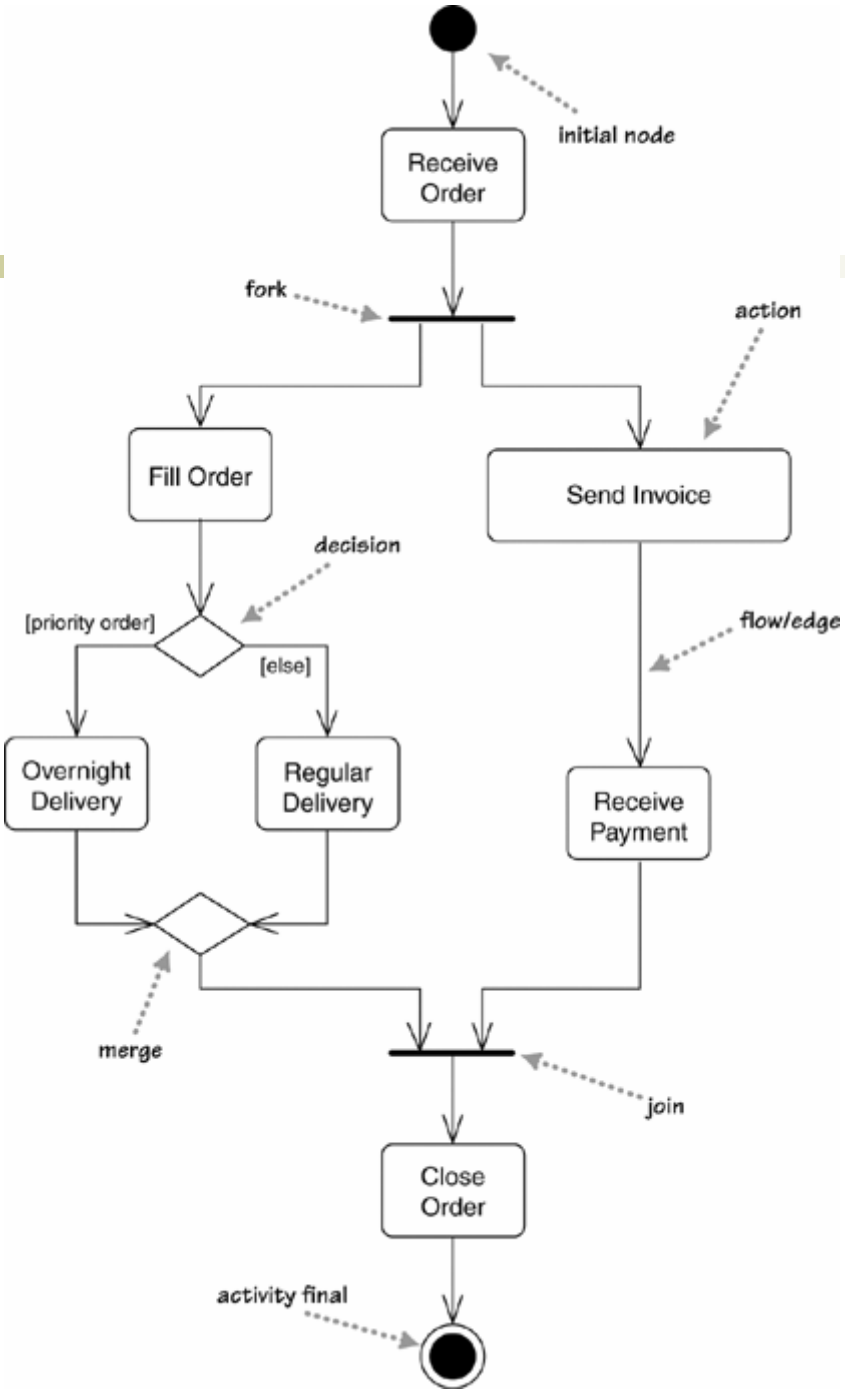
- State diagrams are good at describing the behavior of an object across several use cases.
- State diagrams are not very good at describing behavior that involves a number of objects collaborating.
- Real-time designers tend to use state models a lot

2. Activity Diagrams

- Activity diagrams are a technique to describe procedural logic, business process, and work flow.
- In many ways, they play a role similar to flowcharts, but the principal difference between them and flowchart notation is that they support parallel behavior.

2.1 A simple activity diagram

- Next figure shows a simple example of an activity diagram. We begin at the *initial node action* and then do the action *Receive Order*. Once that is done, we encounter a fork. A *fork* has one incoming flow and several outgoing concurrent flows.
- It says that *Fill Order*, *Send Invoice*, and the subsequent actions occur in parallel. Essentially, this means that the sequence between them is irrelevant.

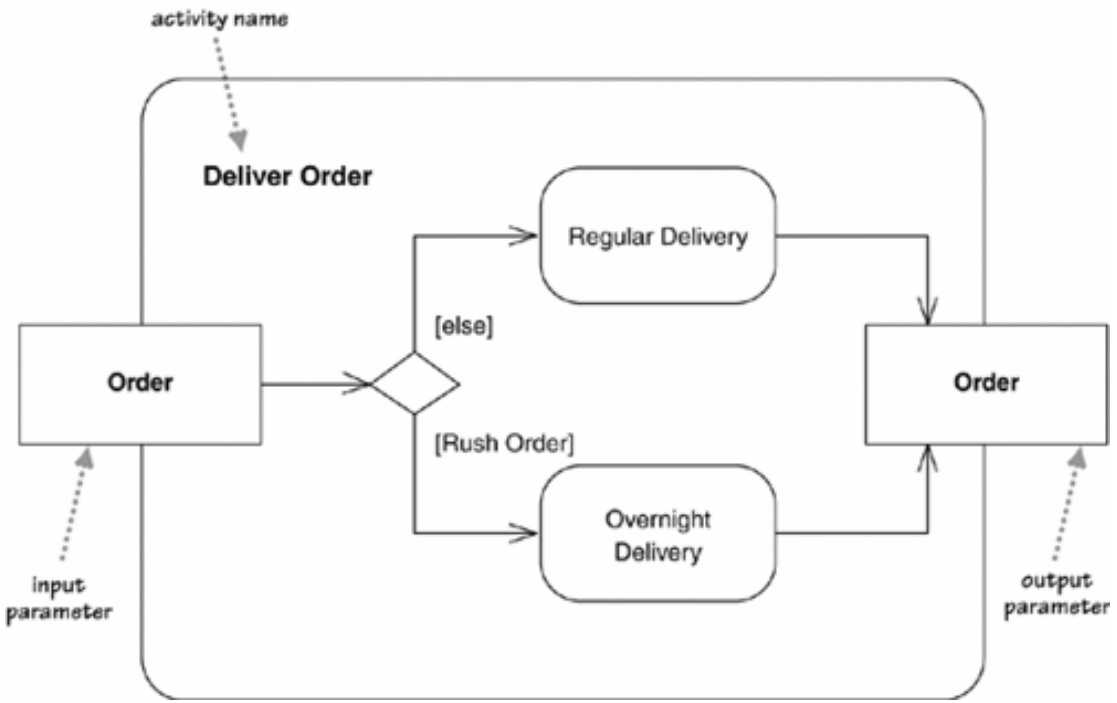

initial node
Receive Order
fork
action
Fill Order
Send Invoice
decision
[priority order]
[else]
flow/edge
Overnight Delivery
Regular Delivery
Receive Payment
merge
join
Close Order
activity final

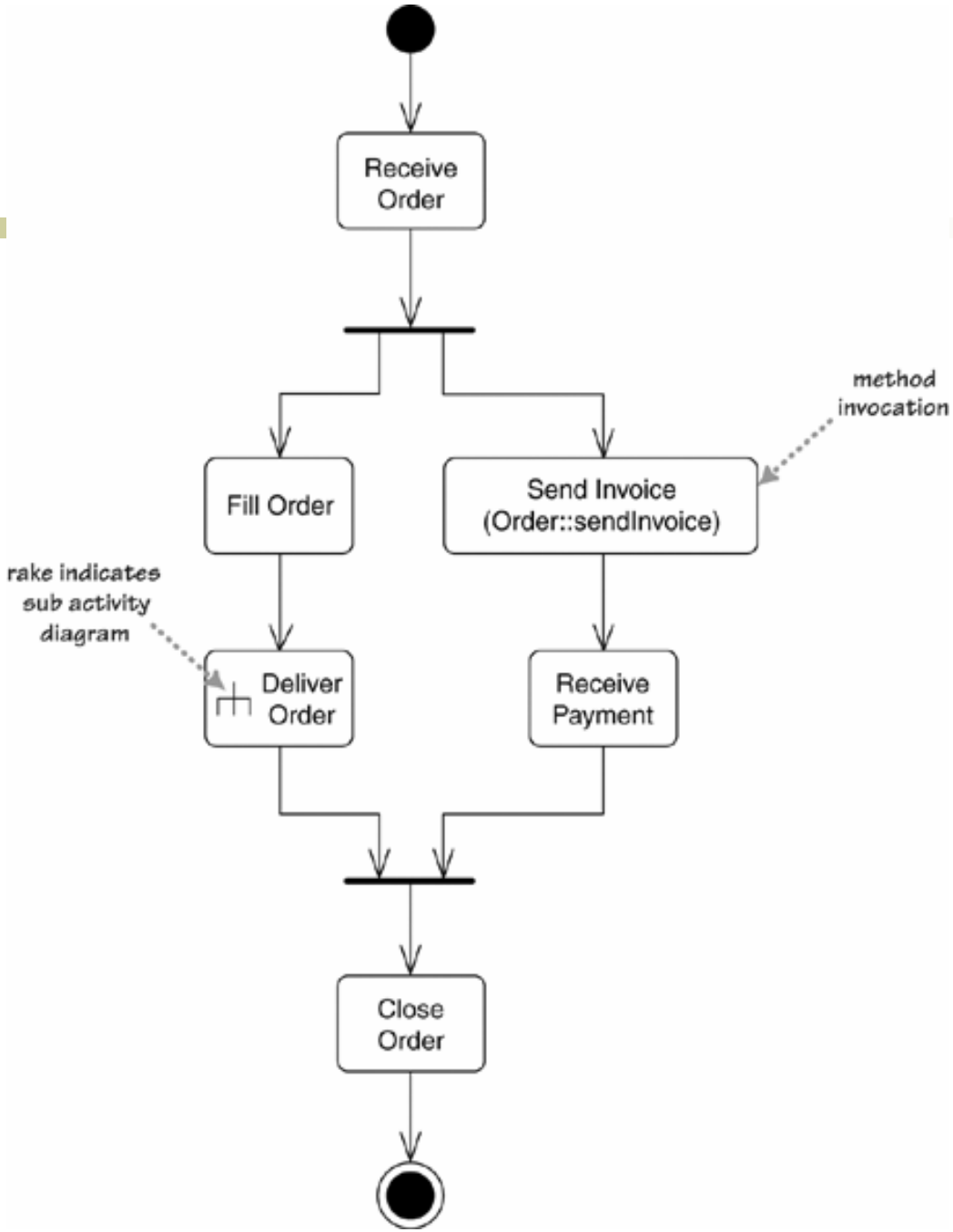
2.1 A simple activity diagram

- With a *join*, the outgoing flow is taken only when all the incoming flows reach the join. So you can close the order only when you have both received the payment and delivered.
- Conditional behavior is delineated by decisions and merges.
  - A decision, called branch in UML 1, has a single incoming flow and several guarded out-bound flows.
  - A merge has multiple input flows and a single output. A merge marks the end of conditional behavior started by a decision.

2.2 Decomposing an Action

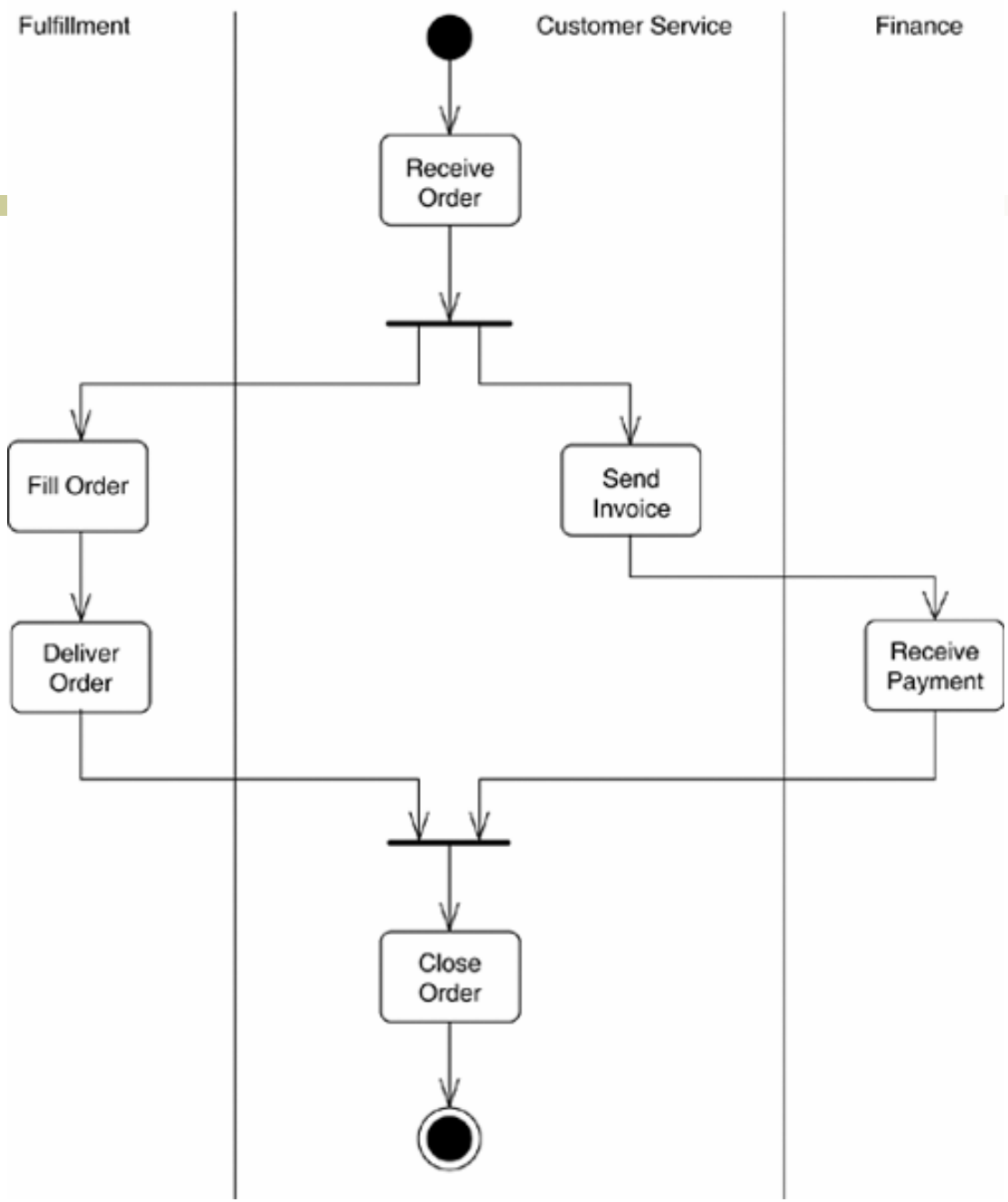
- Actions can be decomposed into subactivities.




Receive Order
Fill Order
Send Invoice (Order::sendInvoice)
method invocation
rake indicates sub activity diagram
Deliver Order
Receive Payment
Close Order

2.3 Partitions

- If you want to show who does what, you can divide an activity diagram into partitions, which show which actions one class or organization unit carries out.


Fulfillment
Customer Service
Finance
Receive Order
Fill Order
Send Invoice
Deliver Order
Receive Payment
Close Order

2.3 Partitions (cont')

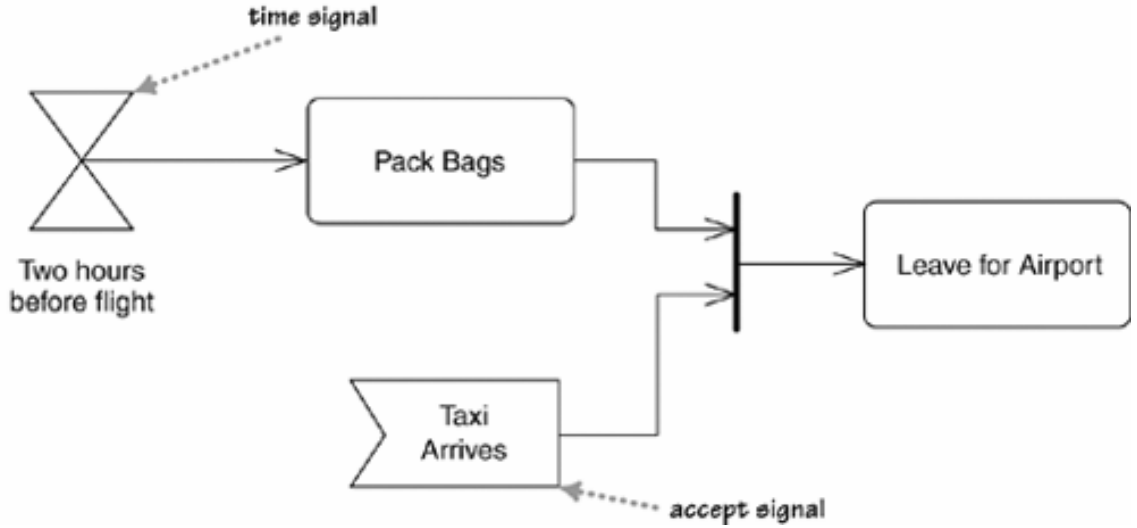
- This style is often referred to as *swim lanes*, for obvious reasons and was the only form used in UML 1.x.
- In UML 2, you can use a two-dimensional grid, so the swimming metaphor no longer holds water. You can also take each dimension and divide the rows or columns hierarchically.

2.4 Signals

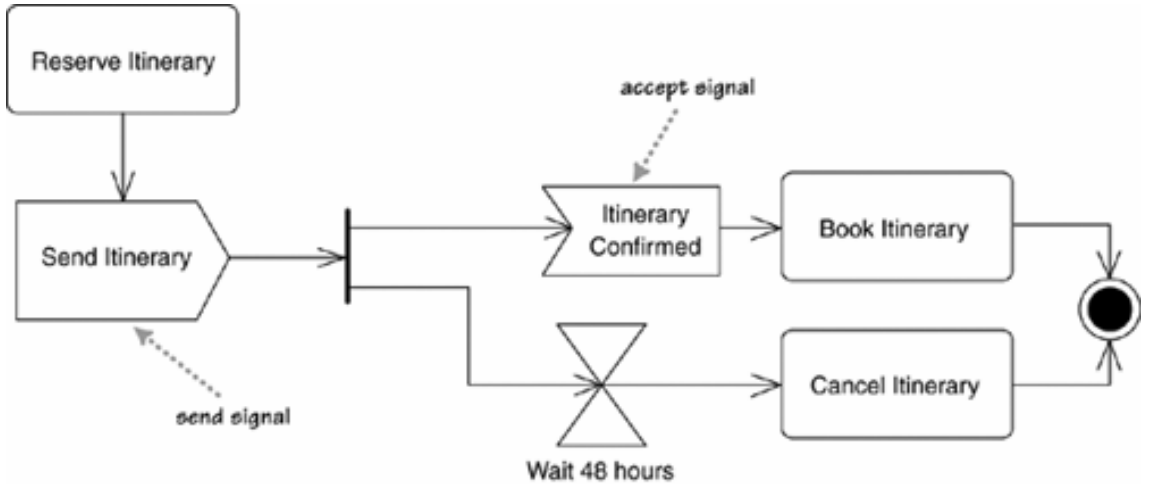
- A *signal* indicates that the activity receives an event from an outside process. This indicates that the activity constantly listens for those signals, and the diagram defines how the activity reacts.

2.4 Signals (cont')

- Accept signals



- Send signal



2.5 Tokens

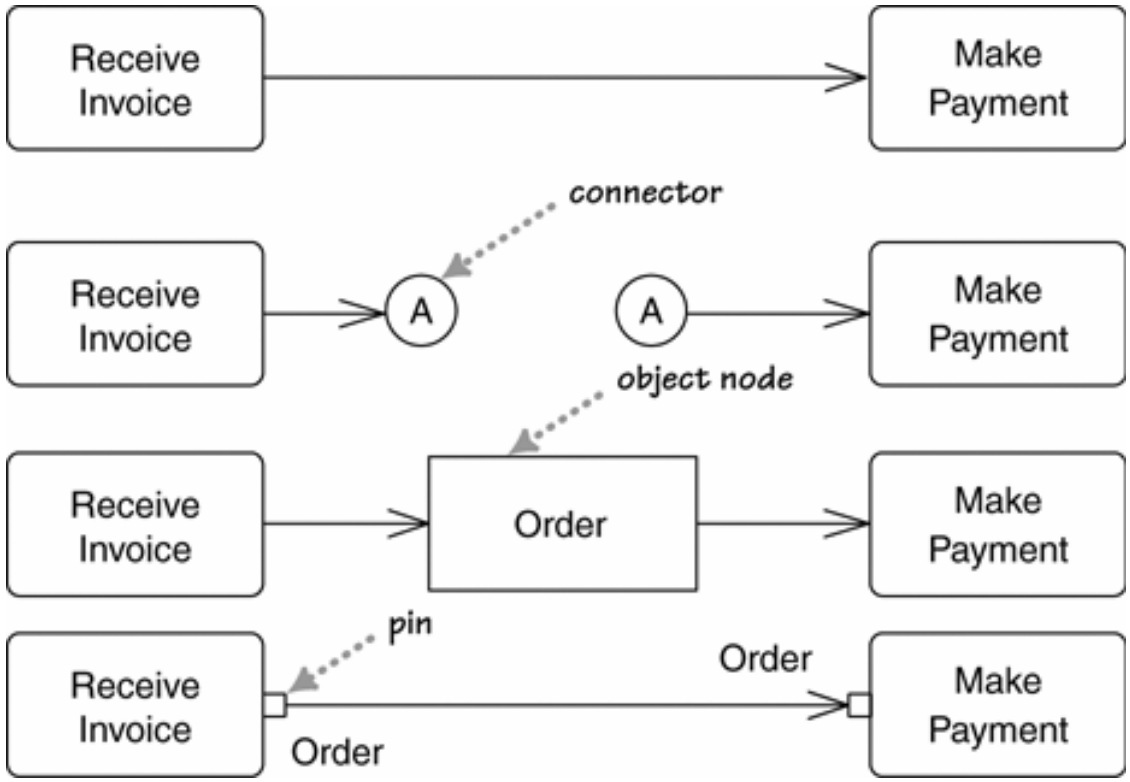
- The activity section of the specification talks a lot about tokens and their production and consumption.
  - The initial node creates a token, which then passes to the next action, which executes and then passes the token to the next. At a fork, one token comes in, and the fork produces a token on each of its outward flows. Conversely, on a join, as each inbound token arrives, nothing happens until all the tokens appear at the join; then a token is produced on the outward flow.

2.6 Flows and Edges

- UML 2 uses the terms flow and edge synonymously to describe the connections between two actions.
- The simplest kind of edge is the simple arrow between two actions. You can give an edge a name if you like, but most of the time, a simple arrow will suffice.

2.6 Flows and Edges

- Four ways of showing an edge



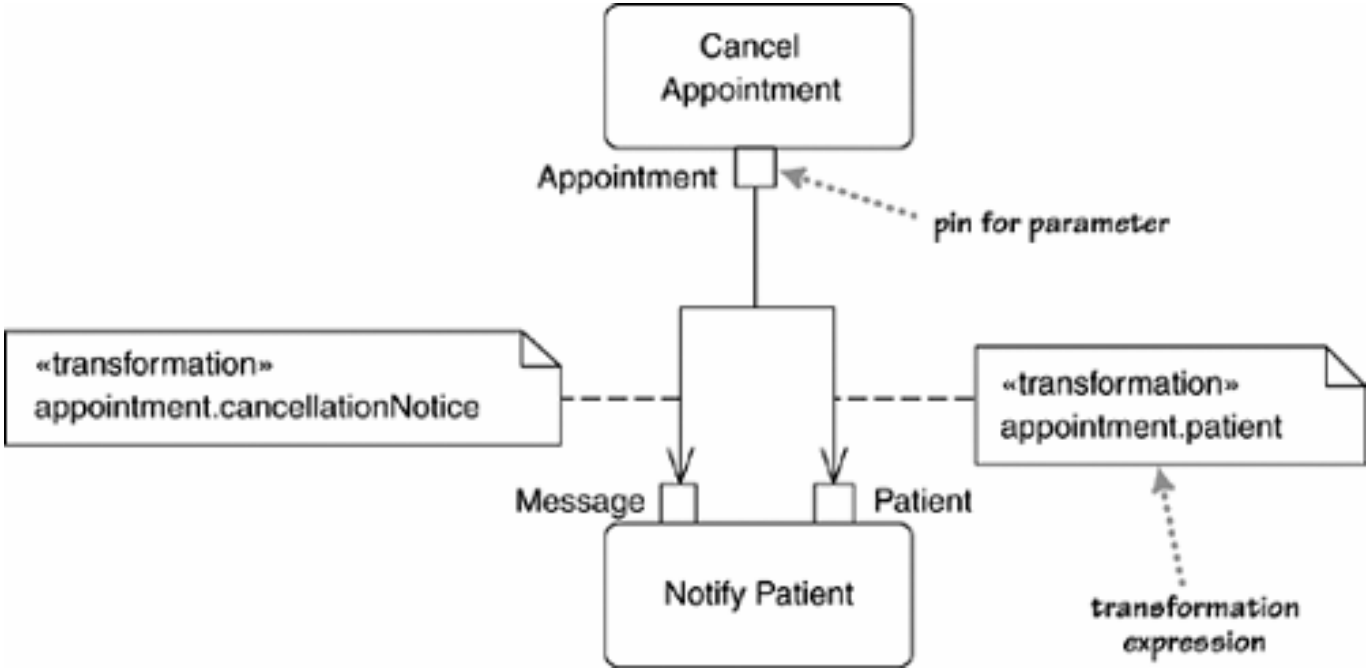
2.7 Pins and Transformations

- Actions can have parameters, just as methods do.
- You don't need to show information about parameters on the activity diagram, but if you wish, you can show them with *pins.*
- If you're decomposing an action, pins correspond to the parameter boxes on the decomposed diagram.

2.7 Pins and Transformations (cont')

- When you're drawing an activity diagram strictly, you have to ensure that the output parameters of an outbound action match the input parameters of another.
- If they don't match, you can indicate a transformation to get from one to another.
- The transformation must be an expression that's free of side effects: essentially, a query on the output pin quary that supplies an object of the right type for the input pin.

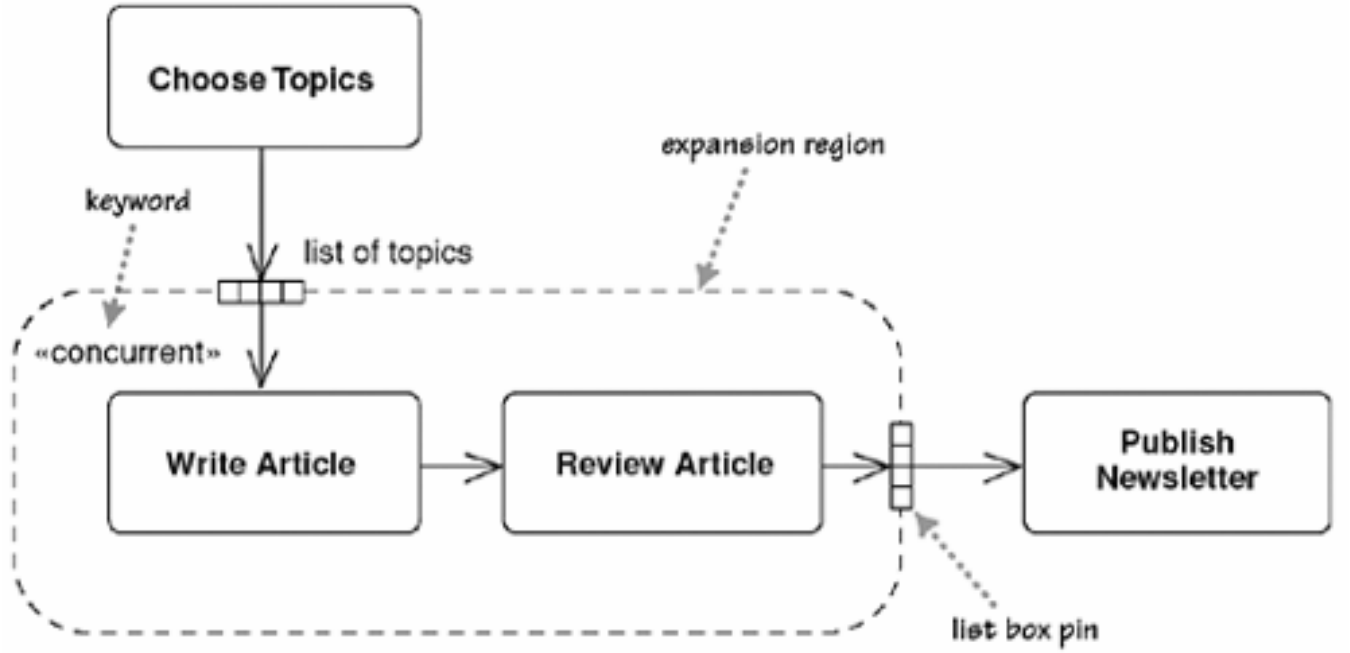
2.7 Pins and Transformations (cont')



2.8 Expansion Regions

- With activity diagrams, you often run into situations in which one action's output triggers multiple invocations of another action. There are several ways to show this, but the best way is to use an expansion region.
- An expansion region marks an activity diagram area where actions occur once for each item in a collection.

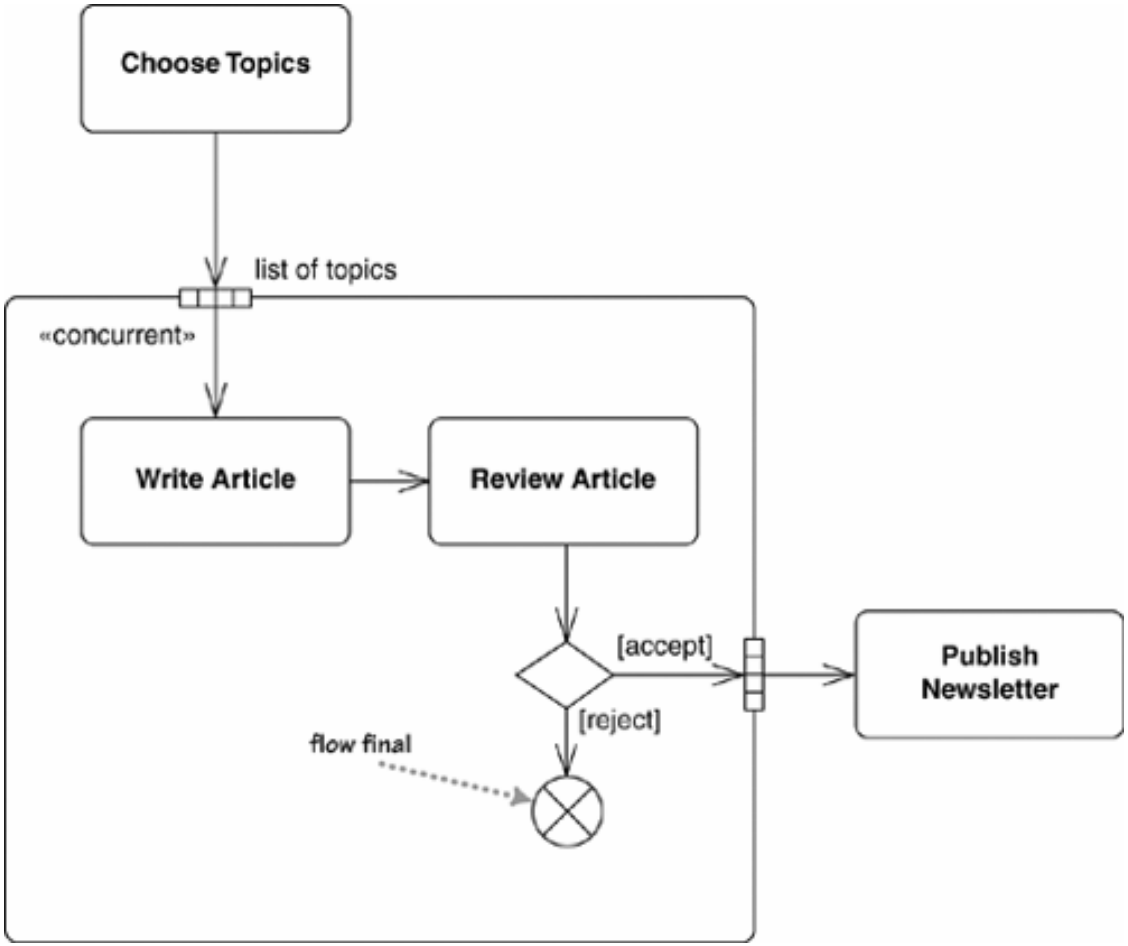
2.8 Expansion Regions (cont')



2.9 Flow Final

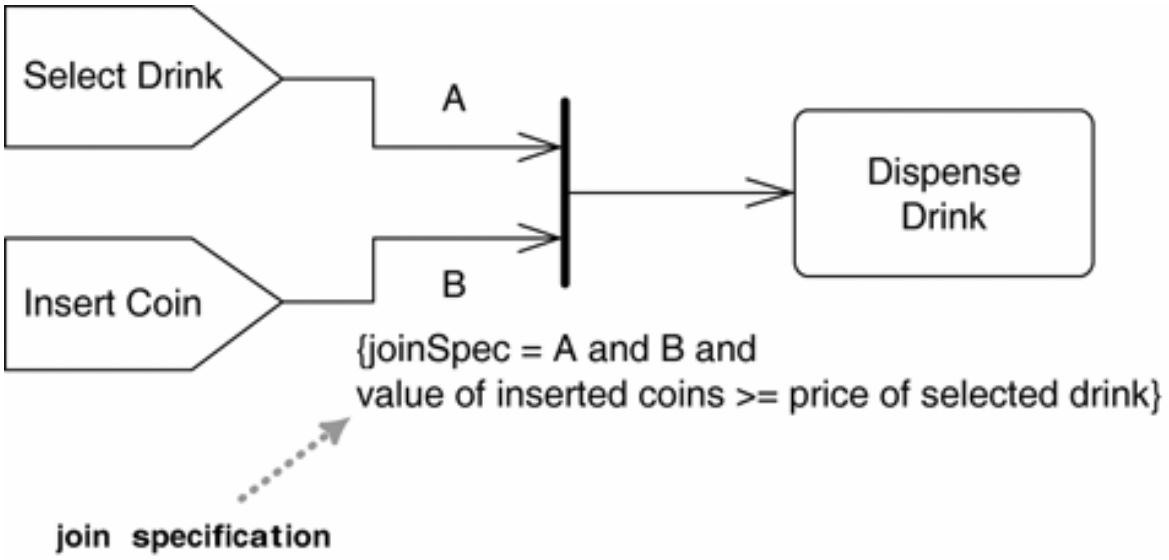
- Once you get multiple tokens, as in an expansion region, you often get flows that stop even when the activity as a whole doesn't end. A flow final indicates the end of one particular flow, without terminating the whole activity.

2.9 Flow Final (cont')



2.10 Join Specifications

- A join specification is a Boolean expression attached to a join. Each time a token arrives at the join, the join specification is evaluated and if true, an output token is emitted.

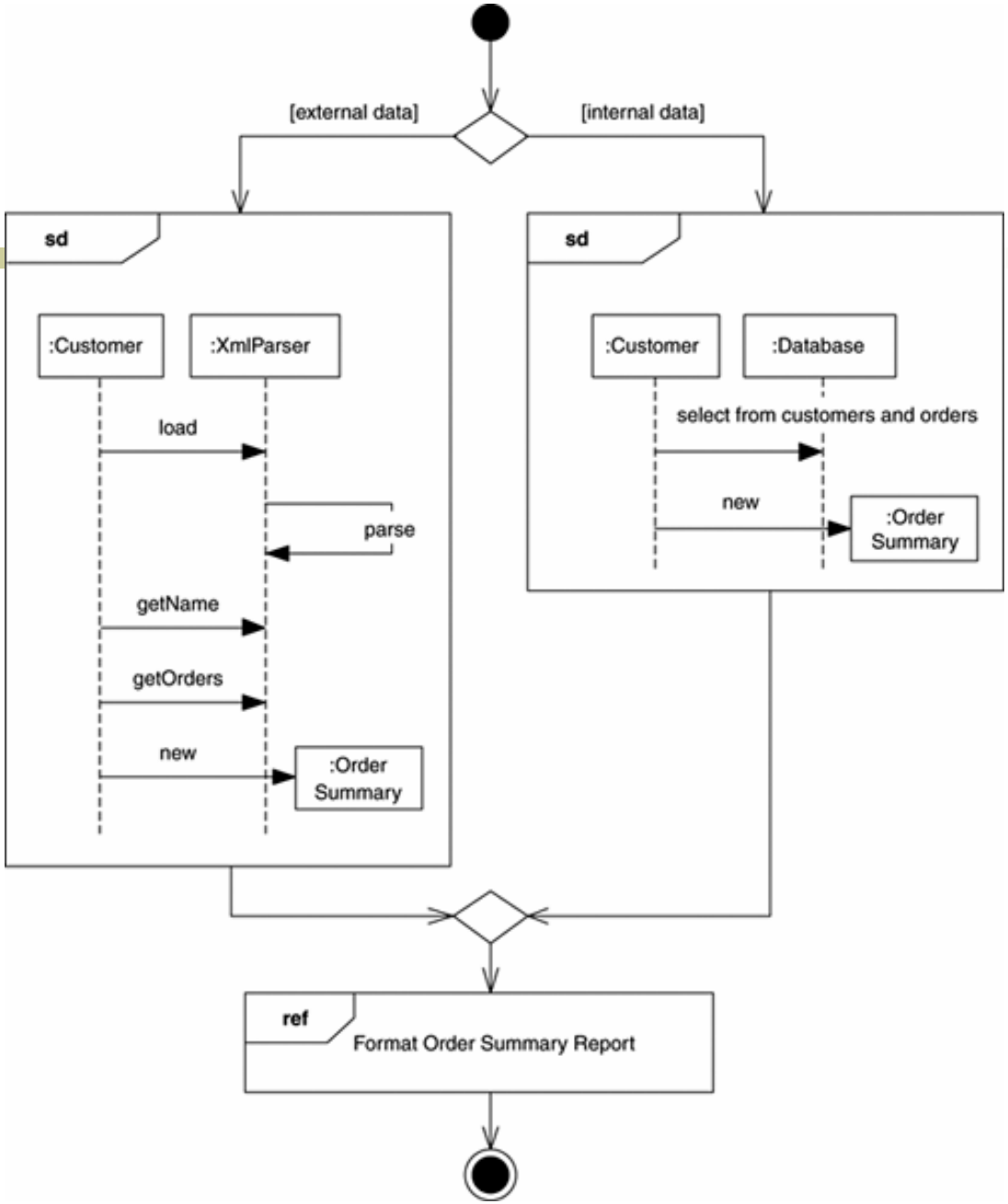


2.11 When to Use Activity Diagrams

- The great strength of activity diagrams lies in the fact that they support and encourage parallel behavior. This makes them a great tool for work flow and process modeling

3. Interaction Overview Diagrams

- Interaction overview diagrams are a grafting together of activity diagrams and sequence diagrams.
- You can think of interaction overview diagrams either as activity diagrams in which the activities are replaced by little sequence diagrams, or as a sequence diagram broken up with activity diagram notation used to show control flow.


[external data]
[internal data]
sd
:Customer
:XmlParser
load
parse
getName
getOrders
new
:Order Summary
sd
:Customer
:Database
select from customers and orders
new
:Order Summary
ref
Format Order Summary Report

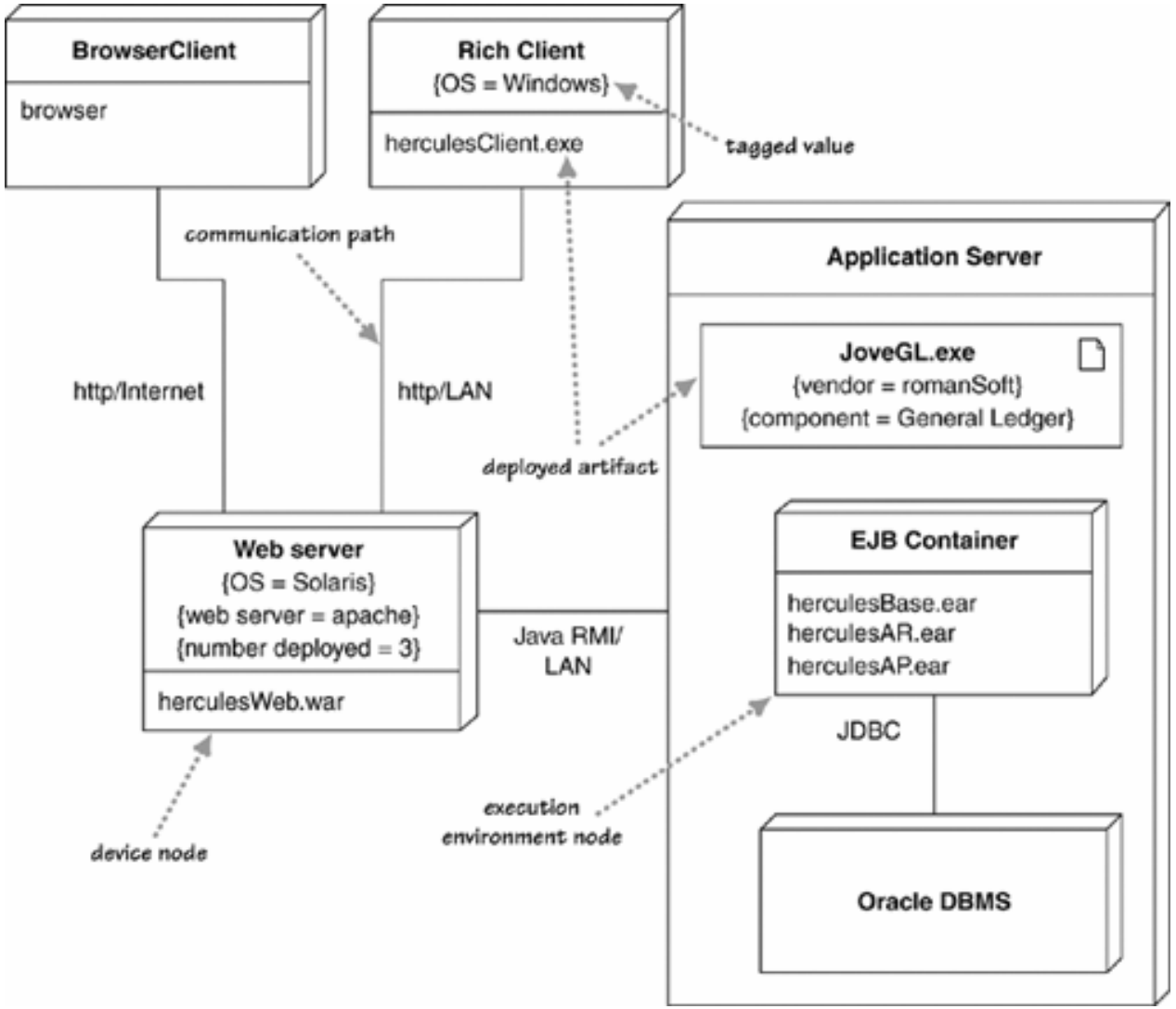
4. Deployment Diagrams

- Deployment diagrams show a system's physical layout, revealing which pieces of software run on what pieces of hardware.
- A node is something that can host some software. Nodes come in two forms.
  - A device is hardware, it may be a computer or a simpler piece of hardware connected to a system.
  - An execution environment is software that itself hosts or contains other software, examples are an operating system or a container process.

4. Deployment Diagrams (cont')

- The nodes contain *artifacts*, which are the physical manifestations of software: usually, files. These files might be executables (such as .exe files, binaries, DLLs, JAR files, assemblies, or scripts), or data files, configuration files, HTML documents, and so on. Listing an artifact within a node shows that the artifact is deployed to that node in the running system.

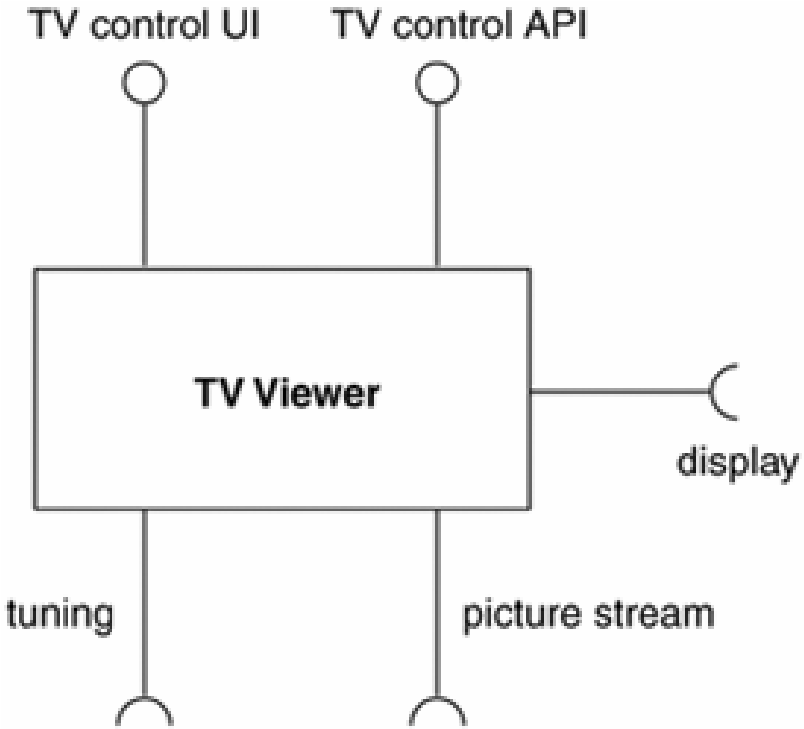
4. Deployment Diagrams (cont')

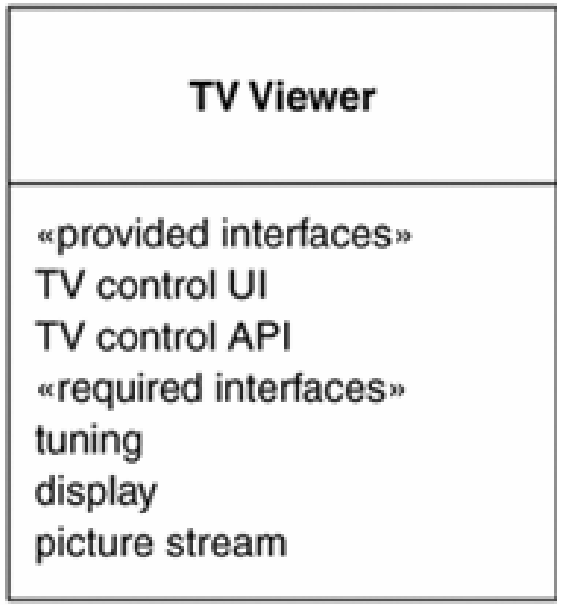


5. Composite Structures Diagram

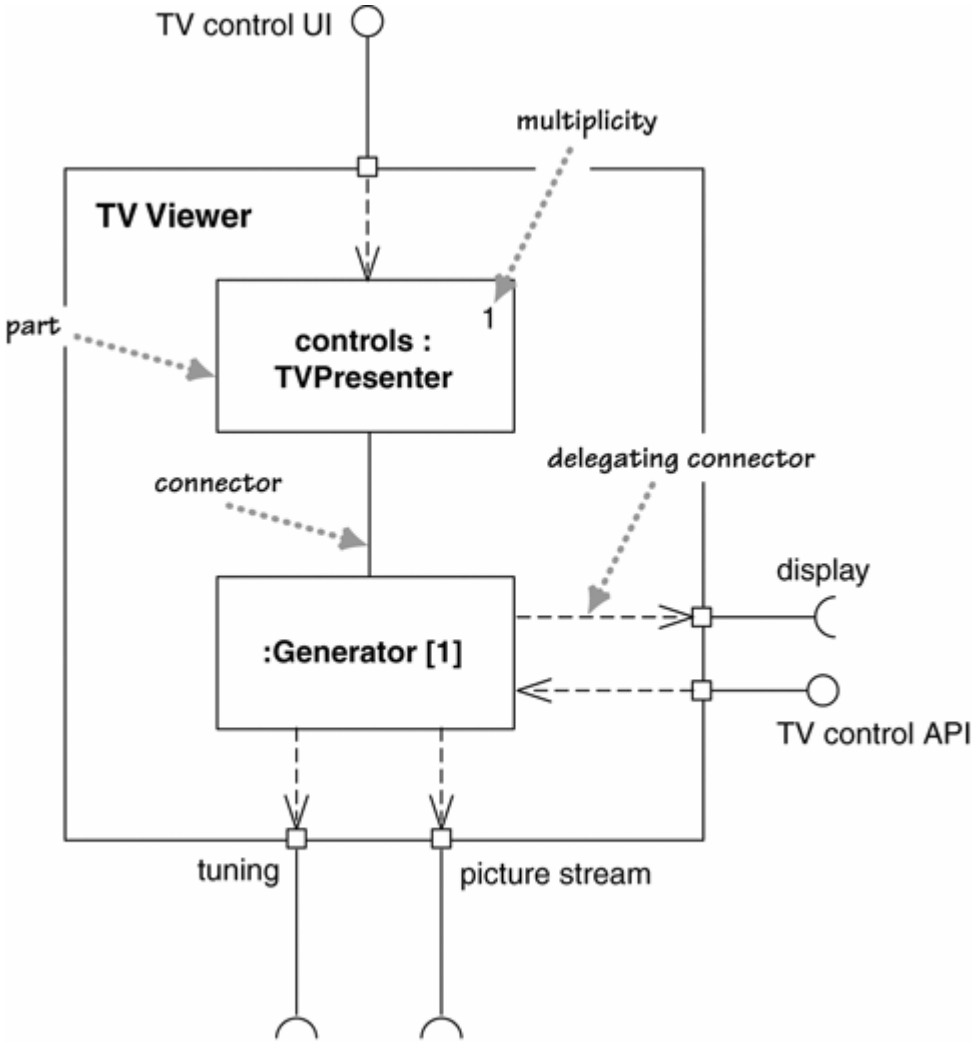
- One of the most significant new features in UML 2 is the ability to hierarchically decompose a class into an internal structure. This allows you to take a complex object and break it down into parts.

5.1 Two ways of showing a Class and its interfaces

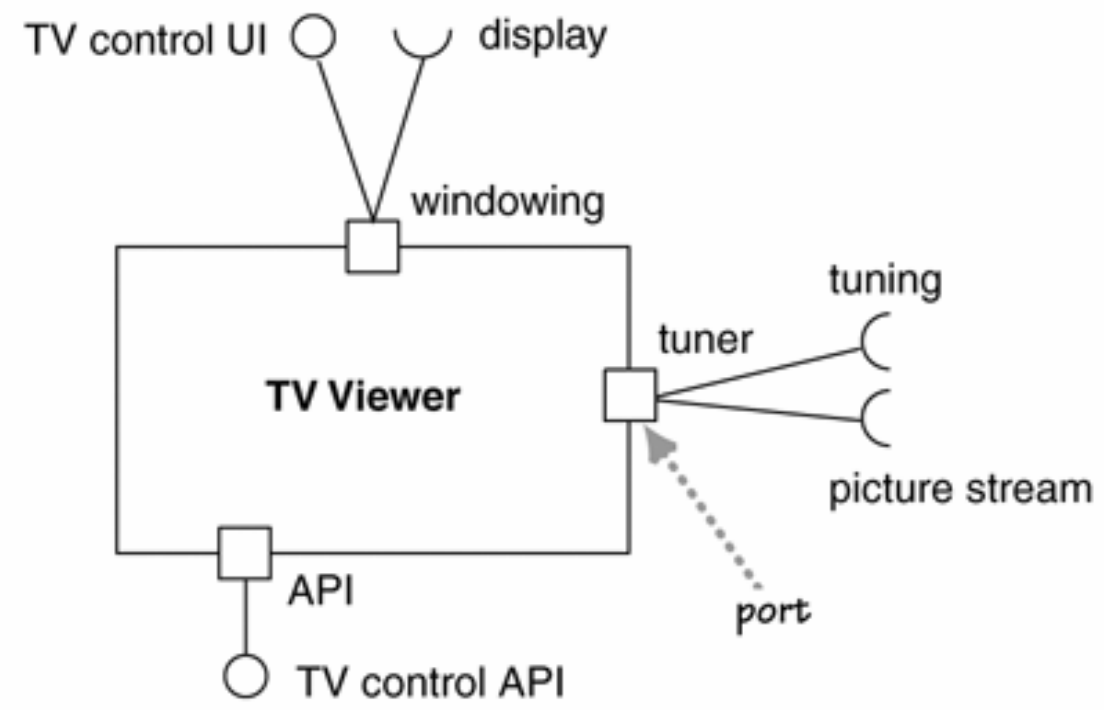




5.2 Internal view of a component

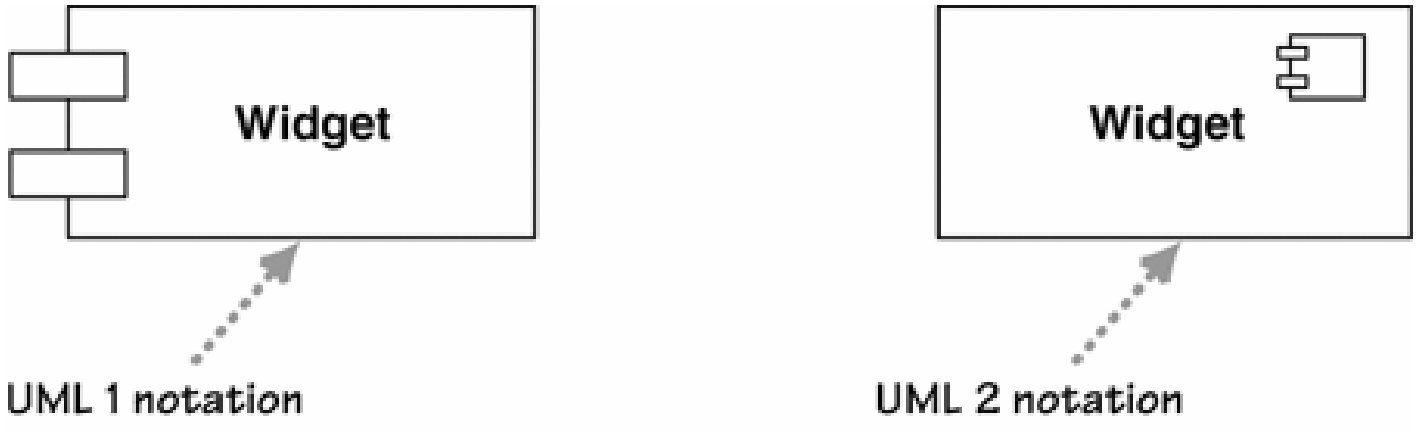


5.3 A component with multiple ports

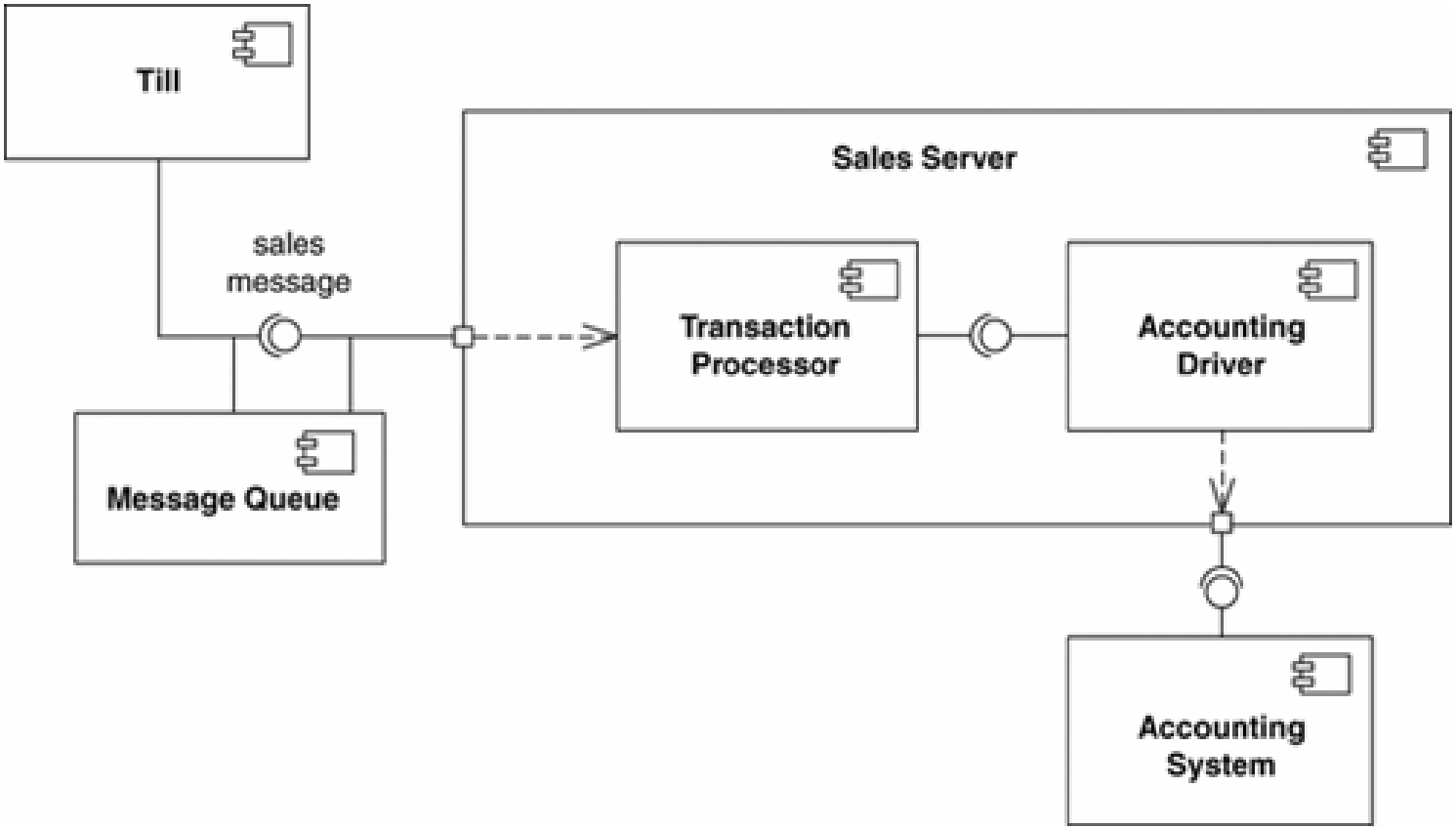


6. Component Diagrams

- UML 1 had a distinctive symbol for a component. UML 2 removed that icon but allows you to annotate a class box with a similar-looking icon. Alternatively, you can use the «component» keyword.



6.1 An example component diagram



考试事宜

- 期末考试占总成绩的60%
- 共8题
- 因为有画图题目，需要带尺子考试