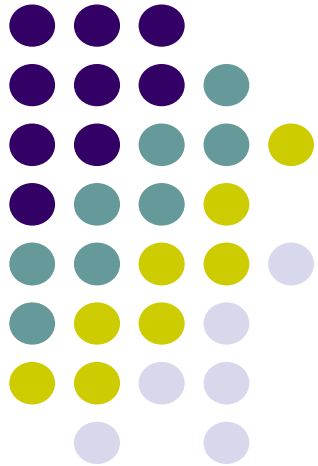
State and Activity Diagrams

System Behavior

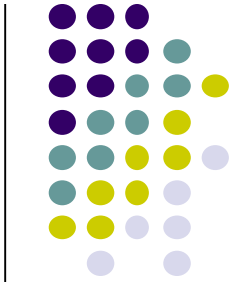
State Diagrams

Activity diagram

Examples



System Behavior



System behavior – described by:

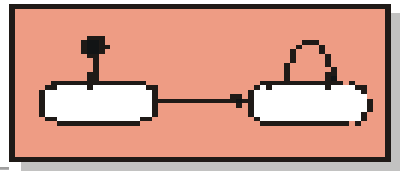
- ***Use case diagrams*** - use cases and scenarios
- ***Interactions diagrams*** – sequence and communication (collaboration) diagrams
- ***State transition diagrams*** – show the behavior inside an object

A State Specification enables you to display and modify the properties and relationships of a state on:

- a ***state*** (before UML 2.0 - Statechart) ***diagram*** or on
- an ***activity diagram***.

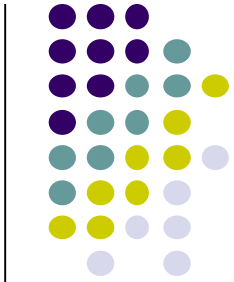
State Diagrams
(until UML 2.0 – Statechart Diagrams)

State Diagram	A **state (statechart) diagram** shows a state machine, a dynamic behavior that specifies the sequences of states that an object goes through during its life in response to events, together with its responses and actions.



- A state machine diagram models the behaviour of a single object, specifying *the sequence of events that an object goes through during its lifetime in response to events*.
- It shows the sequences of ***states*** that an object goes through, the ***events*** that cause a ***transition*** from one state to another, and the ***actions*** that result from a state change.

State Diagrams (cont.)



State (state-chart) diagrams versus activity diagrams:

- Orientation - statechart diagrams are ***state centric***, while activity diagrams are ***activity centric***
- Purpose - a statechart diagram is typically used to model the ***discrete stages*** of an object's lifetime, whereas an activity diagram is better suited to model the ***sequence of activities*** in a process.

State

- ***State*** - represents a condition or situation during the life of an object during which it:
  - **Satisfies some condition**
  - **Performs some action**
  - **Waits for some event**
- May be of type:
  - **simple or composite state**
  - **real or pseudo-state**
- Each state represents the cumulative history of its behavior. The state icon appears as a rectangle with rounded corners and a name (i.e., Wait). It also contains a compartment for actions.

Start, End and Terminate States

A ***start (initial) pseudostate***:

- explicitly shows the beginning of a workflow on an activity diagram or the beginning of the events that cause a transition on a statechart diagram
- source of single transition to the default state of the diagram or the composite state
- can be *only one*.

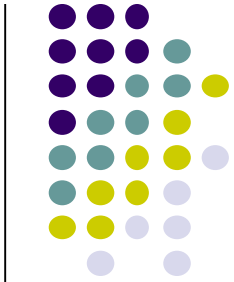
An ***end (final) state***:

- represents a final or terminal state
- final states can be *several*
- represent completion of the process (region)

A ***terminate pseudostate***:

- implies termination of the state machine
- by *destruction* of its context object

Pseudostates



- initial pseudostate
- terminate pseudostate
- entry point
- exit point
- choice
- join
- fork
- junction
- shallow history pseudostate
- deep history pseudostate

State Transition

A *state transition* indicates that an object in the source state will perform certain specified actions and enter the destination state when:

- ***Non-automatic*** - a specified event occurs
- ***Automatic*** - when certain conditions are satisfied.

A state transition is a relationship between *two states*, *two activities*, or between *an activity and a state*.
It takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type

State Transition - Naming

Naming - transitions are labeled with the following syntax:

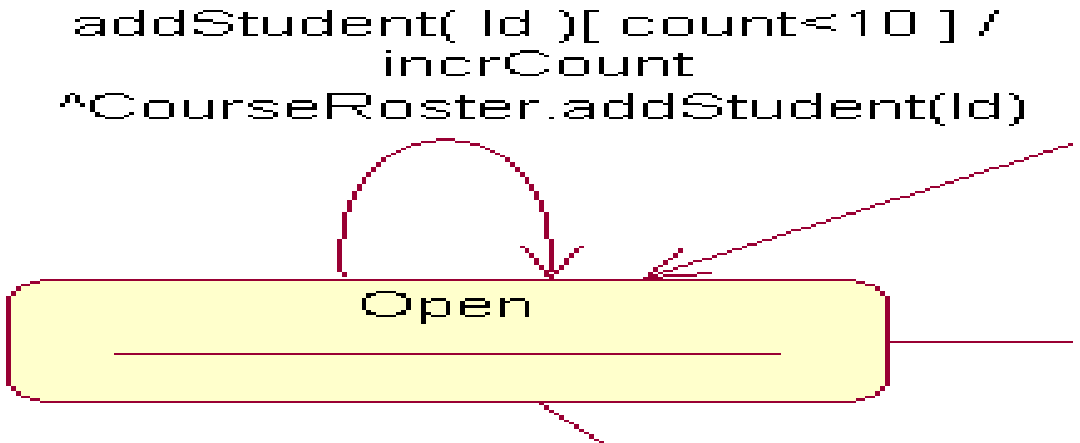
```
event (arguments) [condition] /
action ^target.sendEvent (args)
```

Triggering event: behavior (operation) that occurs in the state transition

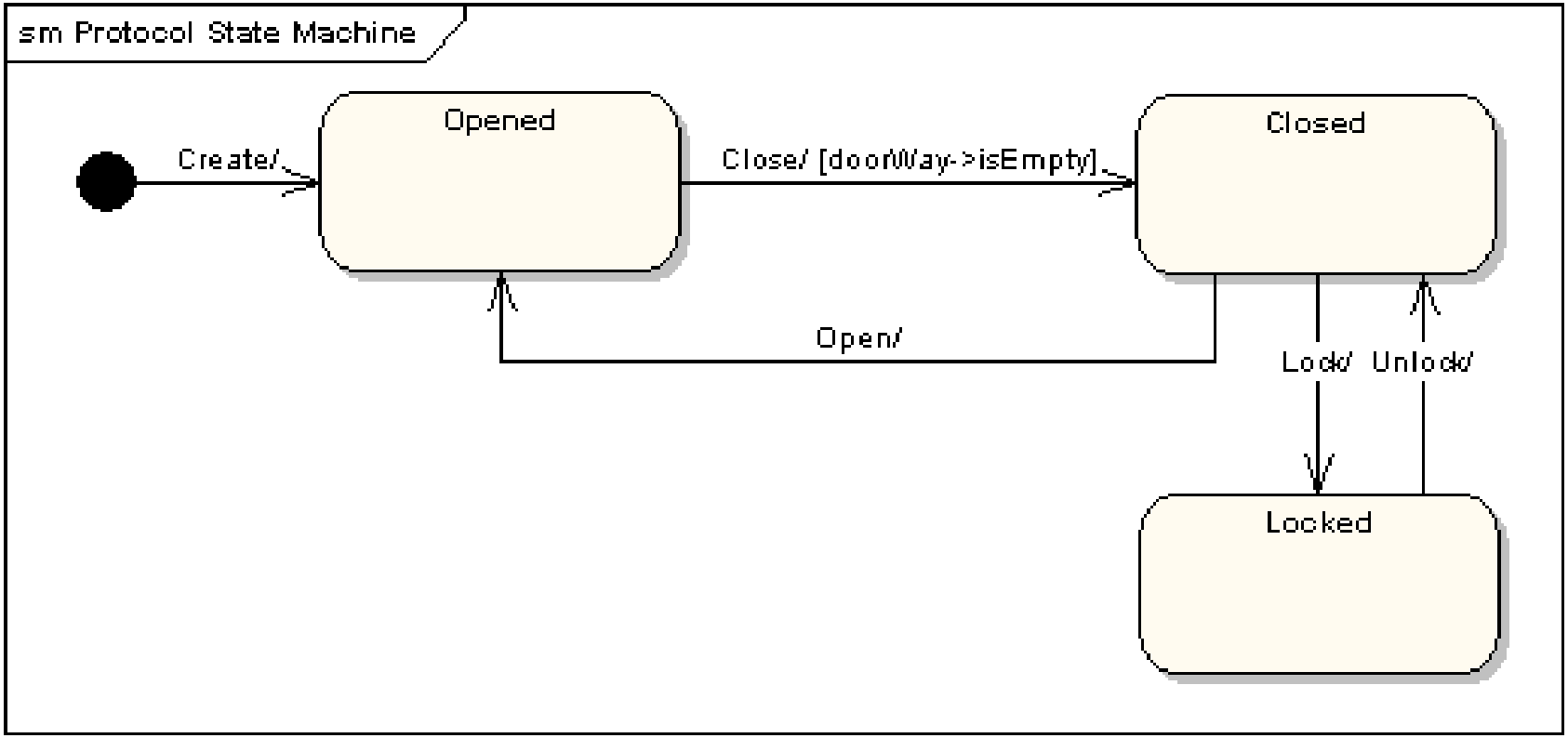
Guard condition: a boolean expression over the attributes that allows the transition

Action: an action over the attributes

Send event: an event invoked in a target object

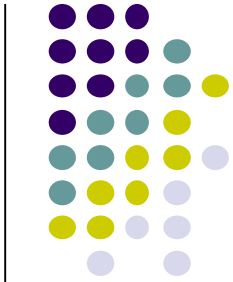


Example (Sparxsystems Ltd.)



N.B.: not all events are valid in all states.

Actions vs Activities

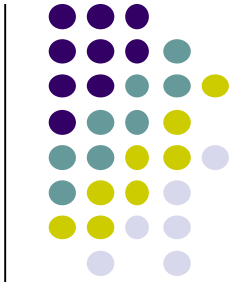


- An action is considered *to take zero time and cannot be interrupted*. It has *an atomic execution and therefore completes without interruption*

- In contrast, an activity is a more complex collection of behavior that may run for a long duration.

- An activity may be interrupted by events, in which case, it does not run to completion.

State Activities



Each state on a statechart (or activity) diagram may contain any number of ***internal activities***.

An activity is best described as a "task" that takes place while inside a state.

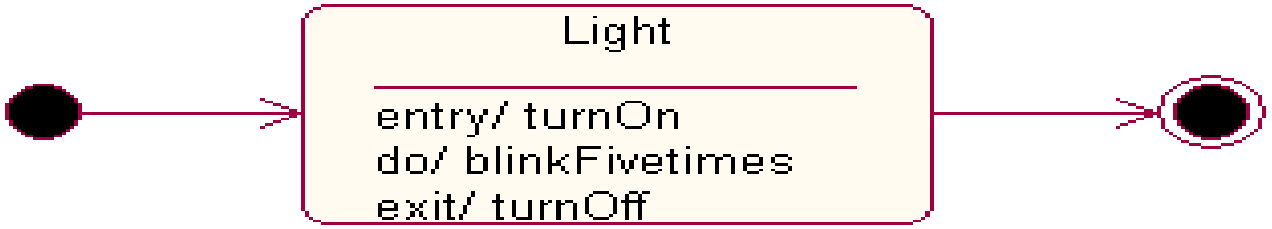
An activity forms an abstraction of a computational procedure.

State activity types

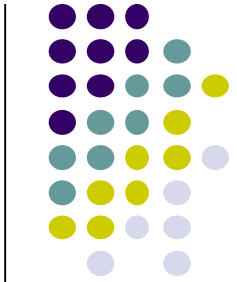
There are four possible activities types within a state:

- ***On Entry*** – the "task" must be performed when the object enters the state
- ***On Exit*** – the "task" must be performed when the object exits the state
- ***Do*** – the "task" must be performed when the object while in the state, until exiting it
- ***On Event*** – the "task" triggers an action on a specific event

Example:



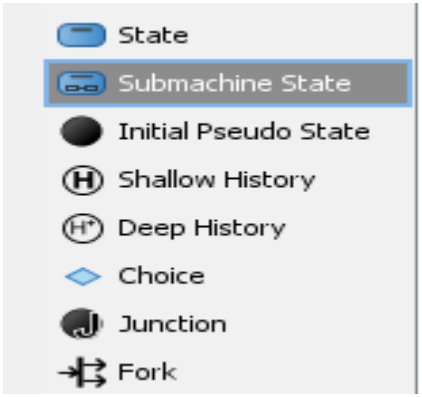
Composite (Nested) States



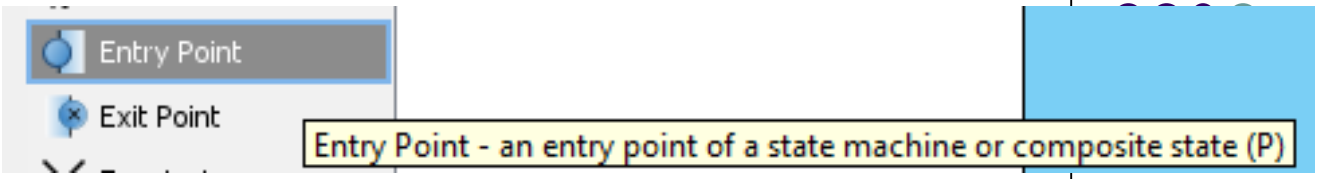
- States may be nested to any depth level.
- Enclosing states are referred to as ***superstates***, and everything that lies within the bounds of the superstate is referred to as its contents.
- Nested states are called ***substates***. Nested states can be moved, resized, hide, and transitioned to/from just as if they were top level states.
- Substates could be sequential (disjoint) or concurrent.
- UML 2.4 defines composite state as the state which contains one or more ***regions***.
- A state is not allowed to have both regions and a ***submachine***.

Submashine State

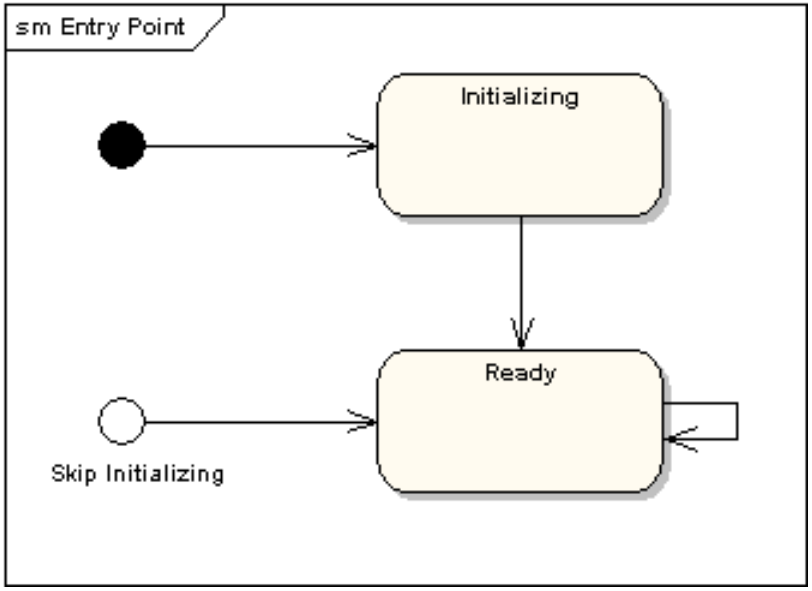
- hides the decomposition of a composite state
- useful in cases of a large number of states nested inside a composite state not fitting in its graphical space
- represented by a simple state graphic with a special "composite" icon
- the contents of the composite state are shown in a separate diagram. The "hiding" is a matter of graphical convenience and has no semantic significance in terms of access restrictions



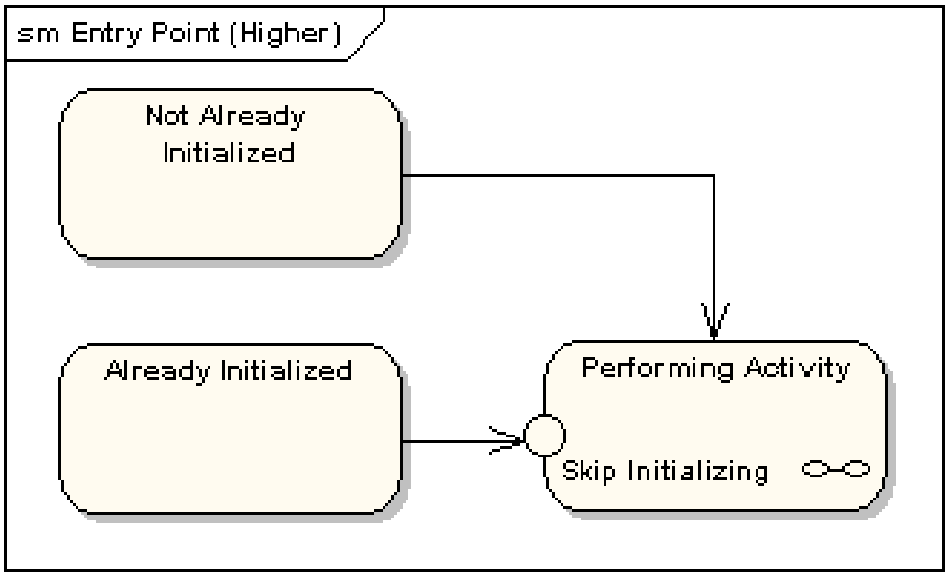
Entry Point



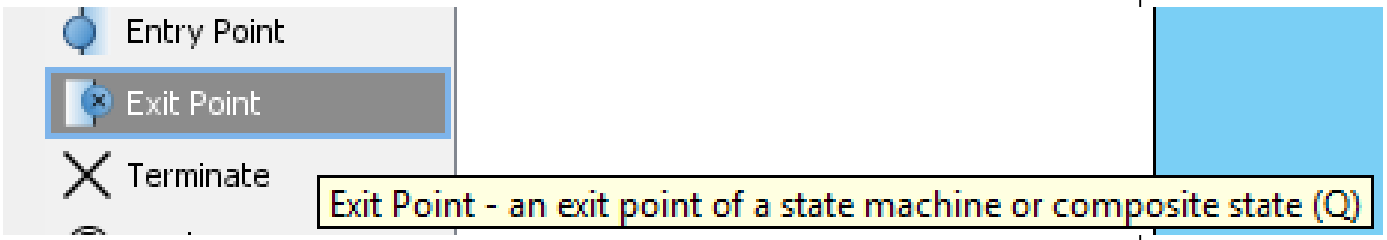
- Sometimes you won't want to enter a sub-machine at the normal initial state. E.g., in the following sub-machine it would be normal to begin in the "Initializing" state, but if for some reason it wasn't necessary to perform the initialization, it would be possible to begin in the "Ready" state by transitioning to the *named entry point.*



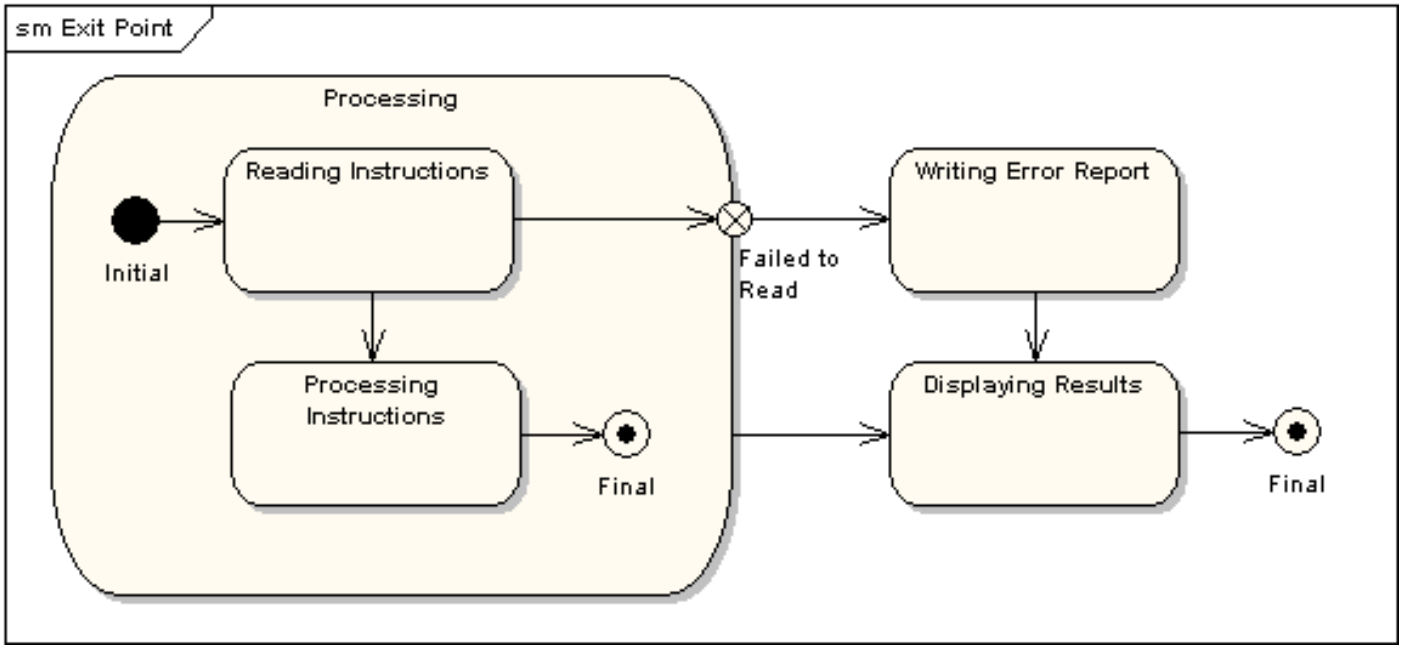
1 level up

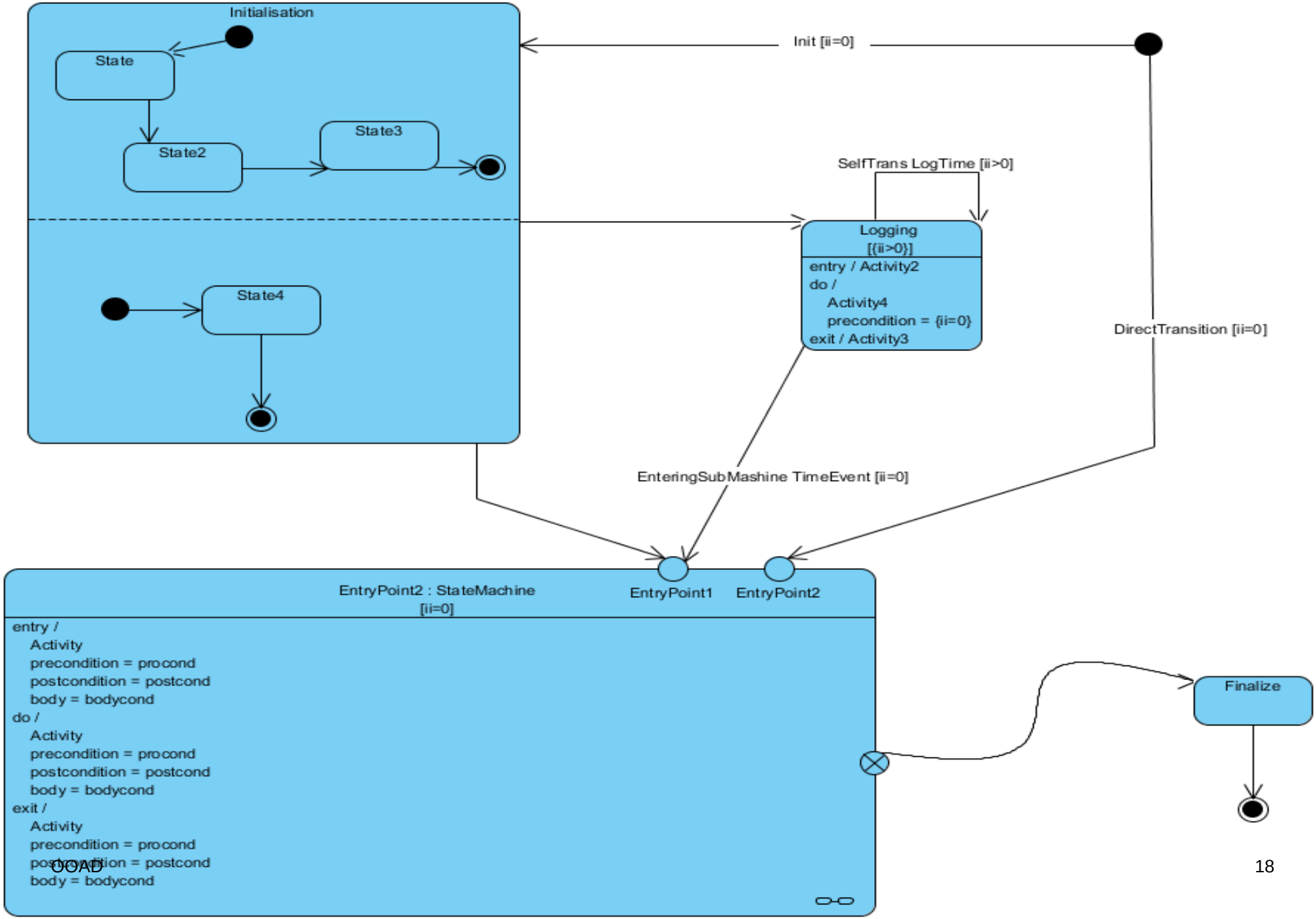


Exit Point

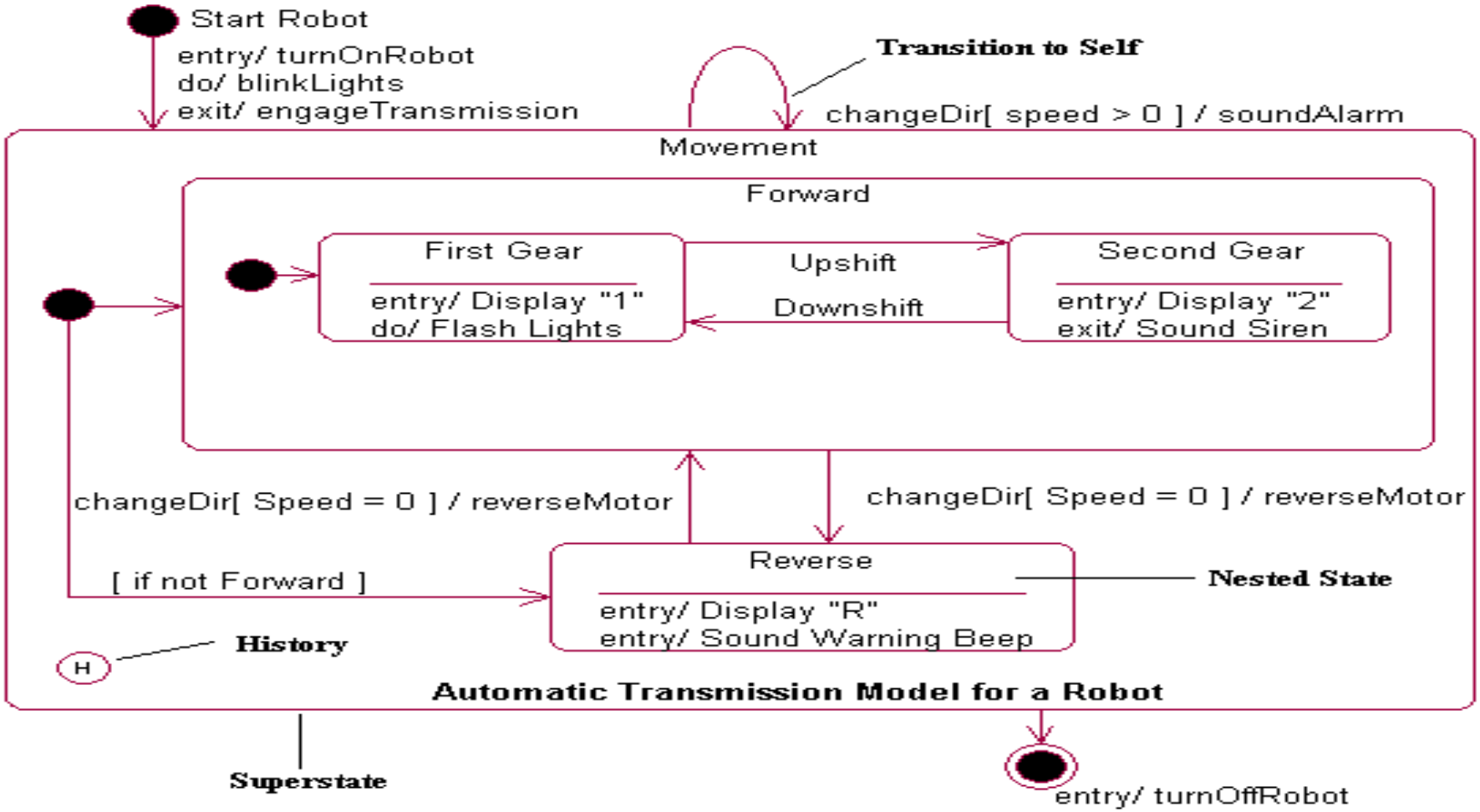


In a similar manner to entry points, it is possible to have named alternative exit points.
Below, the state executed after the main processing state depends on which route is used to transition out of the state.

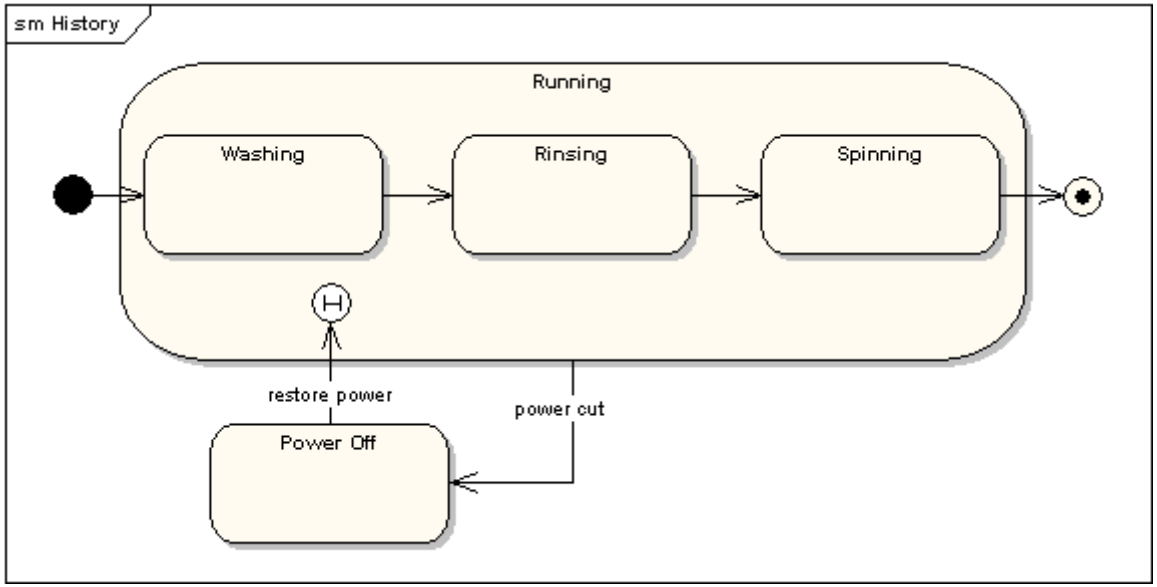



Initialisation
State
State2
State3
State4
Init [ii=0]
SelfTrans LogTime [ii>0]
Logging
[{ii>0}]
entry / Activity2
do /
Activity4
precondition = {ii=0}
exit / Activity3
DirectTransition [ii=0]
EnteringSubMashine TimeEvent [ii=0]
EntryPoint2 : StateMachine
[ii=0]
EntryPoint1
EntryPoint2
entry /
Activity
precondition = procond
postcondition = postcond
body = bodycond
do /
Activity
precondition = procond
postcondition = postcond
body = bodycond
exit /
Activity
precondition = procond
postcondition = postcond
body = bodycond
Finalize

Nested States – a Robot example



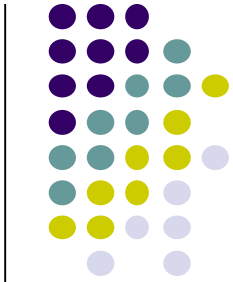
History Pseudo-State



A history state is used to remember the previous state of a state machine when it was interrupted.

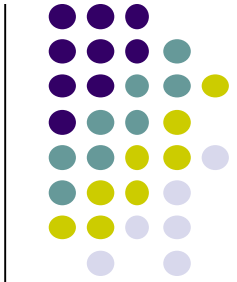
Example: a state machine of a washing machine - when a washing machine is running, it will progress from "Washing" through "Rinsing" to "Spinning". If there is a power cut, the washing machine will stop running and will go to the "Power Off" state. Then when the power is restored, the Running state is entered at the "History State" symbol meaning that it should resume where it last left-off.

Shallow history pseudostate



- **Shallow history pseudostate (H)** represents the most recent active substate of its containing state (but not the substates of that substate).
- A composite state can have at most one shallow history vertex.
- A transition coming into the shallow history vertex is equivalent to a transition coming into the most recent active substate of a state.
- At most one transition may originate from the history connector to the default shallow history state.
- The entry action of the state represented by the shallow history is performed.

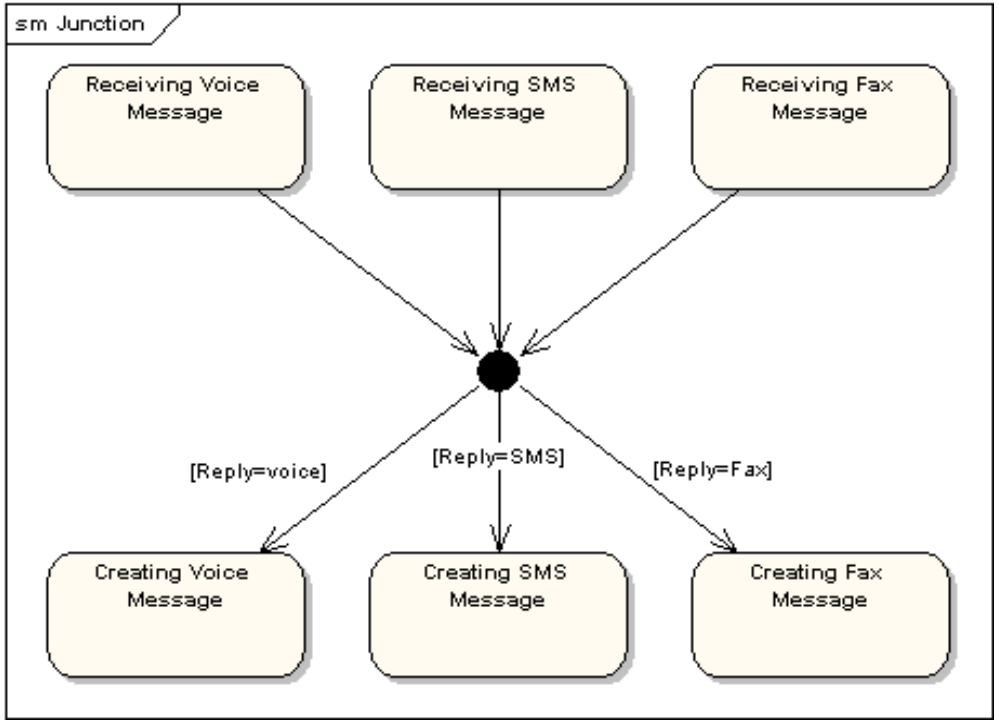
Deep history pseudostate



- **Deep history pseudostate (H*)** represents the most recent active configuration of the composite state that directly contains this pseudostate (the state configuration that was active when the composite state was last exited).
- A composite state can have at most one deep history vertex.
- At most one transition may originate from the history connector to the default deep history state.
- Entry actions of states entered on the implicit direct path from the deep history to the innermost state(s) represented by a deep history are performed.
- The entry action is preformed only once for each state in the active state configuration being restored.

Junction Pseudo-State

- Junction pseudo-states are used to chain together multiple transitions.
- A single junction can have one or more incoming, and one or more outgoing, transitions; a guard can be applied to each transition.
- In the figure below, three message flows are joint and, next, three guard conditions are applied to each of them.



Incident state diagram [Bruegge&Dutoit]

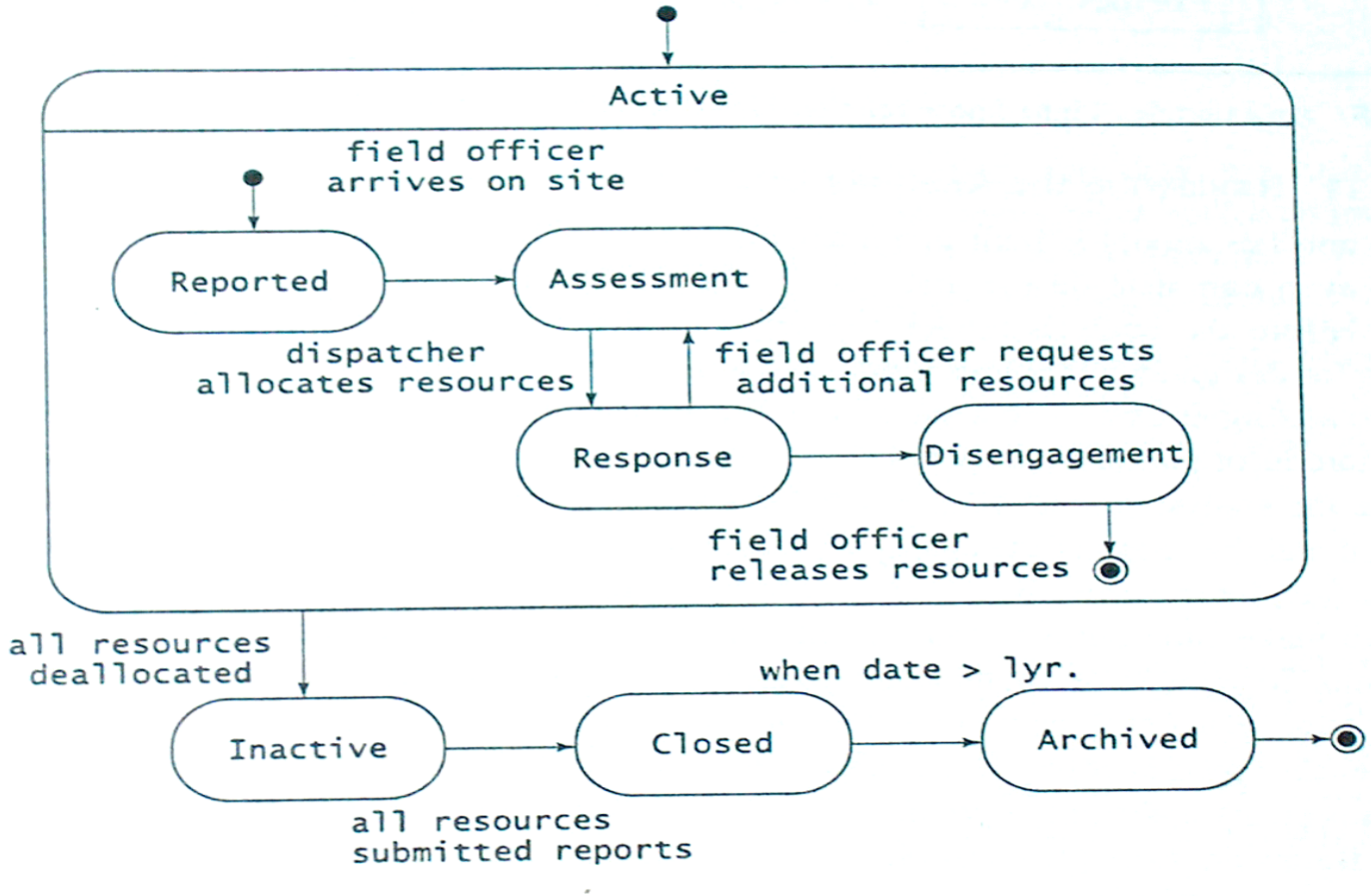
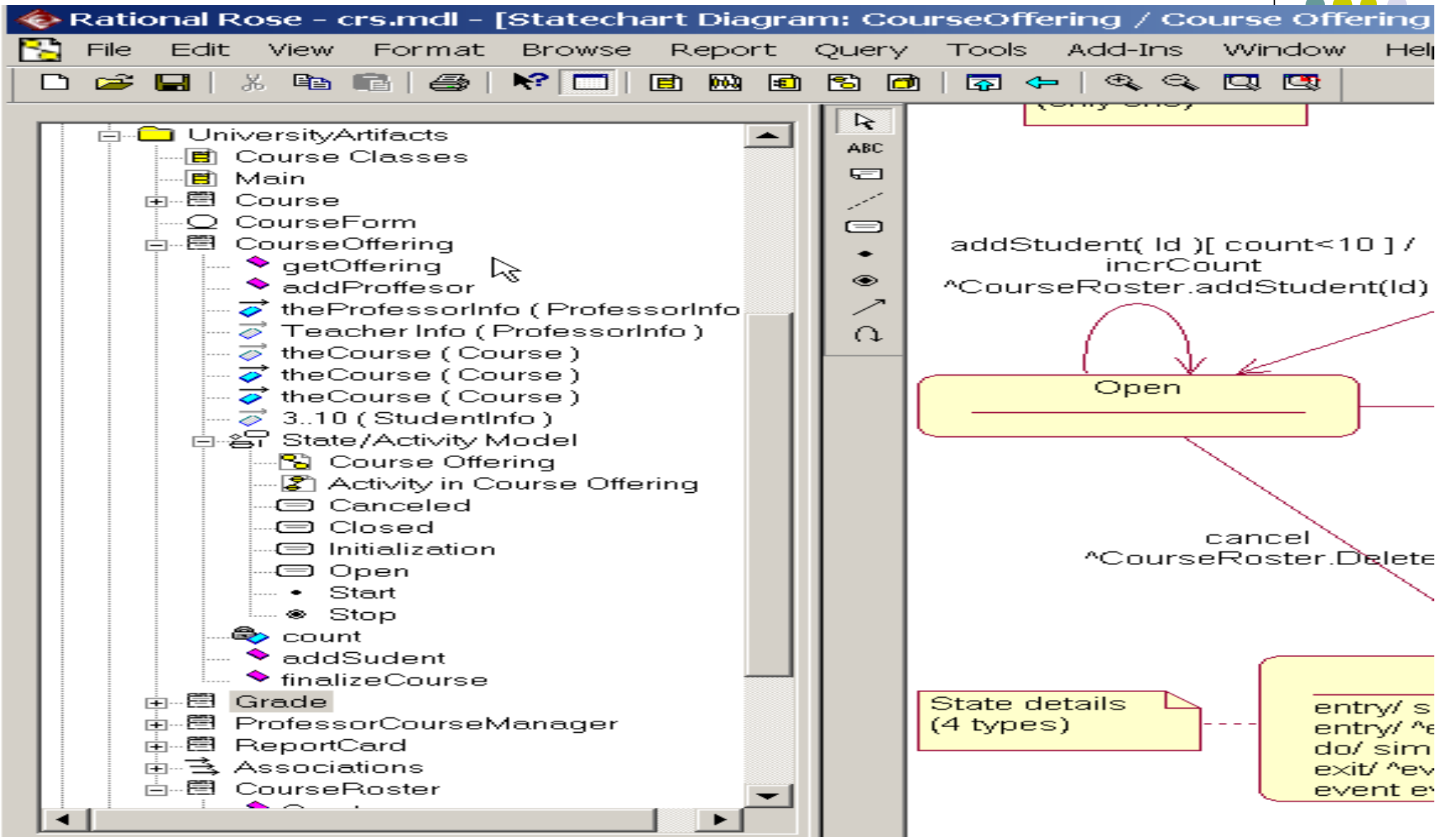
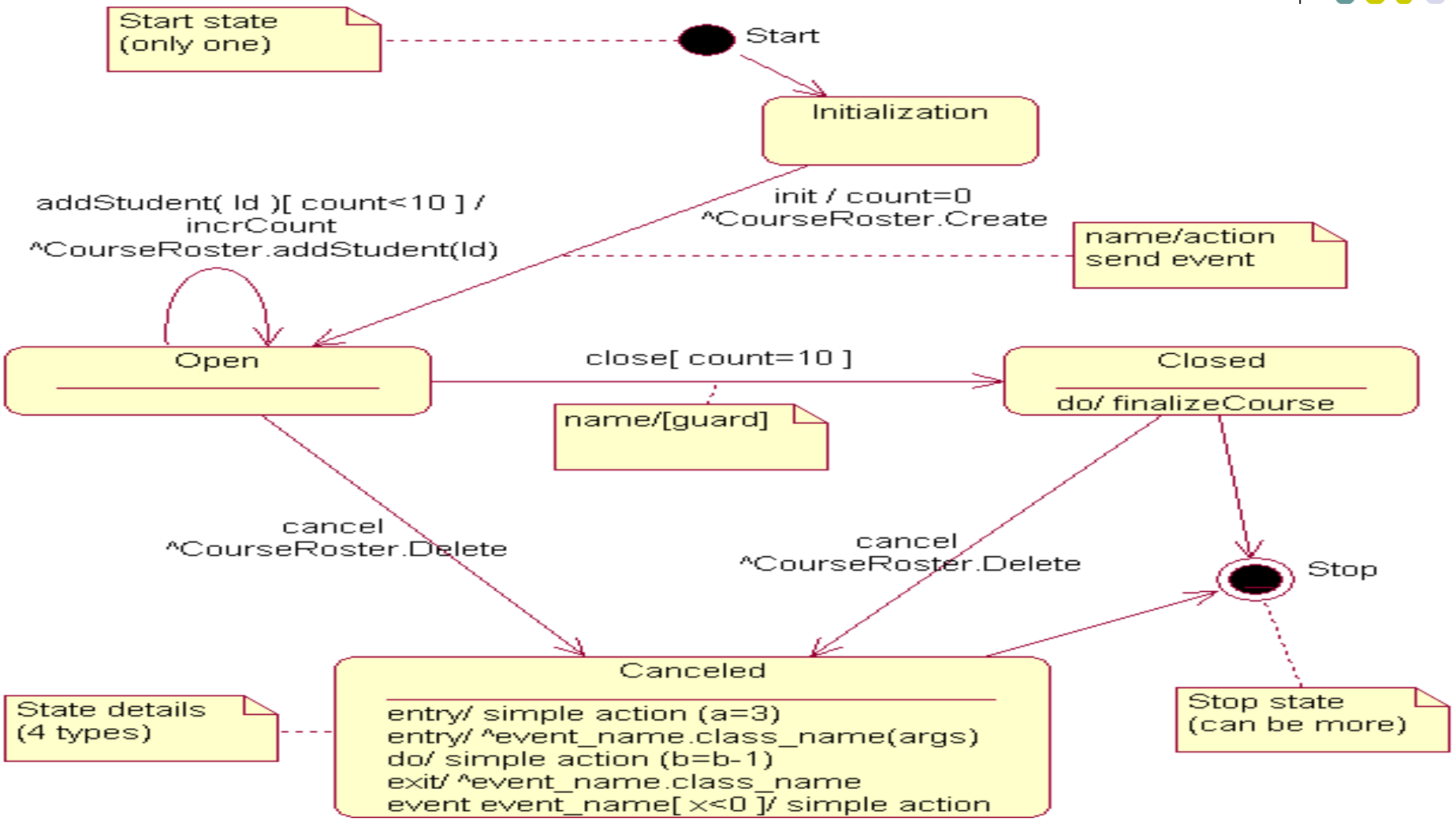


Figure 5-17 UML statechart for Incident.

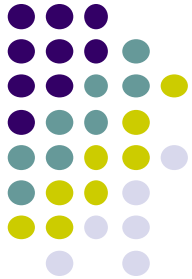
Case study: UCR – the Course Offering Class



Case study1: UCR – the Course Offering Statechart



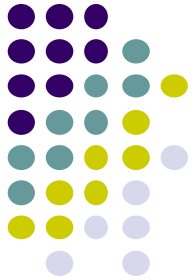
Case study 2: Thread states and life cycle in Java 6 1/4



- **New** is the thread state for a thread which was created but has not yet started.
- At OS level, JVM's **runnable** state could be considered as a composite state with two sub-states. When a thread transitions to the runnable JVM state, the thread first goes into the **ready** sub-state. Thread scheduling decides when the thread could actually start, proceed or be suspended. Thread.yield() is explicit recommendation to thread scheduler to pause the currently executing thread to allow some other thread to execute.
- A thread in the **runnable** state is executing from the JVM point of view but in fact it may be waiting for some resources from the operating system.

Source: http://www.uml-diagrams.org/examples/java-6-thread-state-machine-diagram-example.html

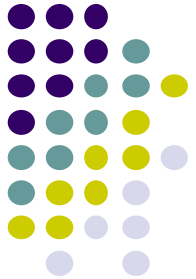
Case study 2: Thread states and life cycle in Java 6 2/4



- **Timed waiting** is a thread state for a thread waiting with a specified waiting time. A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time:
  - Thread.sleep(sleeptime)
  - Object.wait(timeout)
  - Thread.join(timeout)
- A thread is in the **waiting** state due to the calling one of the following methods without timeout:
  - Object.wait()
  - Thread.join()
  - LockSupport.park()

Source: http://www.uml-diagrams.org/examples/java-6-thread-state-machine-diagram-example.html

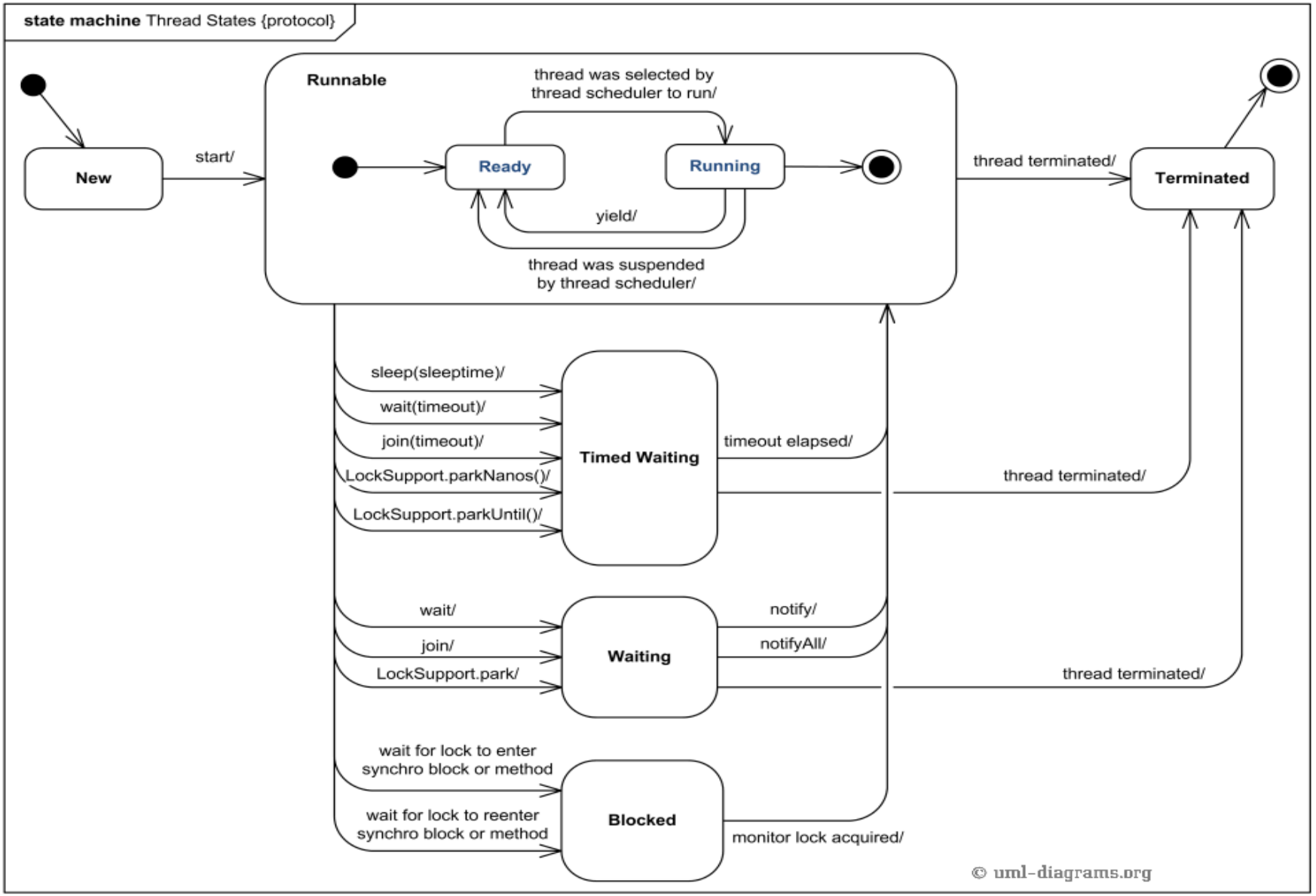
Case study 2: Thread states and life cycle in Java 6 3/4



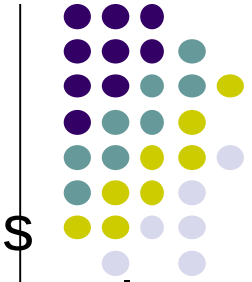
- A thread in the waiting state is waiting for another thread to perform a particular action. For example, a thread that has called Object.wait() on an object is waiting for another thread to call Object.notify() or Object.notifyAll() on that object. It means that waiting state could be made a composite state with states corresponding to these specific conditions.
- Thread is in the **blocked** state while waiting for the monitor lock to enter a synchronized block or method or to reenter a synchronized block or method after calling Object.wait().
- A **synchronized** statement or method acquires a mutual-exclusion lock on behalf of the executing thread, executes a block or method, and then releases the lock.
- After thread has completed execution of run() method, it is moved into **terminated** state.

Source: http://www.uml-diagrams.org/examples/java-6-thread-state-machine-diagram-example.html

Case study 2: Thread states and life cycle in Java 6 4/4



Activity diagrams



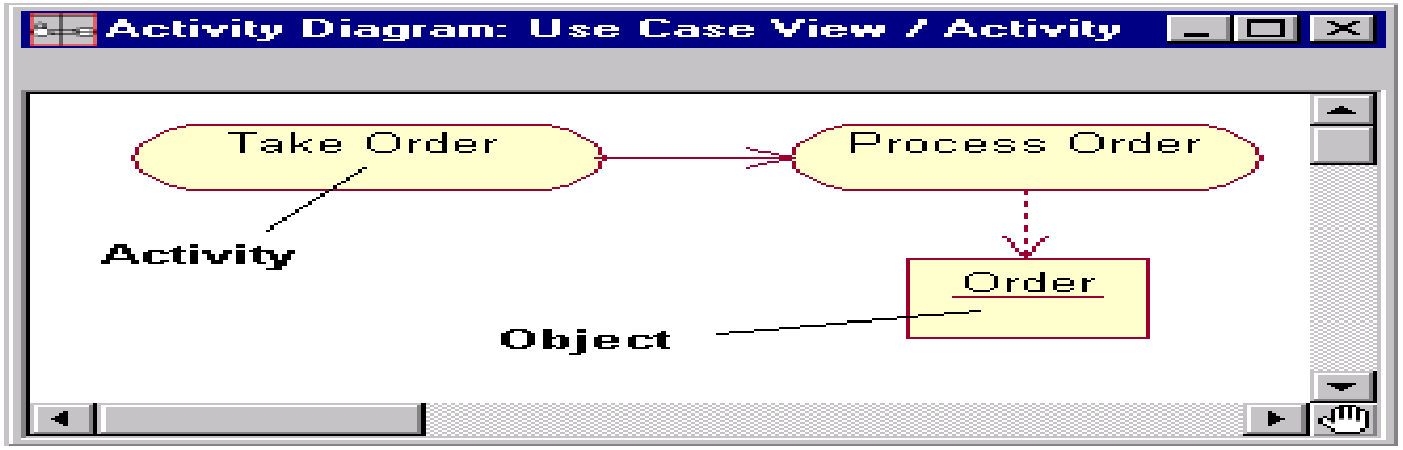
Provide a way to model the *workflow* (sequence of activities producing observable value) of a business process. You can also use activity diagrams to model code-specific information such as a class operation as using flowcharts.
An activity diagram is basically *a special case of a state machine in which most of the states are activities and most of the transitions are implicitly triggered by completion of the actions in the source activities.*

The main difference between activity diagrams and statecharts is activity diagrams are *activity centric*, while statecharts are *state centric*.
An activity diagram is composed by *activities, actions, objects, object flows, pins, decisions, synchronizations, swimlanes, states* and *transitions.*

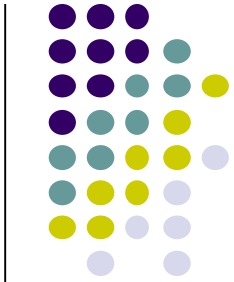
Activities vs States

Activity represent the performance of task or duty in a workflow. It may also represent the execution of a statement in a procedure. ***An activity is similar to a state, but expresses the intent that there is no significant waiting (for events) in an activity***. Transitions connect activities with other model elements and object flows connect activities with objects.

The activity icon appears as a rectangle with rounded ends with a name (Take Order and Process Order) and a compartment for actions.

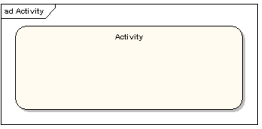


Transitions in activity diagrams

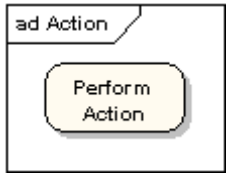


- Transitions in an activity diagram do not have labels
  - They indicate the completion of an action or subactivity and show the sequence of actions or subactivities
  - Consequently, these transitions are not based on external events
- An activity diagram may describe *a use case, an operation or a message*
  - Purpose: to describe implementation-oriented details

Activity versus action



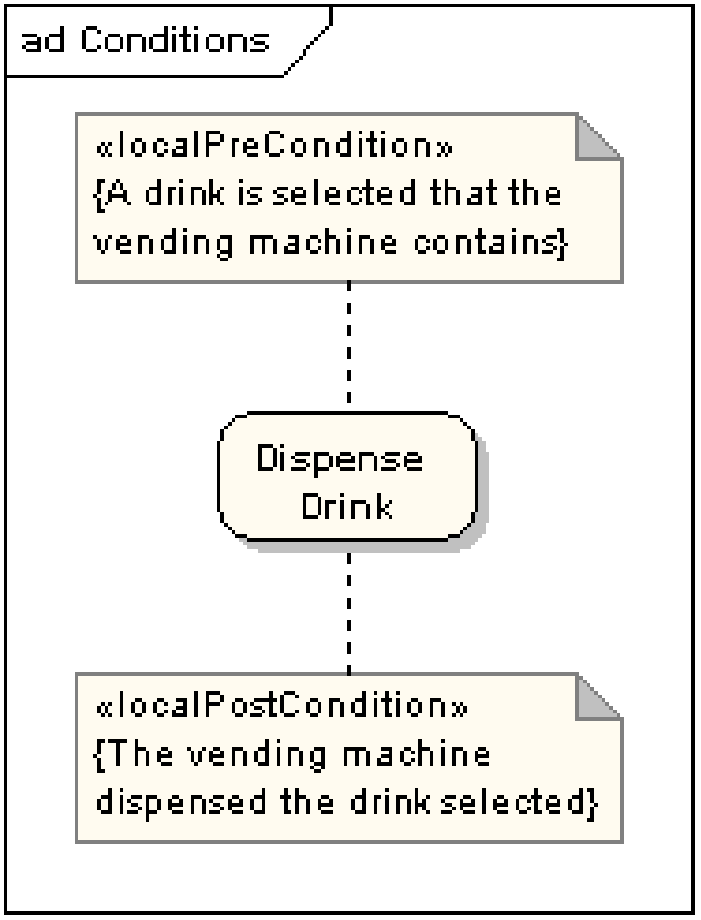
- Activity : A sequence of actions that take finite time and can be interrupted; the specification of a parameterized sequence of behavior. An activity is shown as a round-cornered rectangle enclosing all the actions, control flows and other elements that make up the activity



- Action: An atomic task that cannot be interrupted (at least from user's perspective). An action represents a single step within an activity. Actions are denoted by round-cornered rectangles. An ***action state*** (UML 1.*) represents the execution of an atomic action, typically the invocation of an operation. ***ActionState*** has been replaced, as of UML 2.0, by ***Action***.

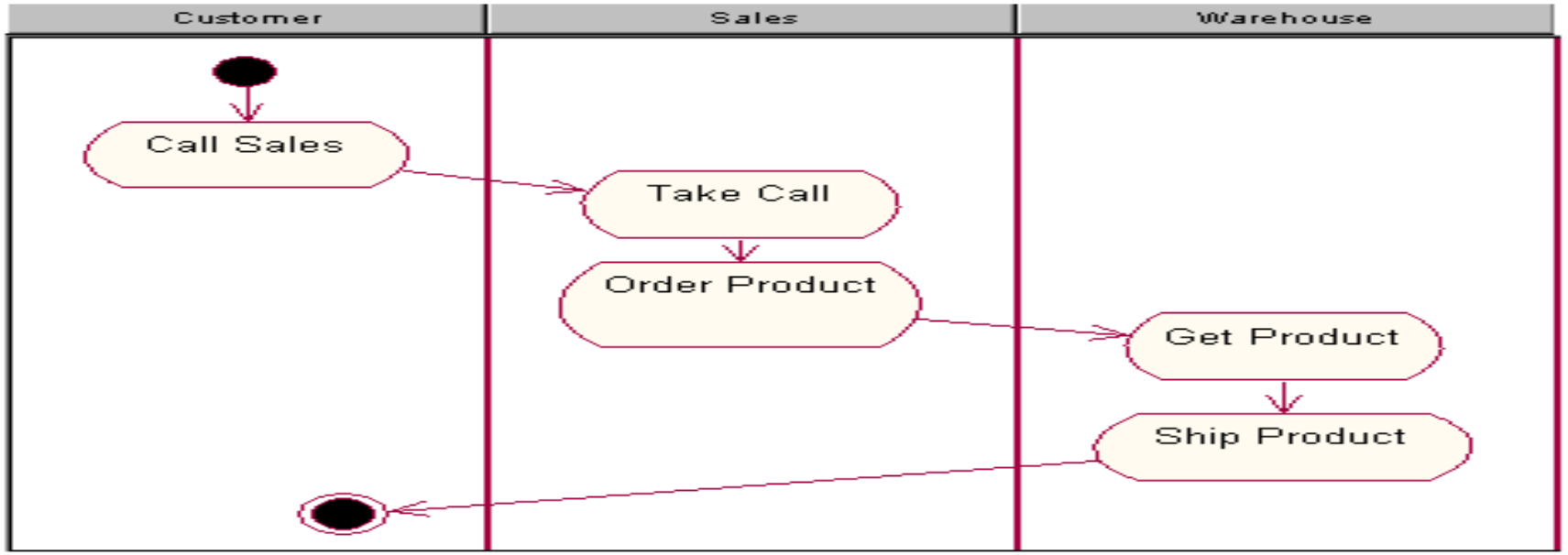
Action constraints

- Constraints can be attached to an action.
- The diagram right-side shows an action with local pre- and post-conditions.



Swimlanes (partitions)

Swimlanes only appear on activity diagrams and determine which unit is responsible for carrying out the specific activity.

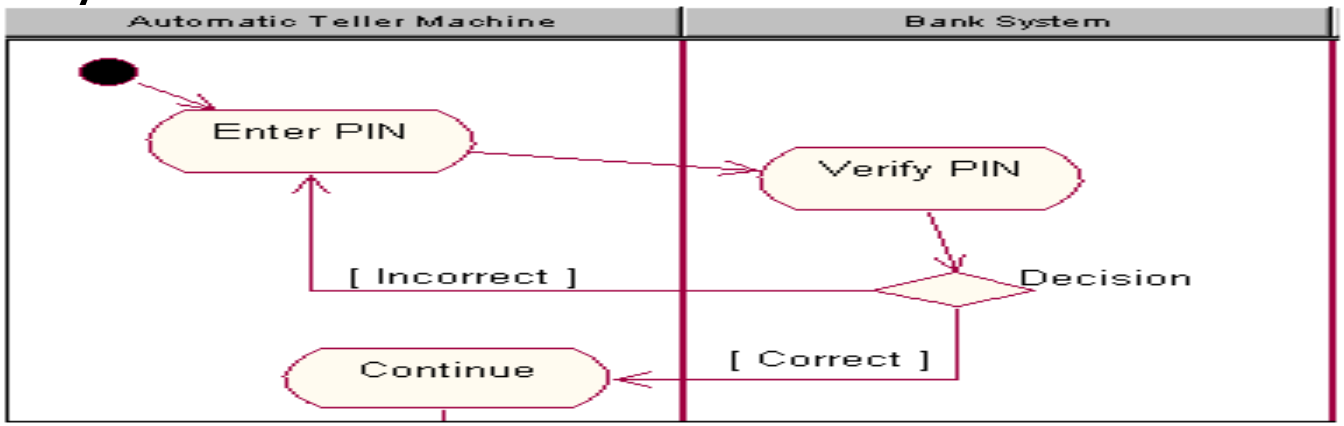


Example: *Get Product* and *Ship Product* activities reside within the *Warehouse* swimlane indicating that the warehouse is responsible for getting the correct product and then shipping the product to the customer. The workflow ends when the customer (noted through the *Customer* swimlane) receives the product.

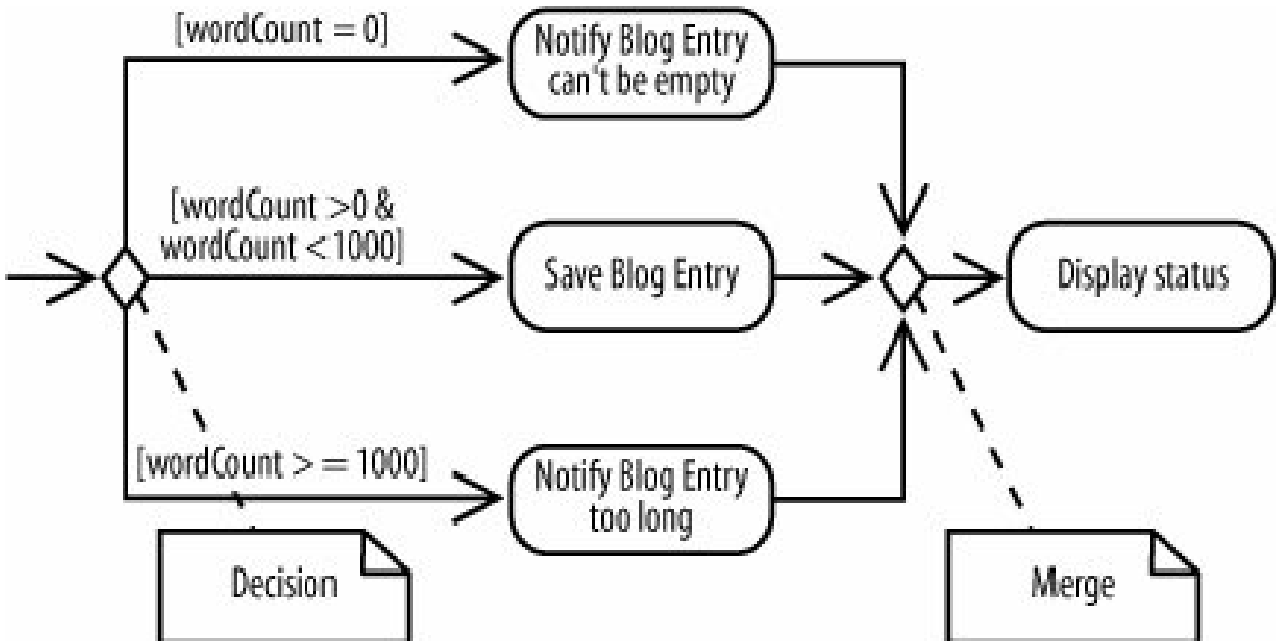
Decision nodes

A decision represents a specific location where the workflow may branch based upon guard conditions. There may be **more than two outgoing transitions** with different guard conditions, but for the most part, a decision will have only two outgoing transitions determined by a Boolean expression.

The following figure displays a decision with [correct] and [incorrect] as the guard conditions. If the personal identification number (PIN) is incorrect, the flow of control goes back to the Enter PIN activity. If it is correct, the flow of control moves to the Continue activity.



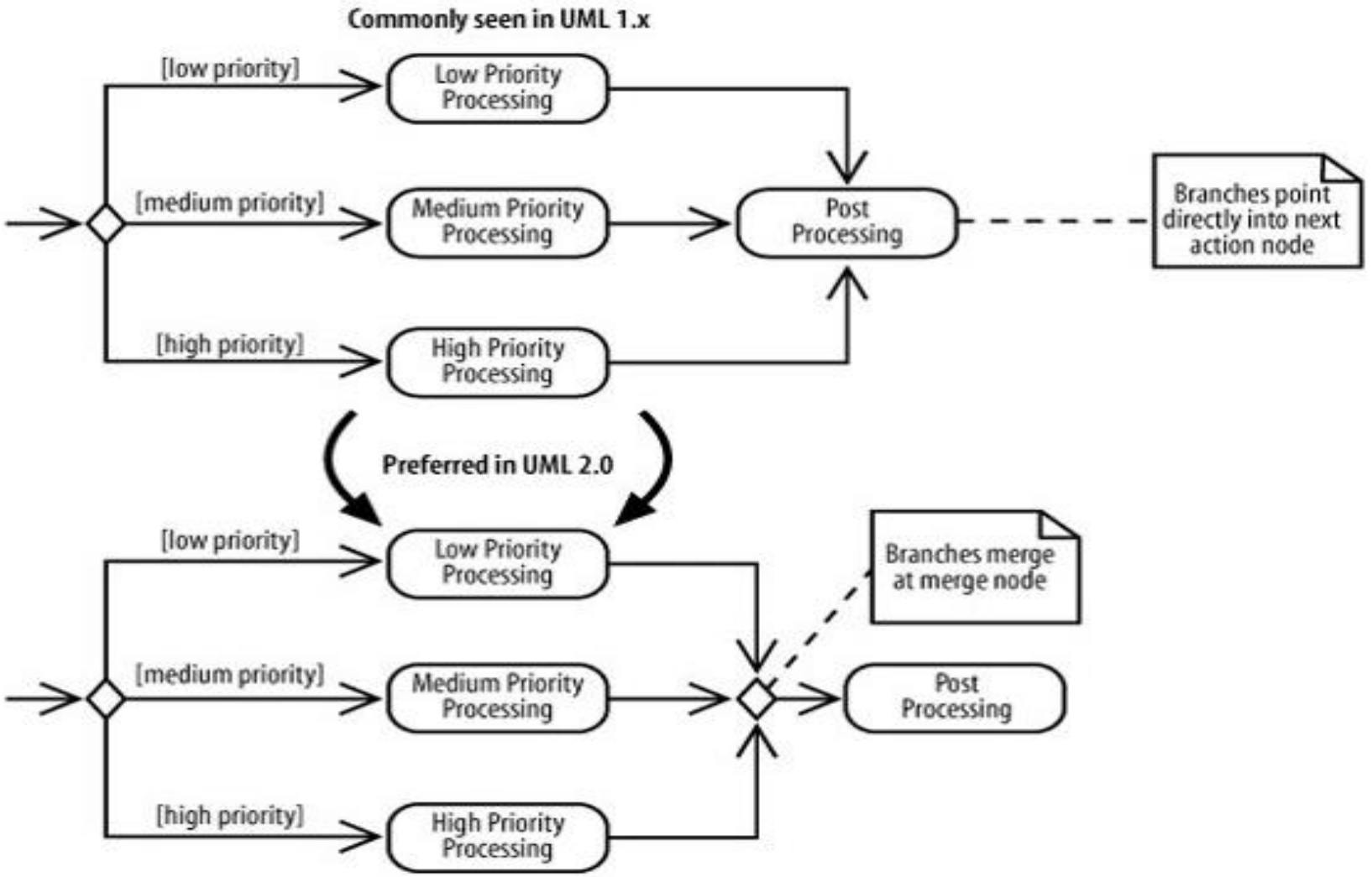
Branch Factor May Be Greater Than 2



An outgoing transition from an action state may end up in a condition box (a diamond symbol) and hence will be split into 2 or 3 transitions

The conditions must be MUTUALLY EXCLUSIVE

Merging Nodes

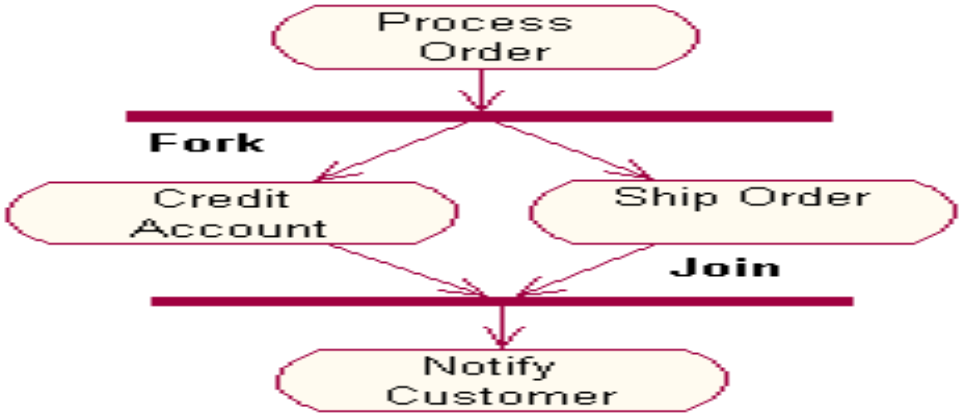


Synchronizations

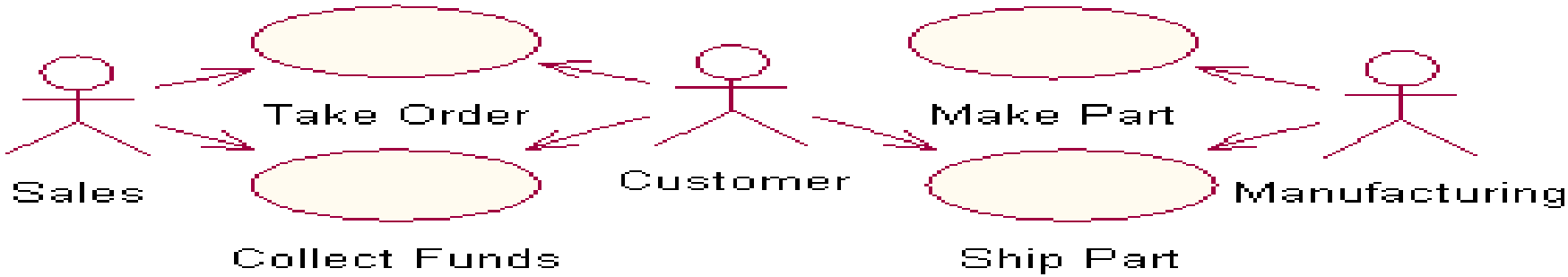
Synchronizations enable you to see a simultaneous workflow. Synchronizations visually define forks and joins representing parallel workflow.

A ***fork*** construct is used to model a single flow of control that divides into two or more separate, but simultaneous flows.

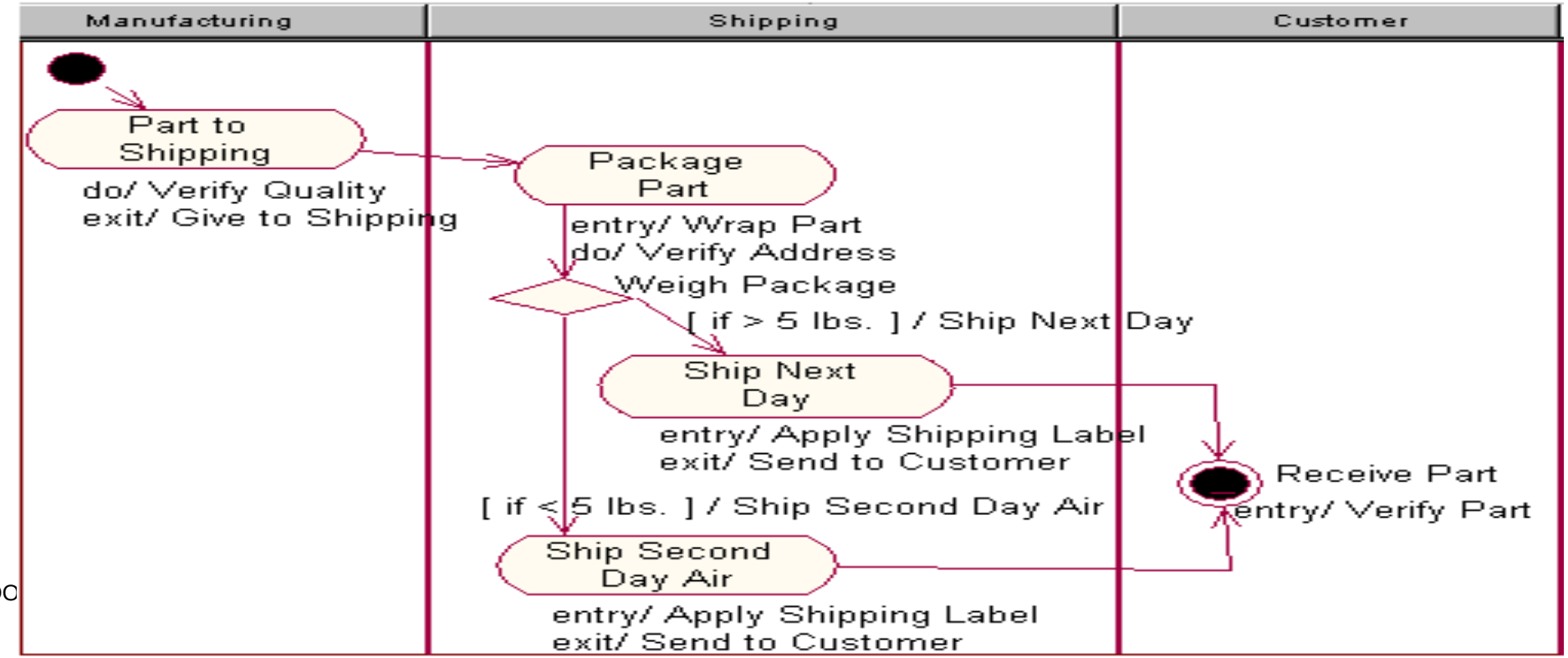
A ***join*** consists of two of more flows of control that unite into a single flow of control. ***All activities*** and states that appear between a fork and join must complete before the flow of controls can unite into one.



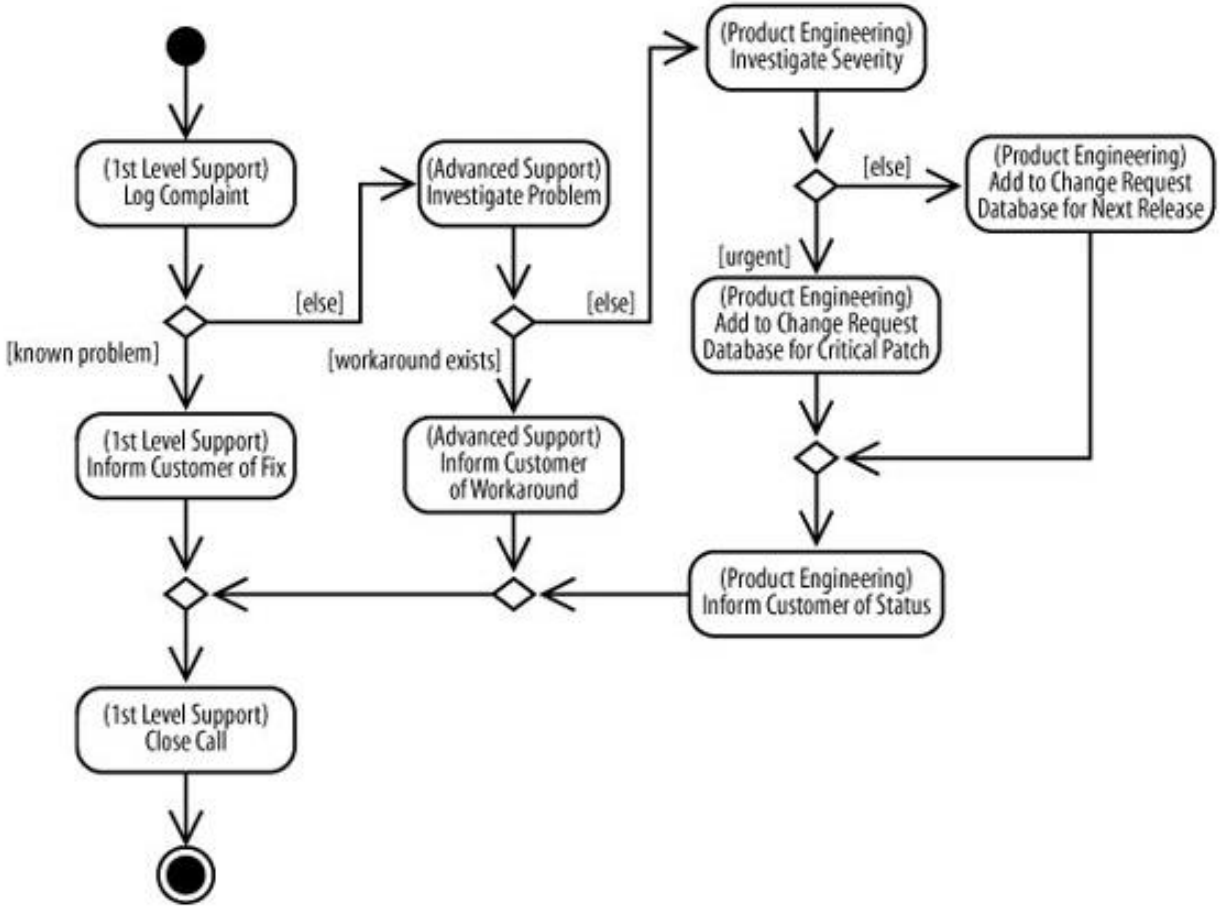
Activity Diagram of a Use-Case Sample



Ordering, making, paying for, and shipping a part for a machine



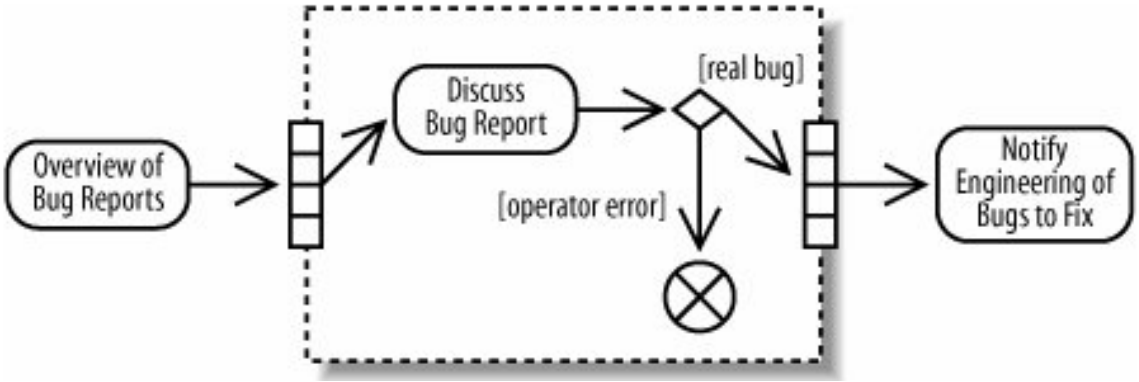
Annotations vs swimlanes



- Annotations can be used instead of swimlanes as a *way of showing responsibility directly in the action*

Expansion Region

- *Expansion regions* show that actions in a region are performed for each item in an input collection.
- For example, an expansion region could be used to model a software function that takes a list of files as input and searches each file for a search term.

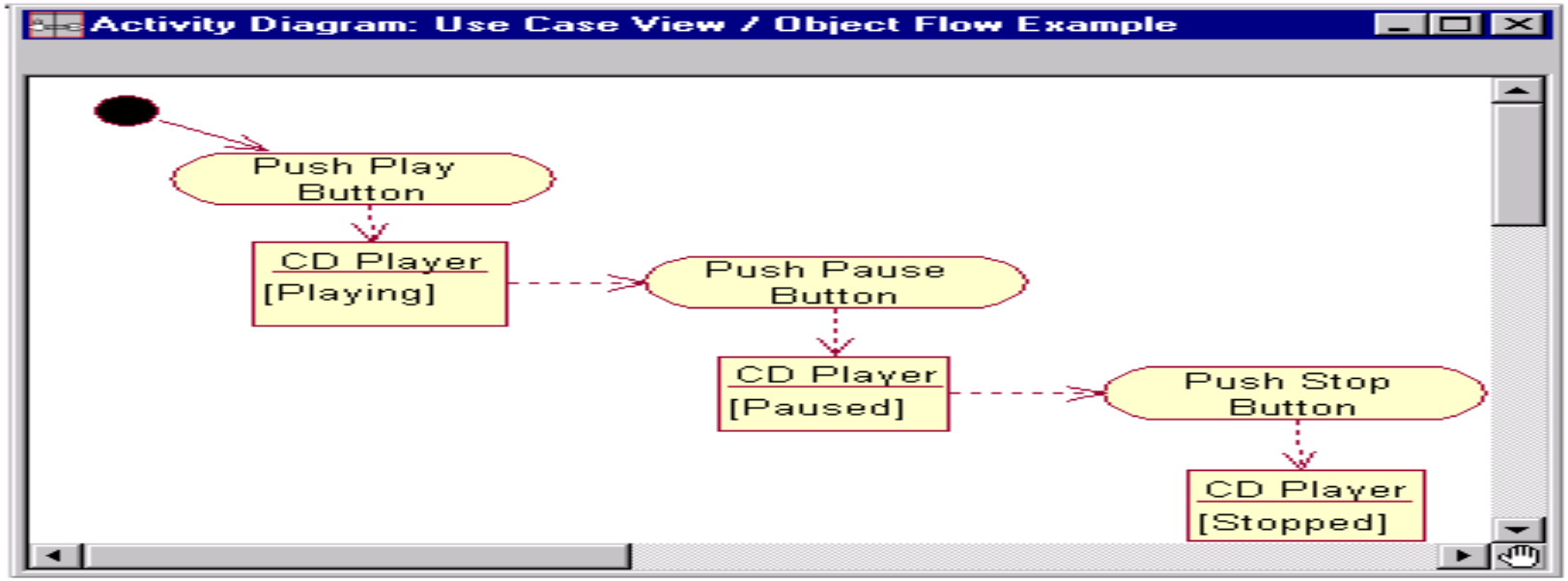


Object Flows (since UML 1.5)

An *object flow* on an activity diagram represents the relationship between an activity and the object that **creates it** (as an output) or **uses it** (as an input).

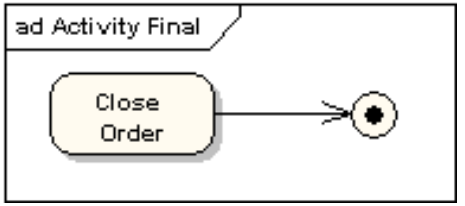
Some UML editors draws object flows as dashed arrows rather than solid arrows to distinguish them from ordinary transitions.

- objects may appear more than once and in several states
- activities may change object state
-

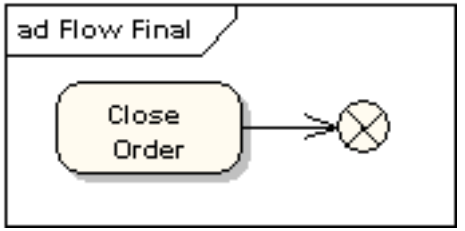


Flow and activity final nodes

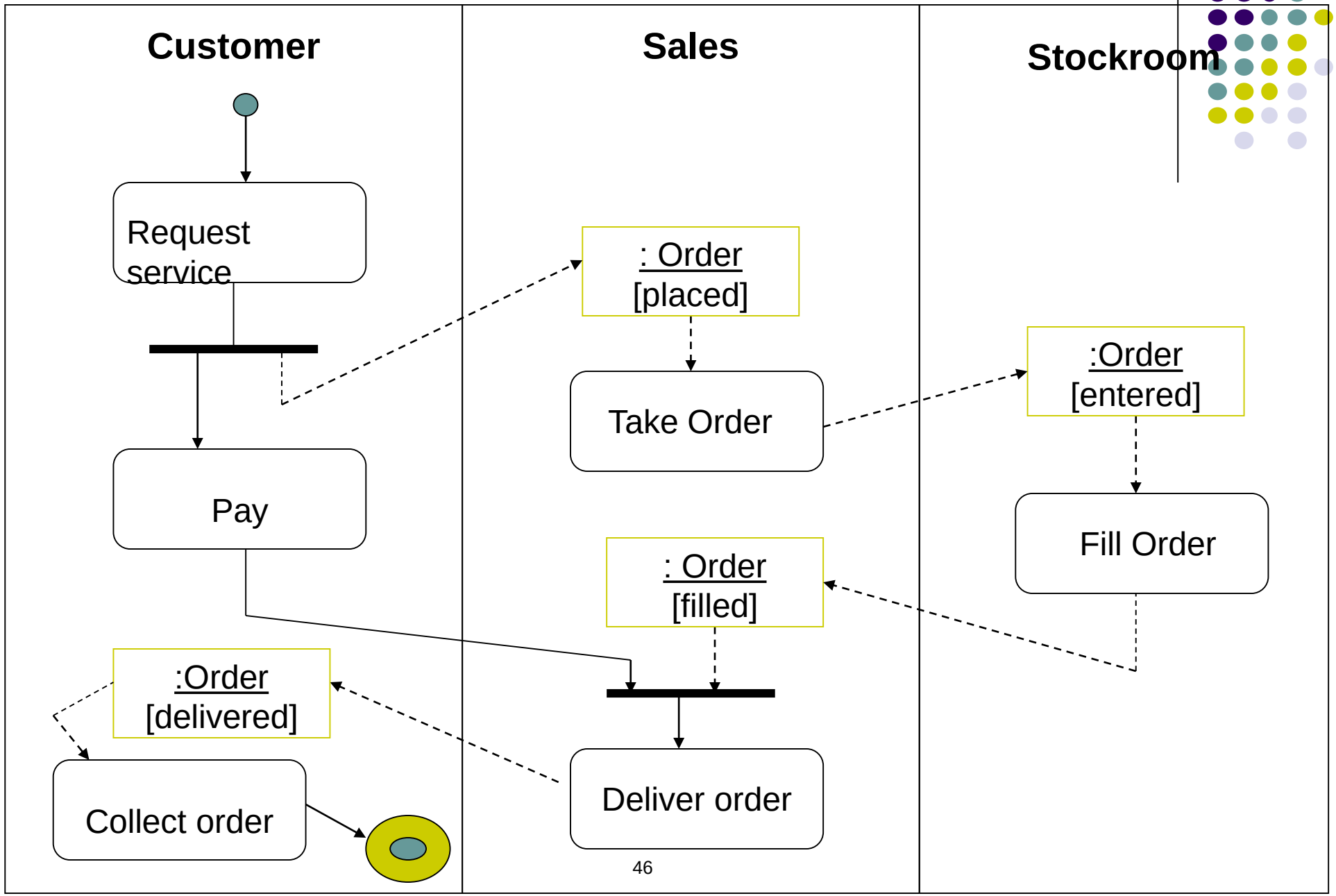
There are two types of final node: activity and flow final nodes.



- The activity final node is depicted as a circle with a dot inside.

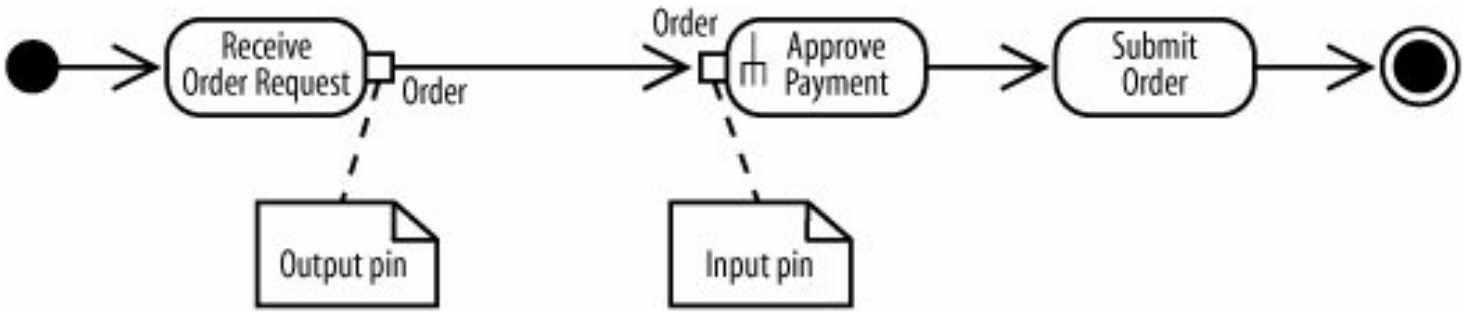


- The flow final node is depicted as a circle with a cross inside.
- The difference between the two node types is that the flow final node denotes the end of a single control flow; the activity final node denotes the end of all control flows within the activity.


Customer
Sales
Stockroom
Request service
Pay
:Order [delivered]
Collect order
: Order [placed]
Take Order
: Order [filled]
Deliver order
:Order [entered]
Fill Order

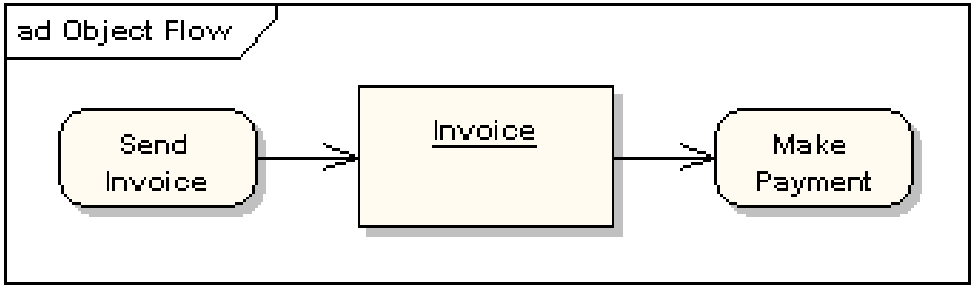
Pins (UML 2.x)

- An *input pin* means that the specified object is input to an action.
- An *output pin* means that the specified object is output from an action.

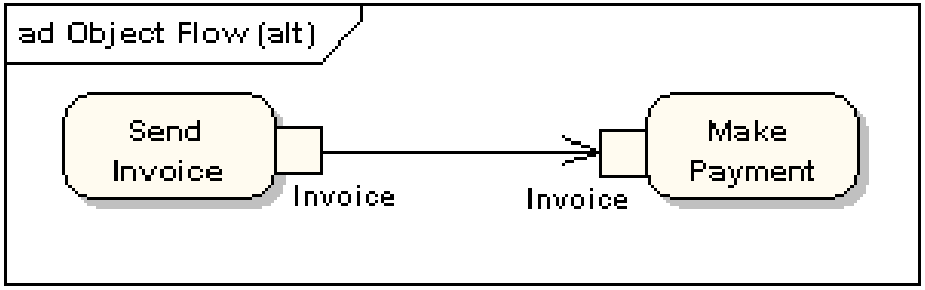


Pins instead object flow

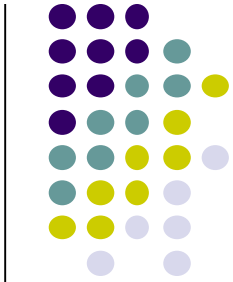
An object flow is shown as a connector with an arrowhead denoting the direction the object is being passed.



An object flow must have an object on at least one of its ends. A shorthand notation for the above diagram would be to use input and output pins.



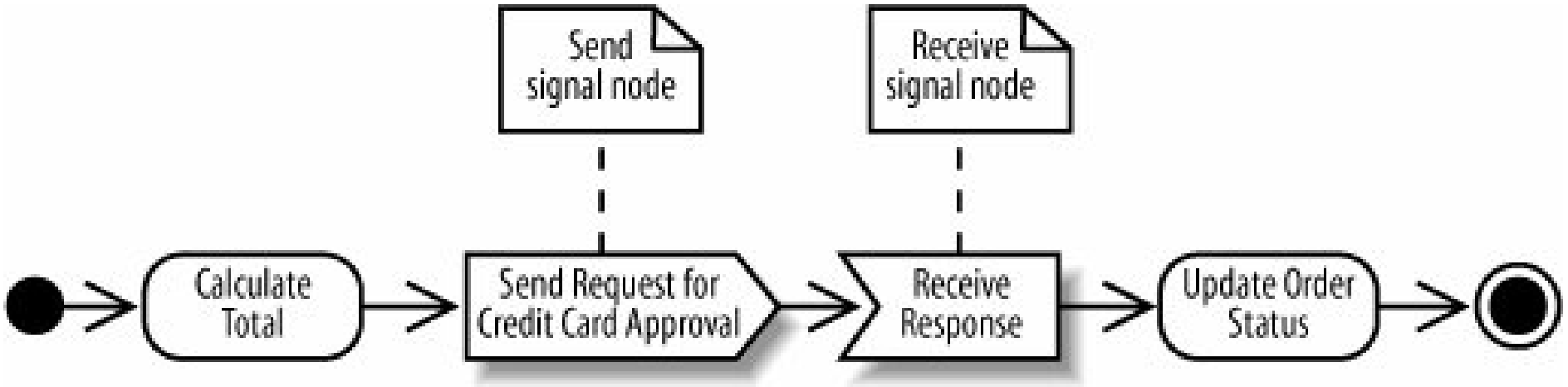
Send and Receive signals 1/2



In activity diagrams, *signals* represent interactions with external participants. Signals are messages that can be **sent** or **received**, e.g.:

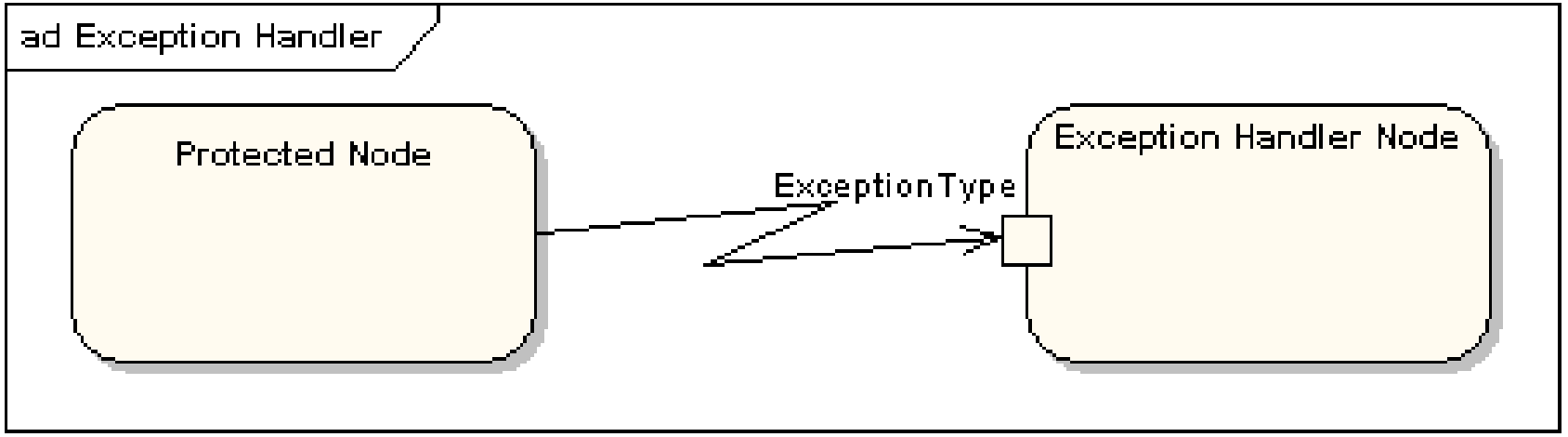
- The receipt of an order prompts an order handling process to begin (**received**, from the perspective of the order handling activity).
- The click of a button causes code associated with the button to execute (**received**, from the perspective of the button event handling activity).
- The system notifies a customer that his shipment has been delayed (**sent**, from the perspective of the order shipping activity).

Send and Receive signals 2/2



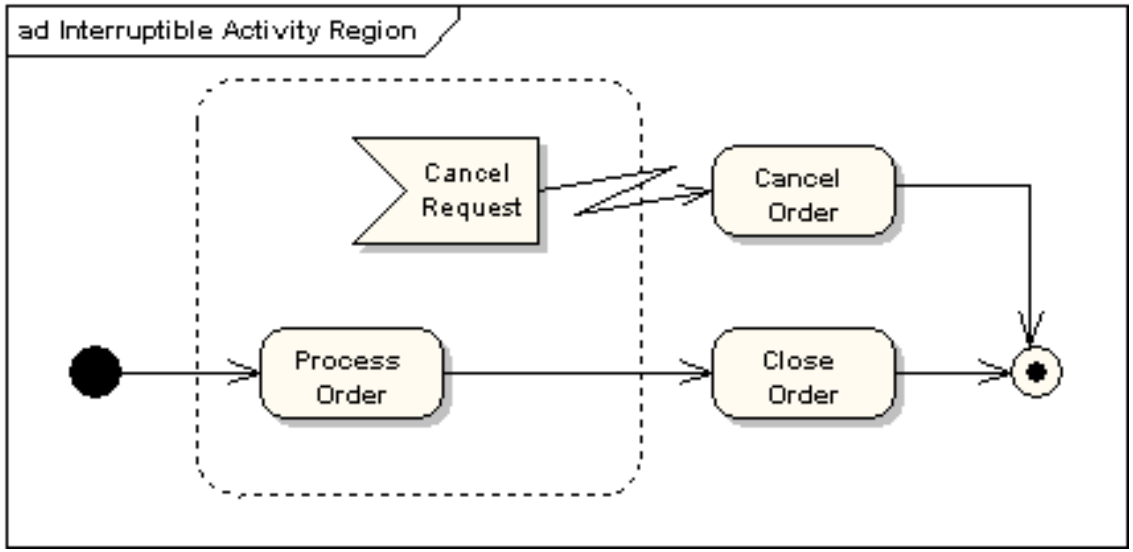
Exception handlers

- Exception handlers can be modeled on activity diagrams as in the example below.



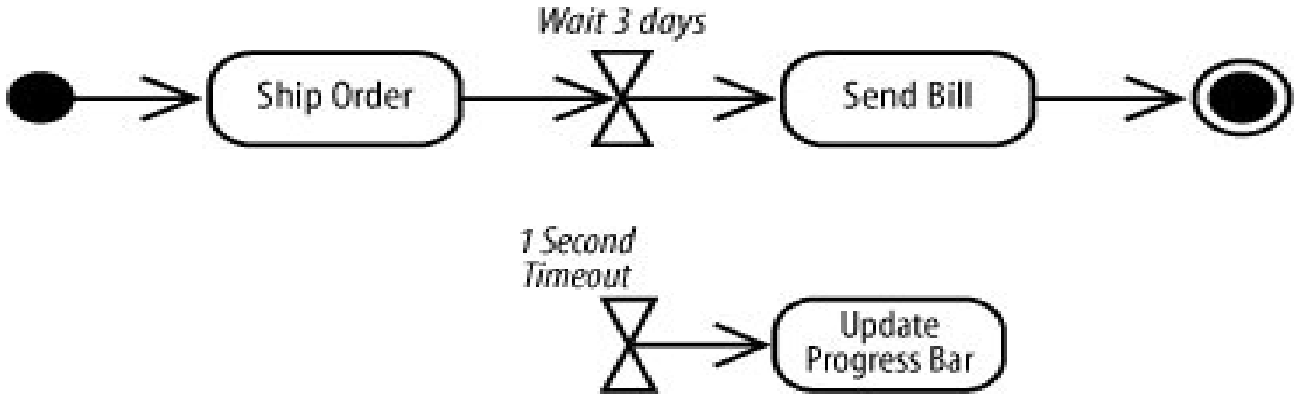
Interruptible Activity Region

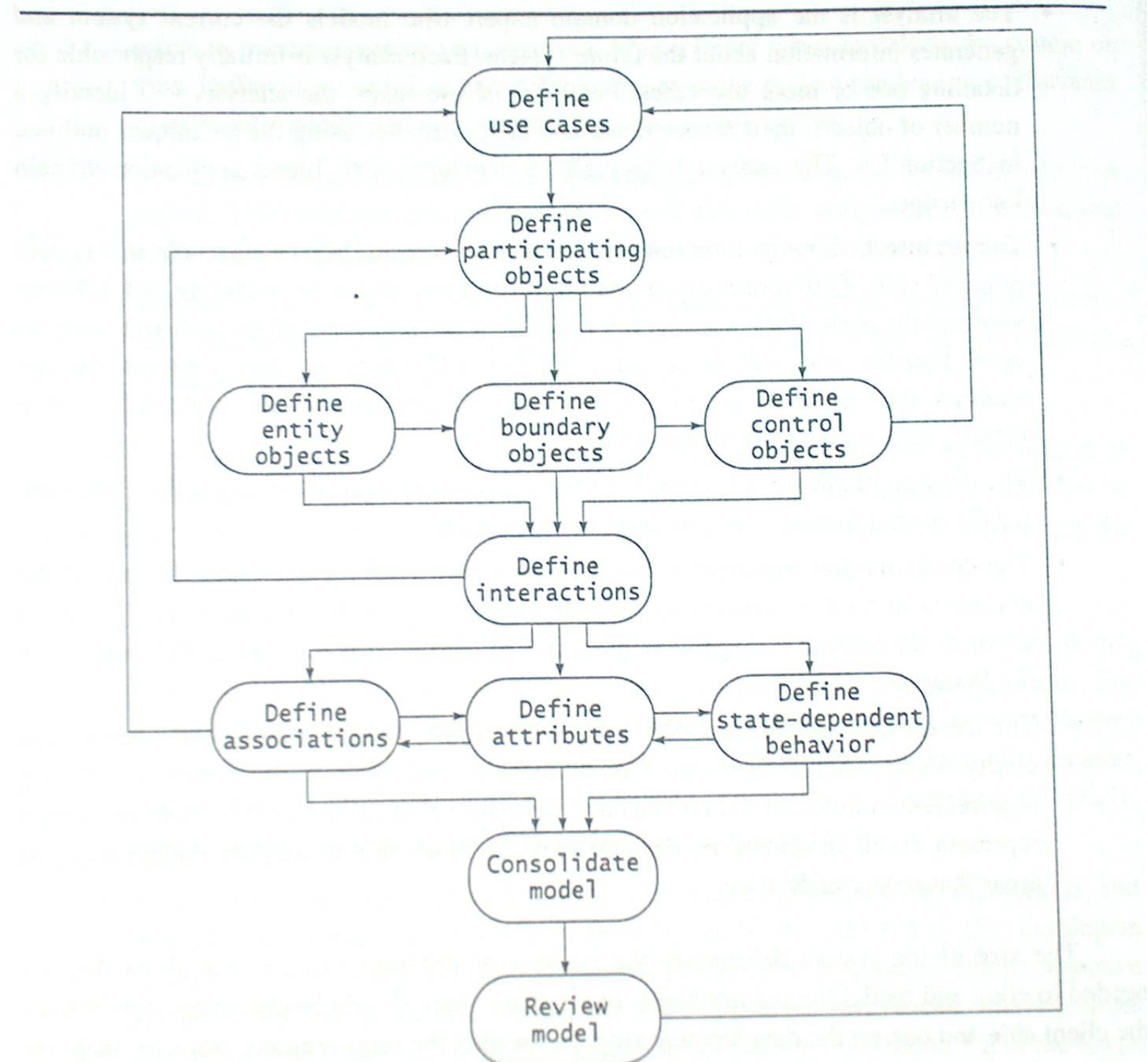
- An interruptible activity region surrounds a group of actions that can be interrupted.
- E.g., the "Process Order" action will execute until completion, when it will pass control to the "Close Order" action, unless a "Cancel Request" interrupt is received, which will pass control to the "Cancel Order" action.



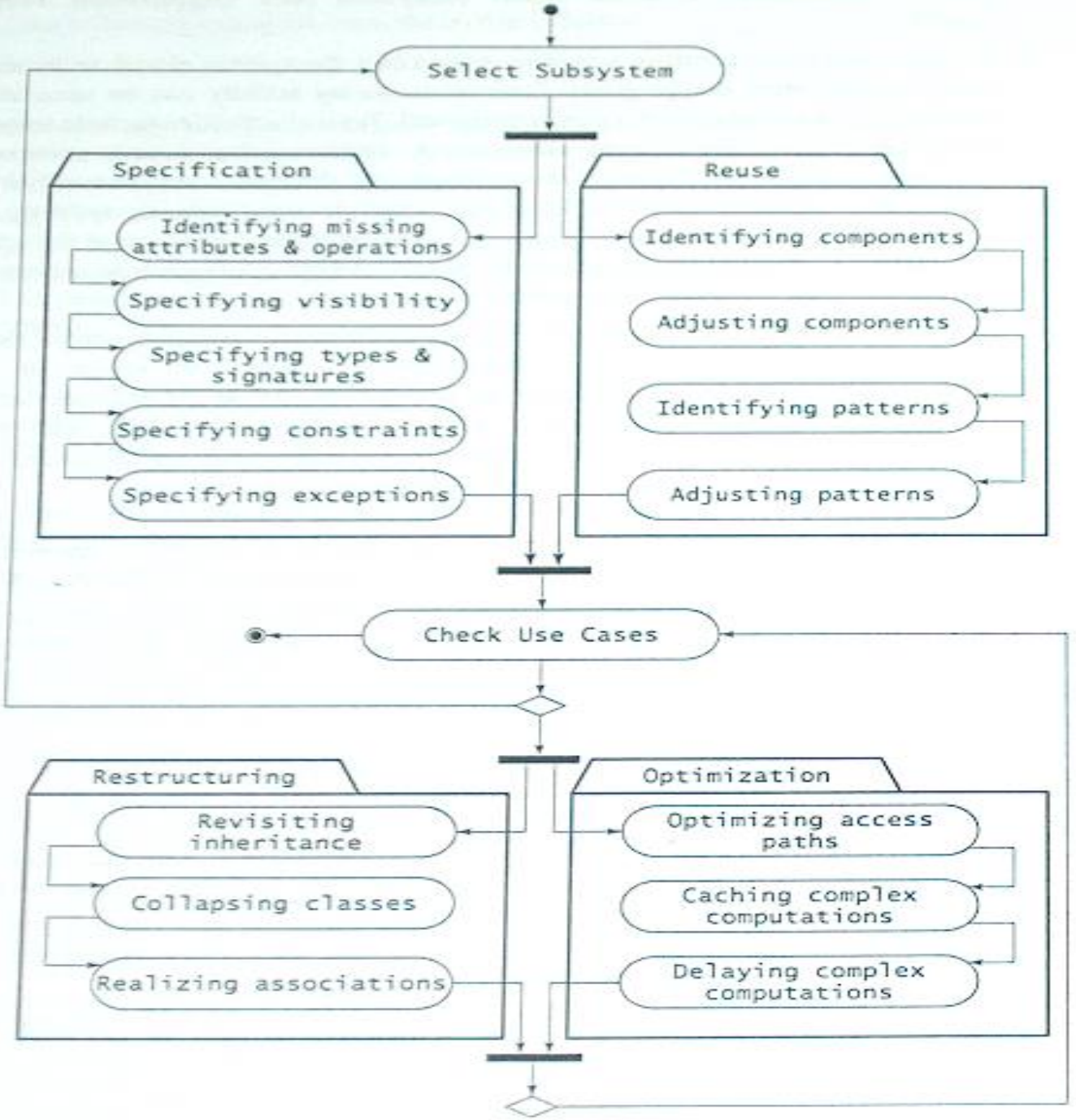
Time Events

- A time event with no incoming flows models a repeating time event





Analyses activities [Bruegge & Dutoit]



Design activities [Bruegge & Dutoit]

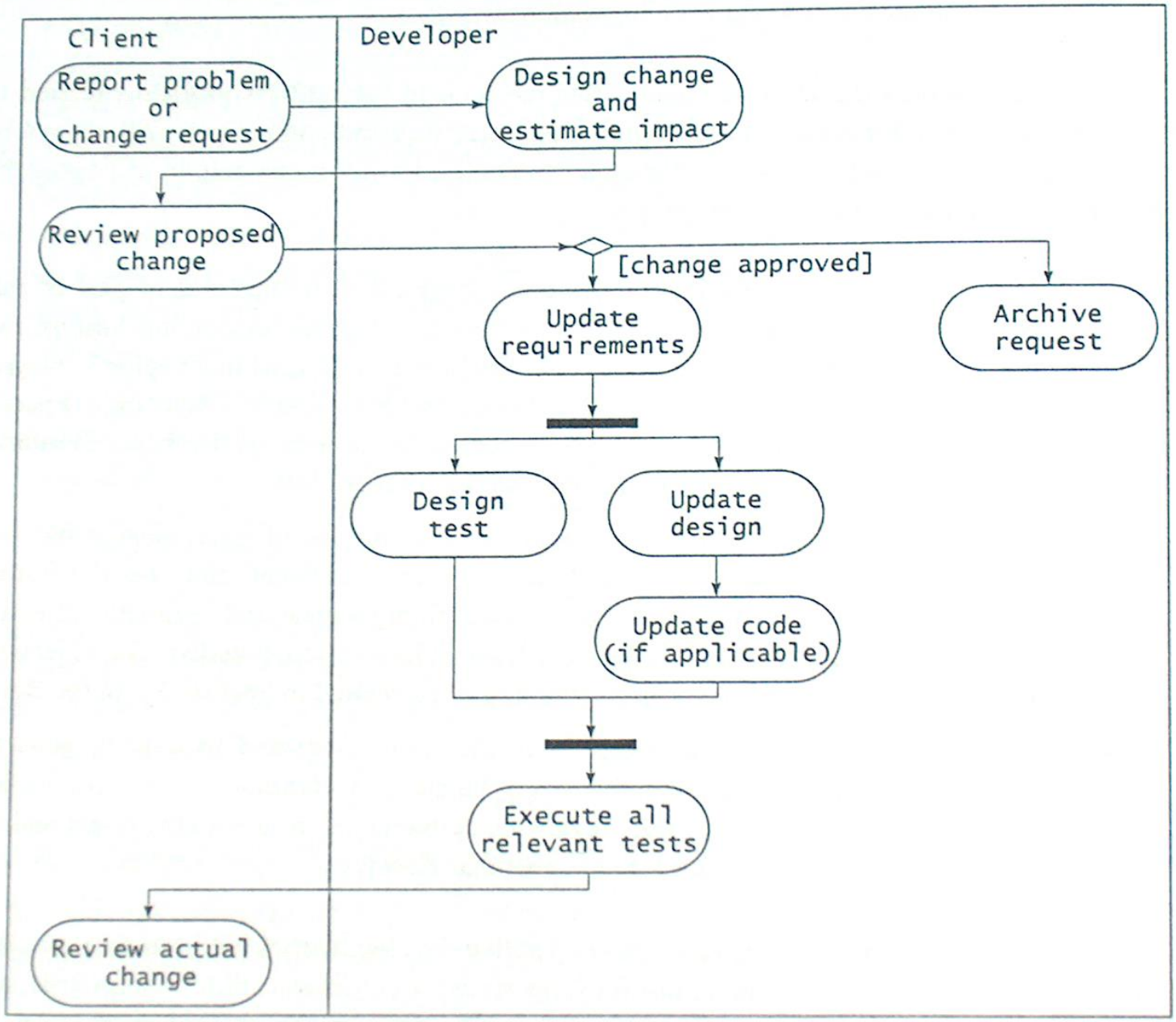


Figure 5-22 An example of a revision process (UML activity diagram).

Revision activities [Bruegge & Dutoit]