Software Analysis and Design (CS:3004)

Activity Diagram

Course Instructor: Nida Munawar

Email Address: nida.munawar@nu.edu.pk

Reference Book: Applying UML and Patterns (An introduction to Object-Oriented Analysis and Design And Iterative Development)

BY Craig Larman

Third Edition

Objectives:

- Discuss and understand activity diagrams
- Understand the elements of activity diagrams
  - Activity
  - Transition
  - Synch. Bar
  - Decision Diamond
  - Start & Stop Markers
- What is an activity diagram?
- Example: Student Enrollment in IIT (SEIIT)
- Activity diagram for a use case in SEIIT
- Basic components in an activity diagram and their notations
- Managing the large activity diagram: Swim Lane
- Activity diagram vs. Flow chart

A Quick Recall…

- **Behavioral UML diagrams,** These UML diagrams visualize how the system behaves and interacts with itself and with users, other systems, and other entities.(What must happen in a system)

- **Structural UML diagrams,** as the name would suggest, show how the system is structured, including the classes, objects, packages, components, etc. in the system and the relationships between those elements.(What is contained in a system)

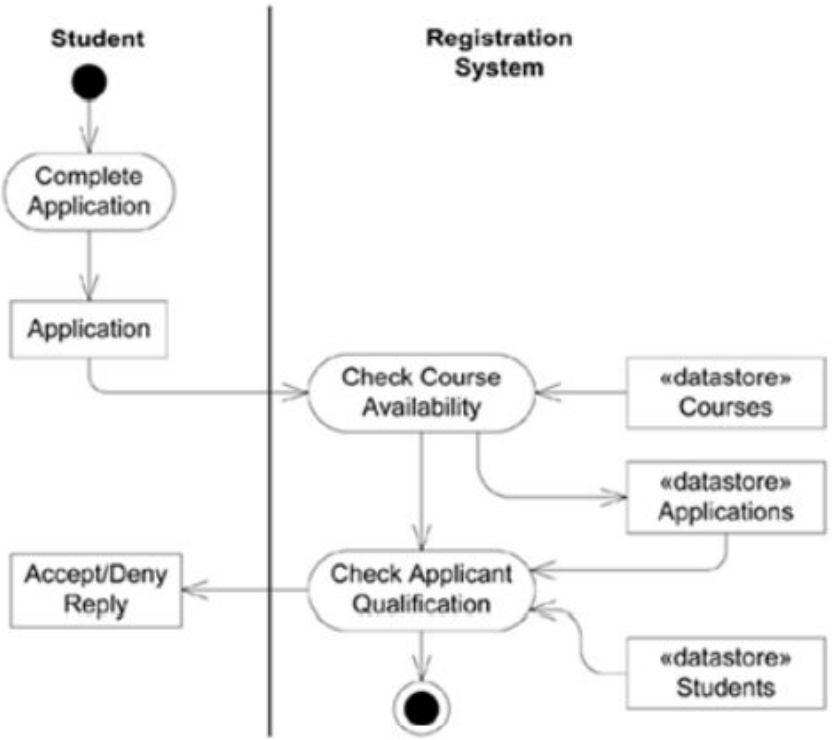
What is an Activity Diagram?

- Represent the dynamic (behavioral) view of a system
- Used for business (transaction) process modeling
- Used to represent flow across use cases or within a use case
- UML activity diagrams are the object oriented equivalent of flow chart and DFDs in function-oriented design approach
- Describes how activities are coordinated.
- Records the dependencies between activities, such as which things can happen in parallel and what must be finished before something else can start.
- Represents the workflow of the process.
- Activity diagram contains activities, transitions between activities, decision points, synchronization

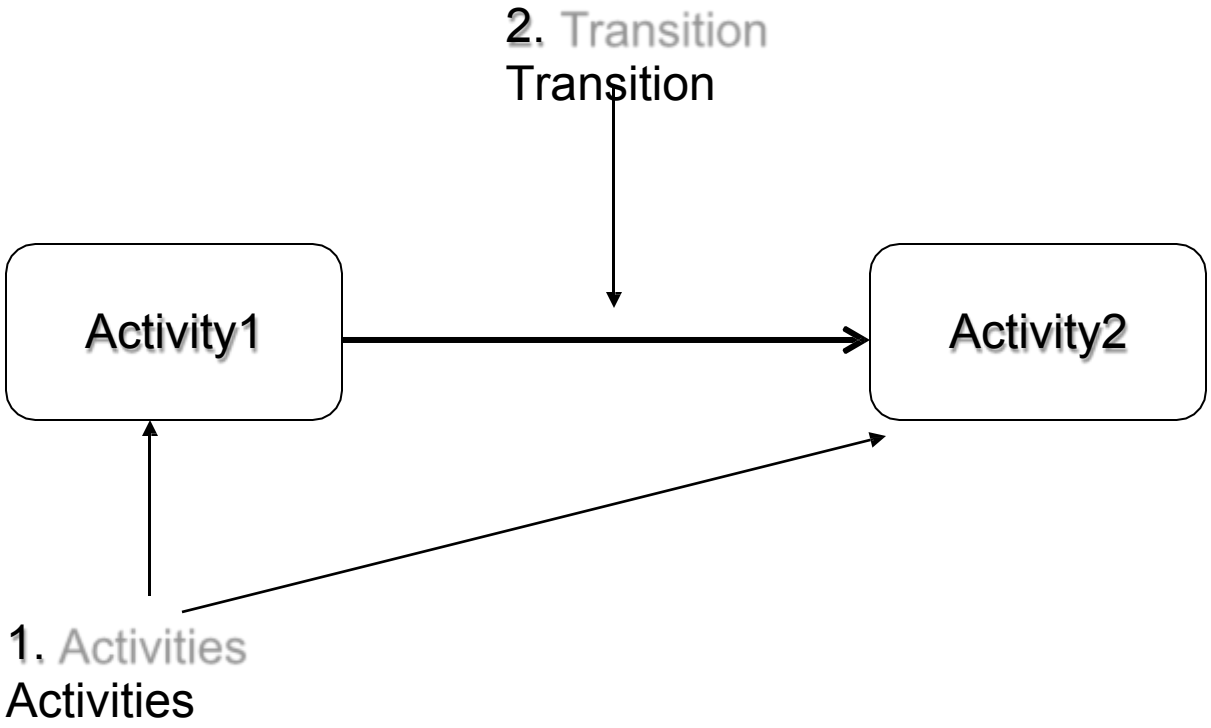
DFD using Activity Diagram

Fortunately, UML activity diagrams can satisfy the same goals they can be used for data flow modeling, replacing traditional DFD notation.

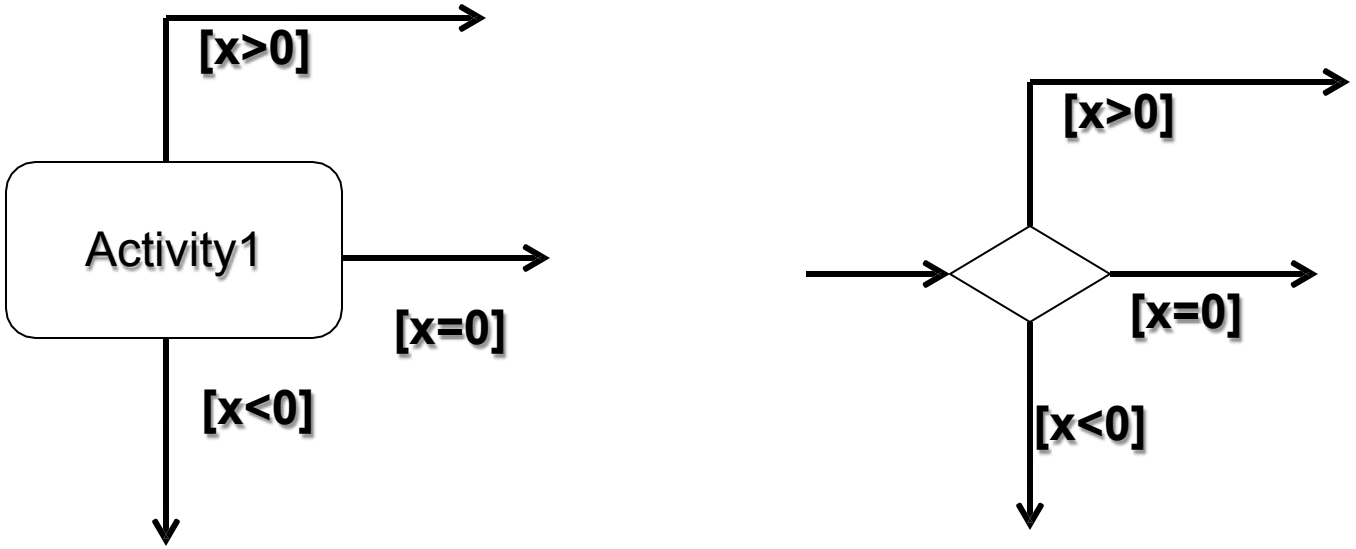
Figure 28.3. Applying activity diagram notation to show a data flow model.



Notation 1 and 2

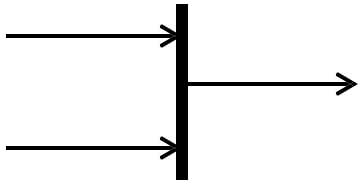


Notation - 3

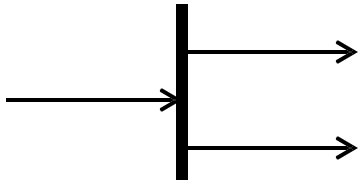


3. Decision Diamond

Notation - 4



4.1 Synch. Bar (Join)

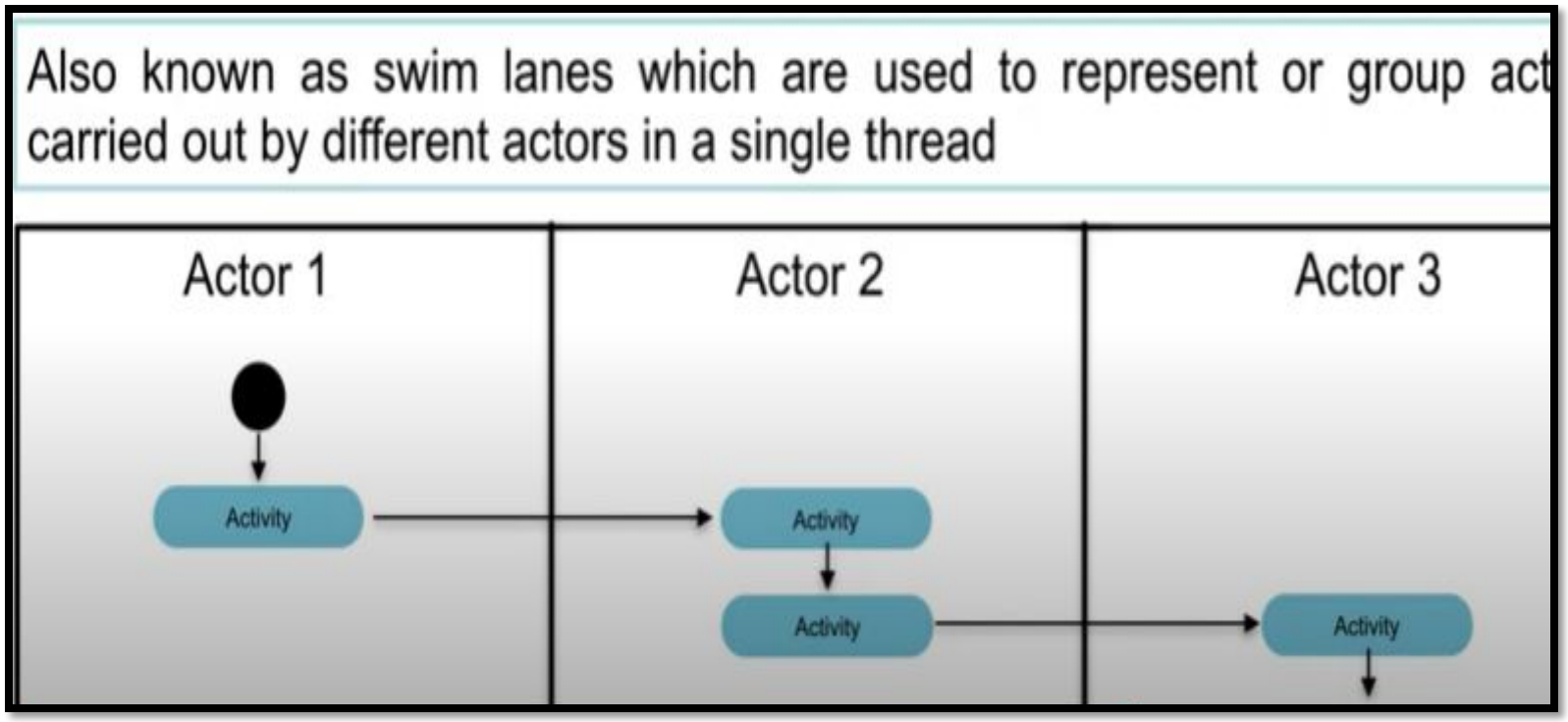


4.2 Splitting Bar (Fork)

Notation - 5

5. Start & Stop Markers

Notation-6



6.Partition

Notations - Guard

[Guard]

Must be true before moving to the next activity

Activity Diagrams (1)

- To model the ***dynamic*** aspects of a system
- It is essentially a ***flowchart***
  - Showing ***flow of control*** from activity to activity
- Purpose
  - Model business workflows
  - Model operations

Activity Diagrams (2)

- Activity diagrams commonly contain
  - Activity states and action states
  - Transitions
  - Objects

Action States and Activity States

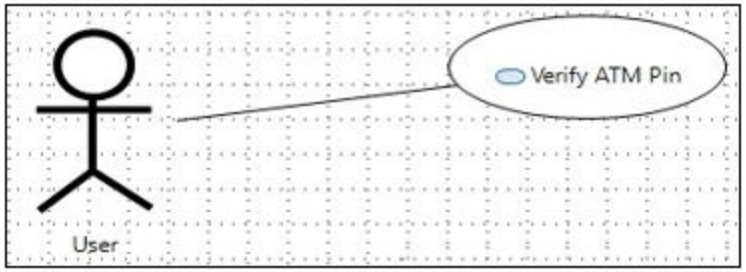
- **Action** states
  - Represents the execution of an atomic action, typically the invocation of an operation.
  - Work of the action state is not interrupted
- **Activity** states can be further decomposed
  - Their activity being represented by other activity diagrams
  - They may be interrupted
- ActionState has is replaced, in UML 2.0, by Action

Transitions (1)

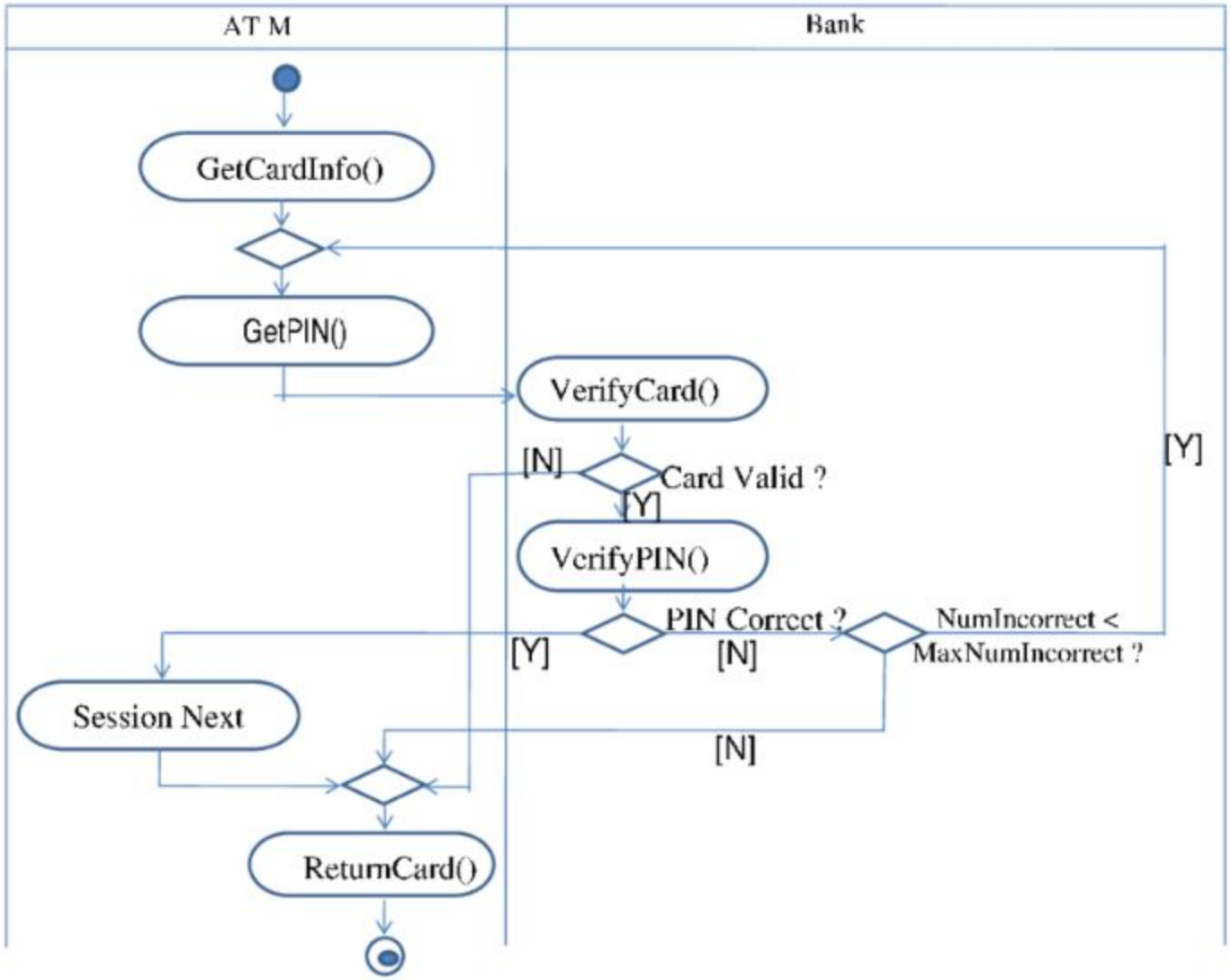
- When the action or activity of a state completes, flow of control passes immediately to the next action or activity state
- A flow of control has to start and end someplace
  - initial state -- a solid ball
  - stop state -- a solid ball inside a circle

convert use case diagram to activity diagram

- A UML activity diagram helps to visualize a certain use case at a more detailed level. It is a behavioral diagram that illustrates the flow of activities through a system.
- **Activity diagram for ATM Verify**



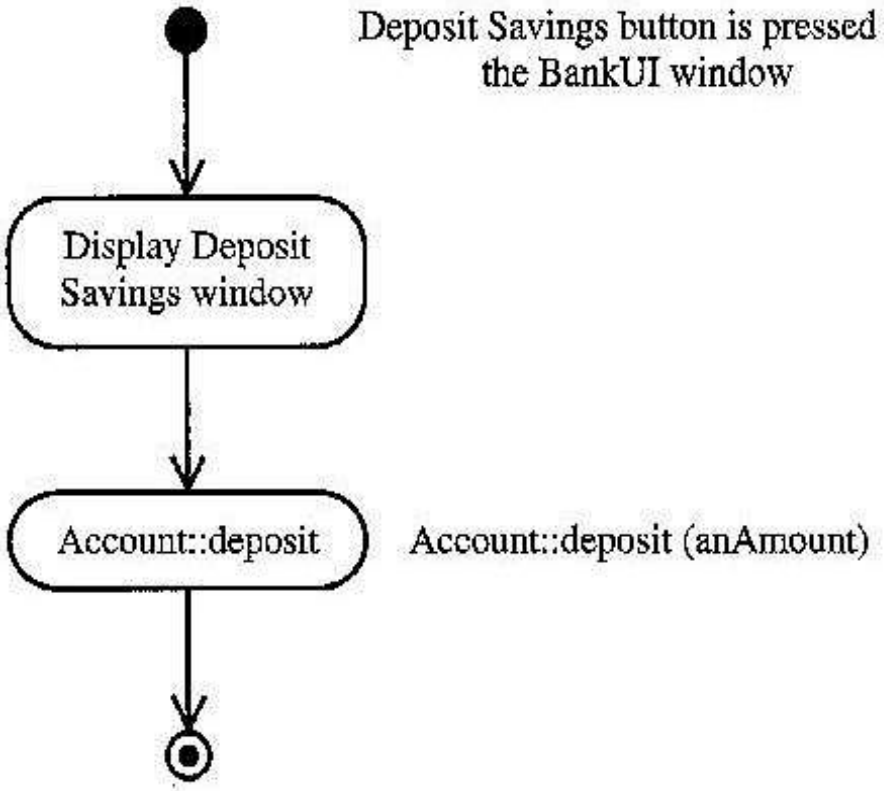
Activity diagram for ATM Verify



Activity Diagram of ATM-Bank login system

Activity Diagram: Example (1)

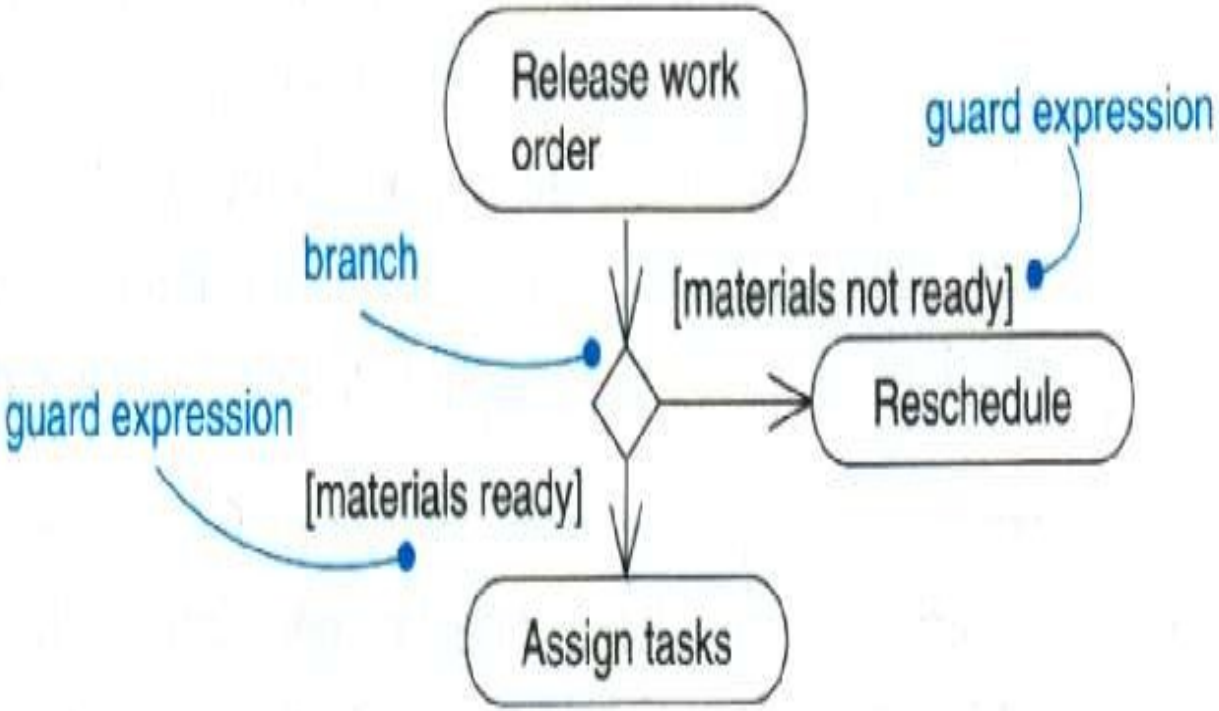
Activity diagram for processing a deposit to a savings account.



Branching (1)

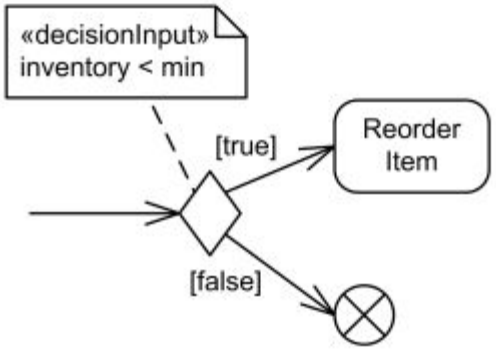
- A branch specifies alternate paths taken based on some Boolean expression
- A branch may have one incoming transition and two or more outgoing ones

Branching (2)


Release work order
guard expression
branch
[materials not ready]
Reschedule
guard expression
[materials ready]
Assign tasks

Decision input behavior

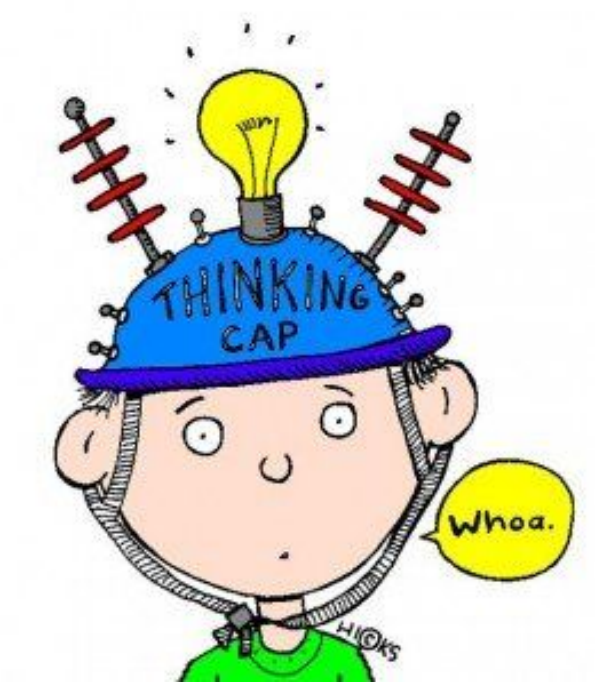
is specified by the keyword **«decisionInput»** and some decision behavior or condition placed in a **note symbol**, and attached to the appropriate decision node.



***Decision node** with **decision input behavior**.*

Class Activity:

Identify the activities performed by an ATM user in an ATM system.



Activity Diagram: Example (2)

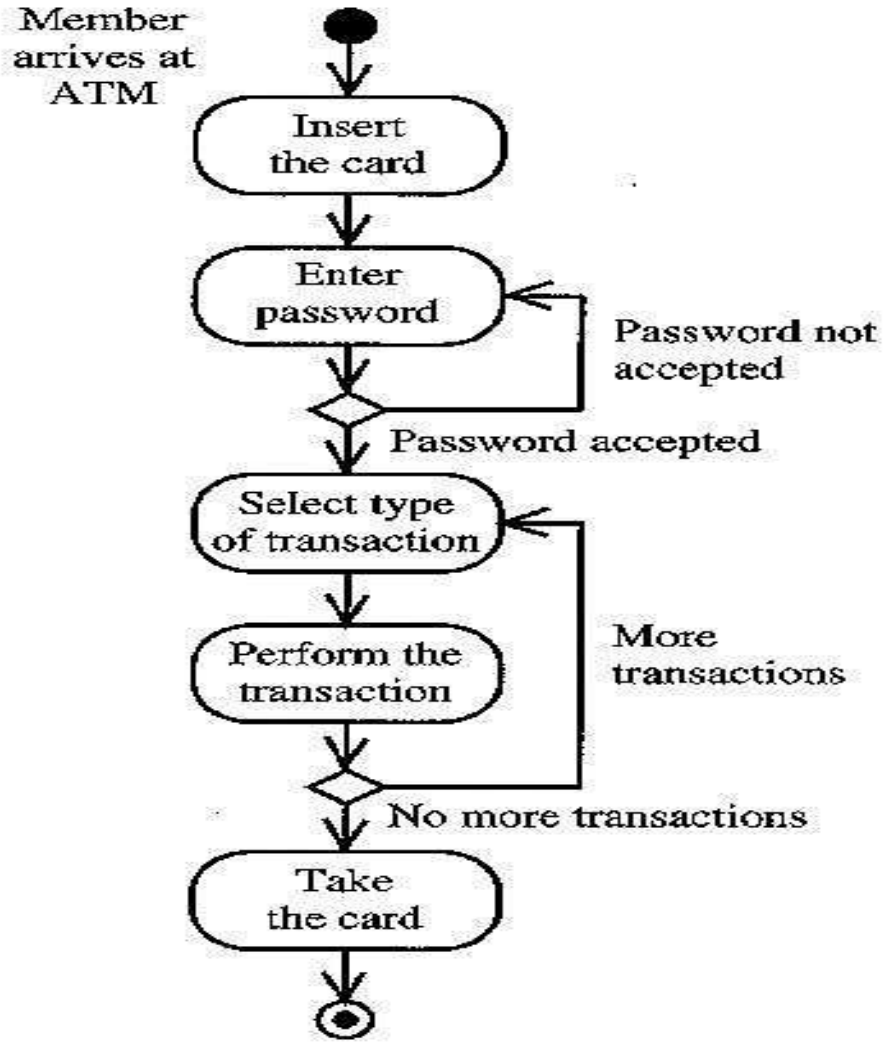
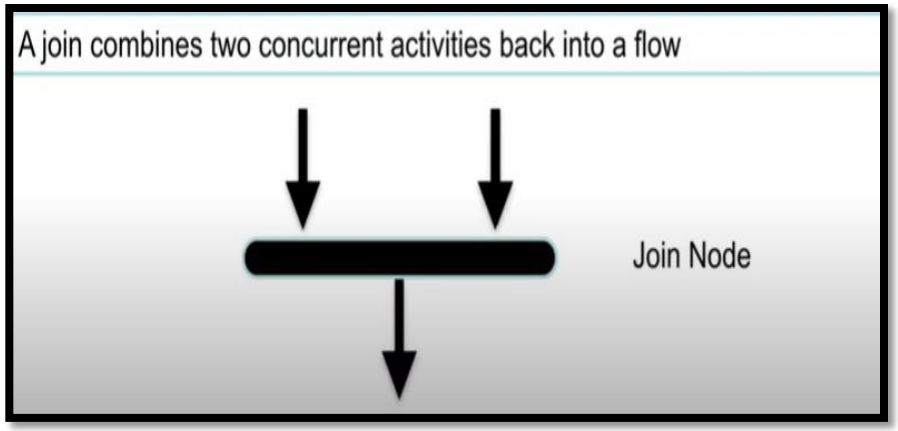
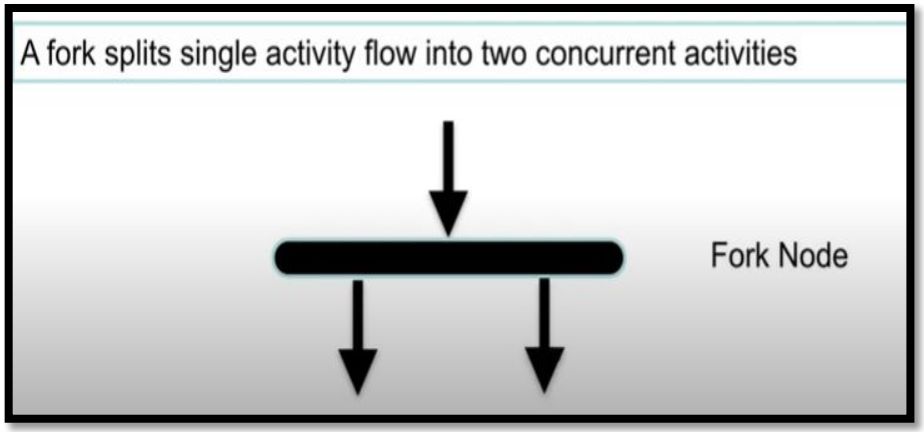


FIGURE 6–8
Activities involved in an ATM transaction.

Forking and Joining

- Use a synchronization bar to specify the forking and joining of parallel flows of control
- A synchronization bar is rendered as a thick horizontal or vertical line
- . The actions are executed concurrently, although they can also be executed one by one. The important thing is that all the parallel edges be performed before they unite





For
k

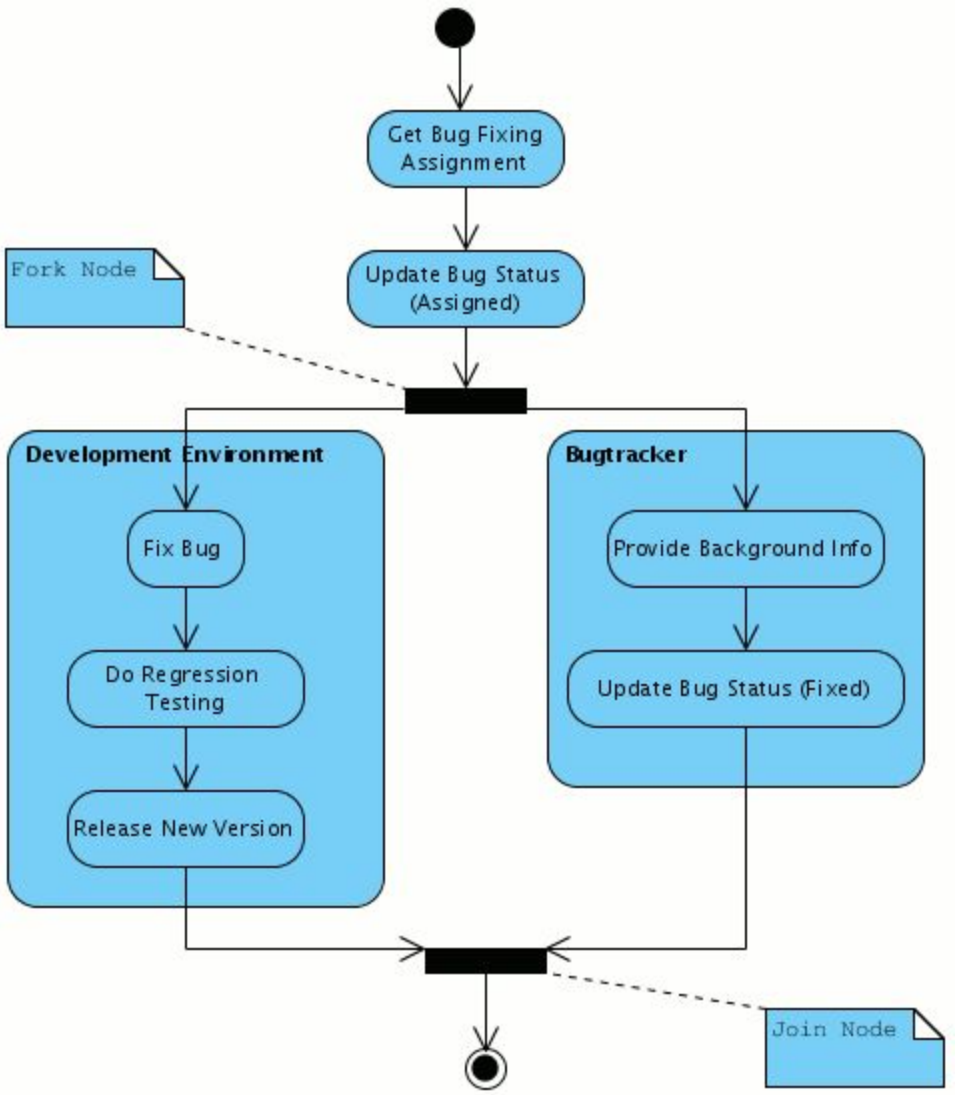
- A fork may have one incoming transitions and

  two or more outgoing transitions
  - each transition represents an independent flow of control
  - conceptually, the activities of each of outgoing transitions are concurrent
    - either truly concurrent (multiple nodes)
    - or sequential yet interleaved (one node)

Join

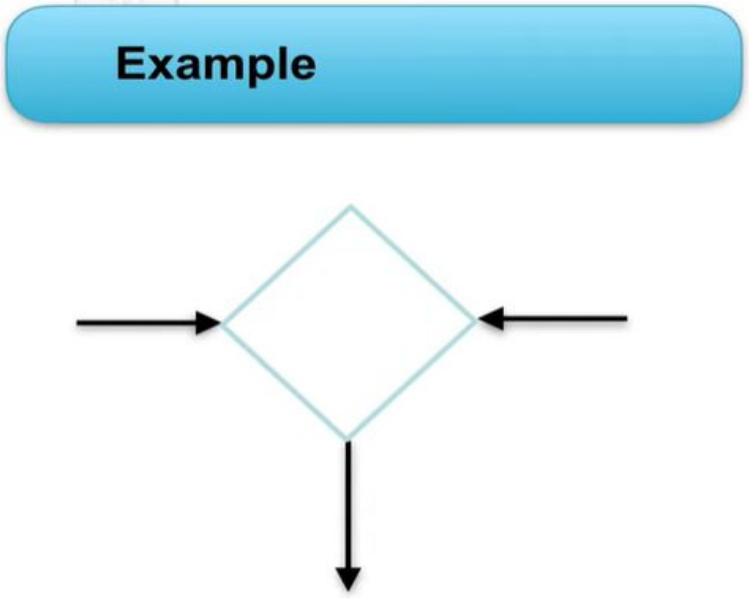
- A join may have two or more incoming transitions and one outgoing transition
  - above the join, the activities associated with each of these paths continues in parallel
  - at the join, the concurrent flows synchronize
    - each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join

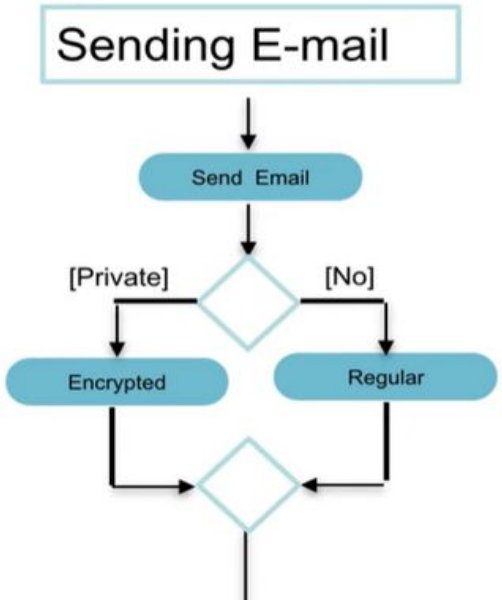
Parallel Activities: Forking and Joining



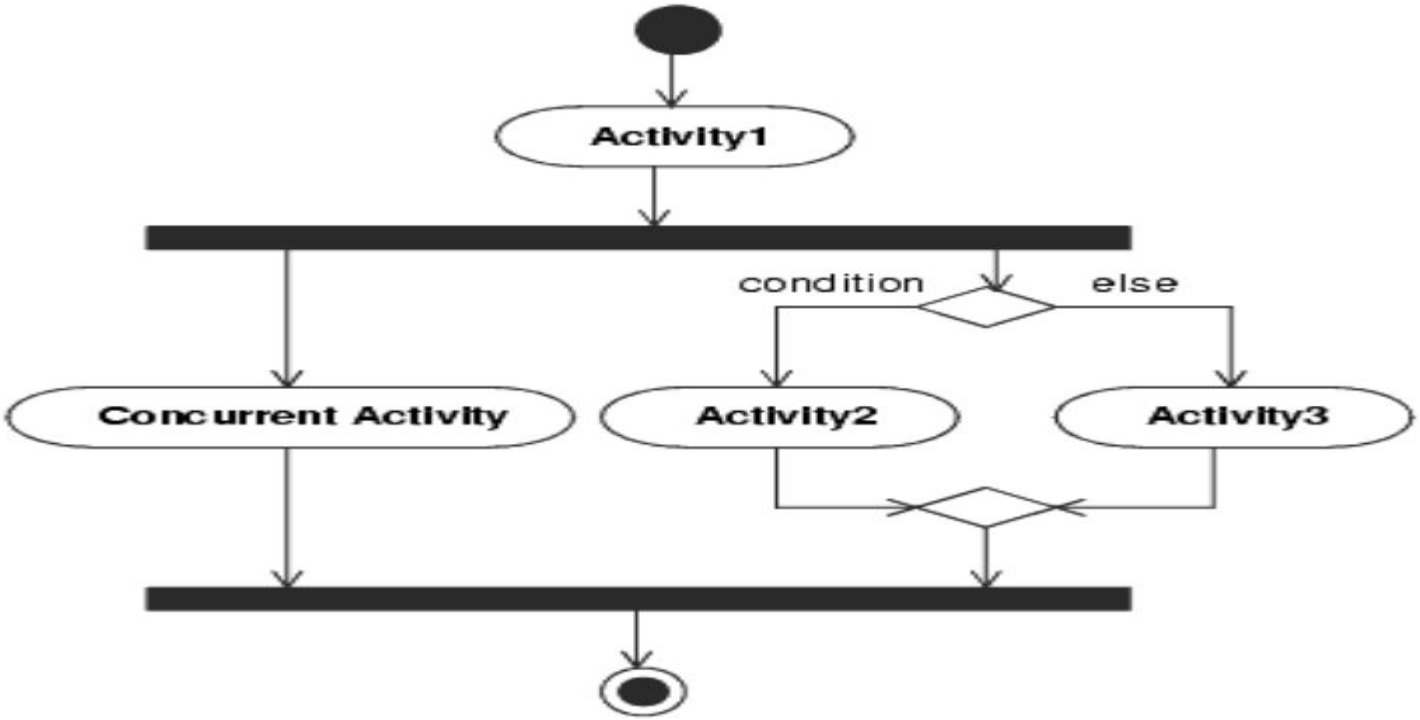
Merge

Two activities are merged with condition and only one activity flows forward
Rejoins mutually exclusive execution path(activities that can not happen concurrently)
Merge node is a control node that brings together multiple incoming **alternate flows** to accept single outgoing flow. There is no joining of tokens. Merge **should not** be used to synchronize **concurrent flows**.



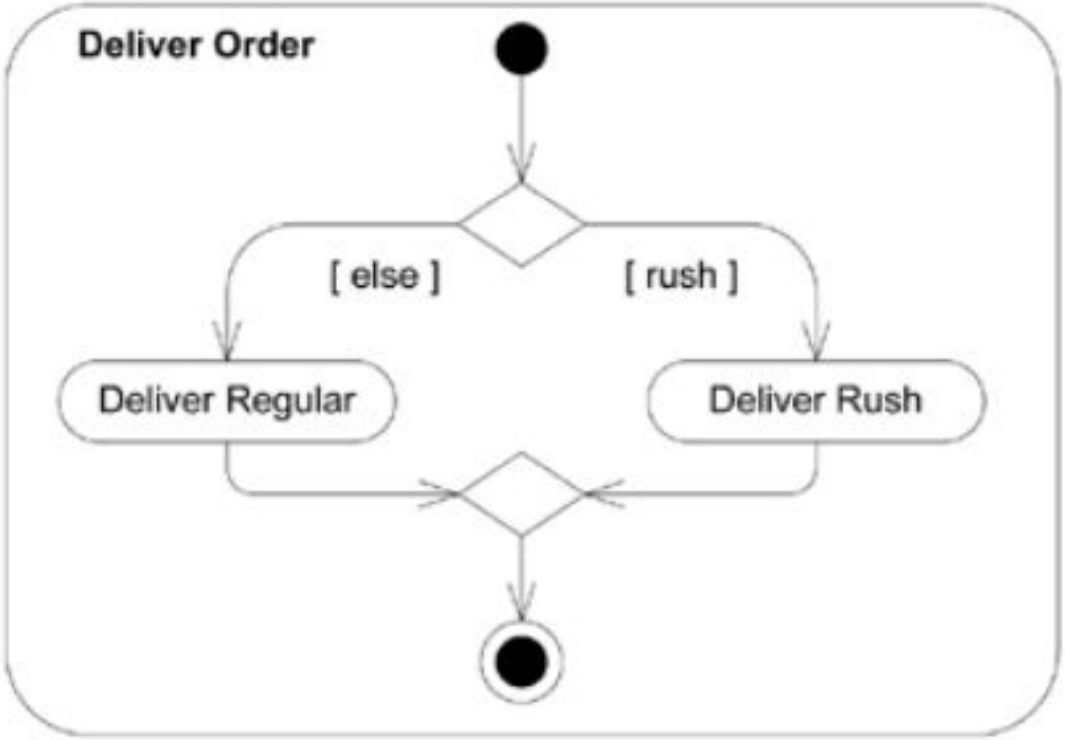


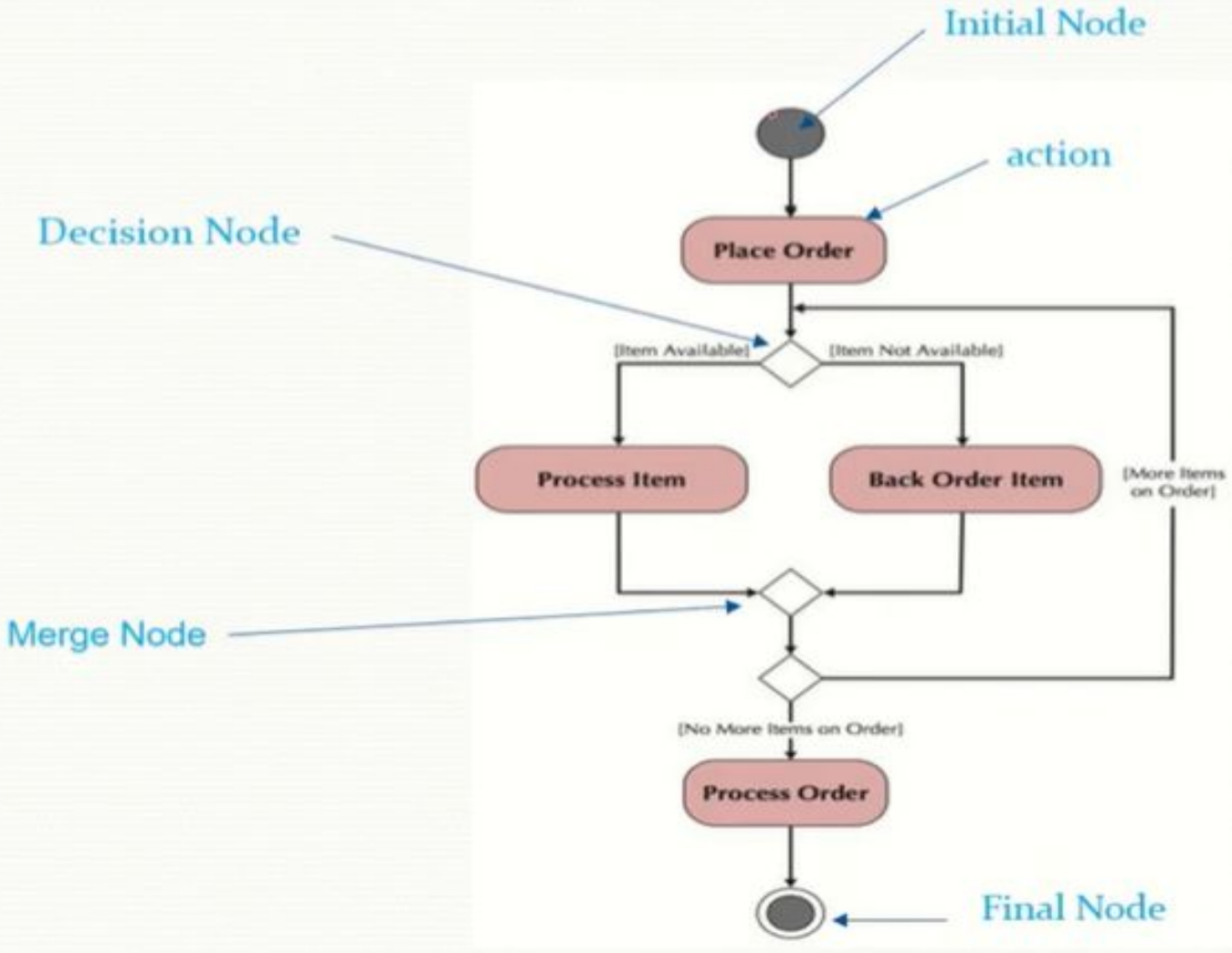
- For example, if a decision is used after a fork, the two flows coming out of the decision need to be merged into one before going to a join; otherwise, the join will wait for **both** flows, only one of which will arrive.

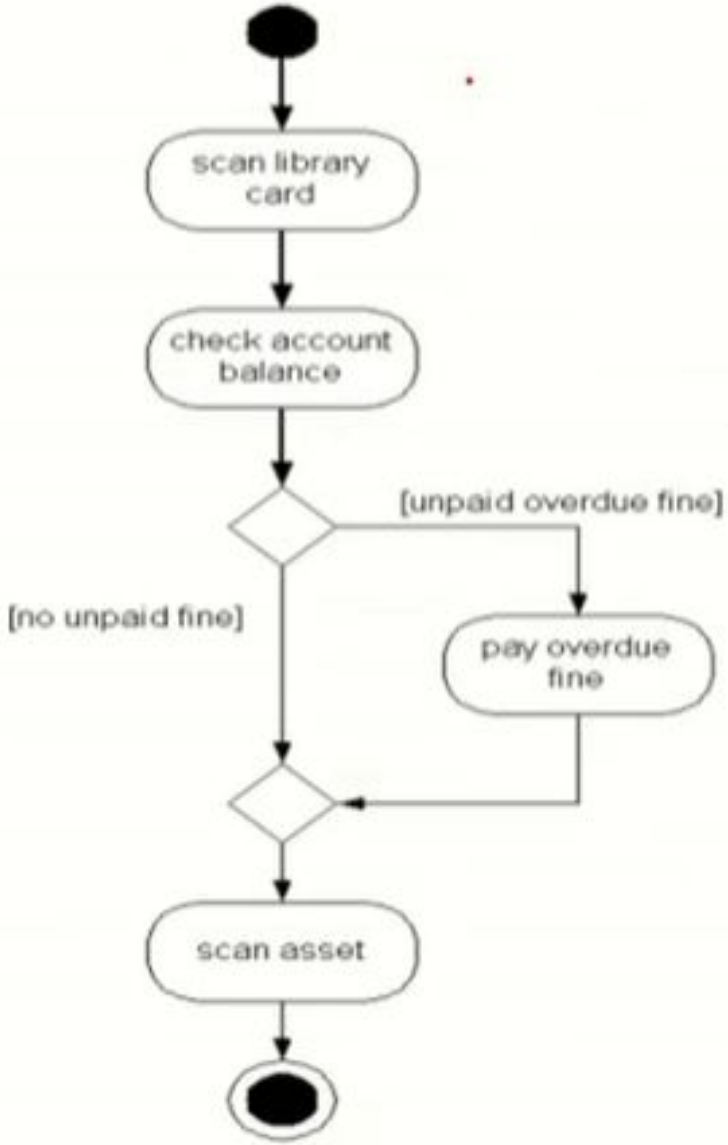


Decision: Any branch happens. Mutual exclusion

Merge: *Any* input leads to continuation. This is in contrast to a *join*, in which case *all* the inputs have to arrive before it continues.




Initial Node
action
Decision Node
Place Order
[Item Available]
[Item Not Available]
Process Item
Back Order Item
[More Items on Order]
Merge Node
[No More Items on Order]
Process Order
Final Node


scan library card
check account balance
[unpaid overdue fine]
[no unpaid fine]
pay overdue fine
scan asset

How to show that an activity is expanded in another activity diagram?

- using the rake symbol.

Figure 28.4. An activity will be expanded in another diagram.

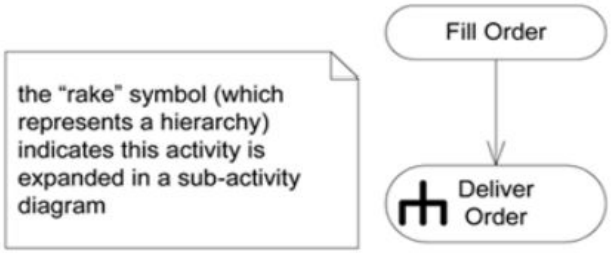
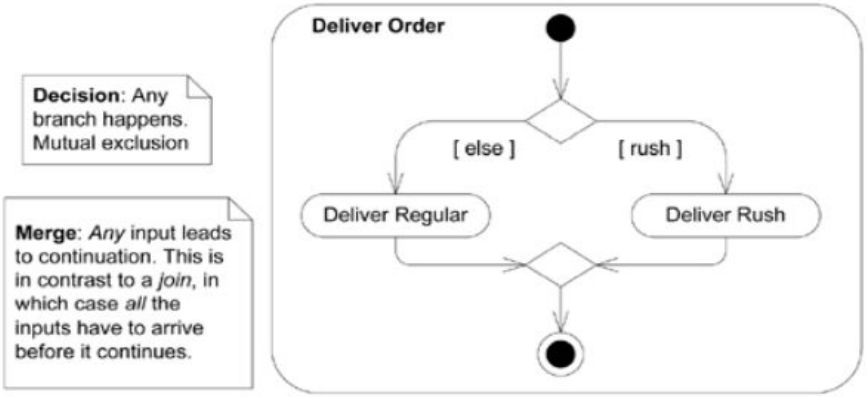


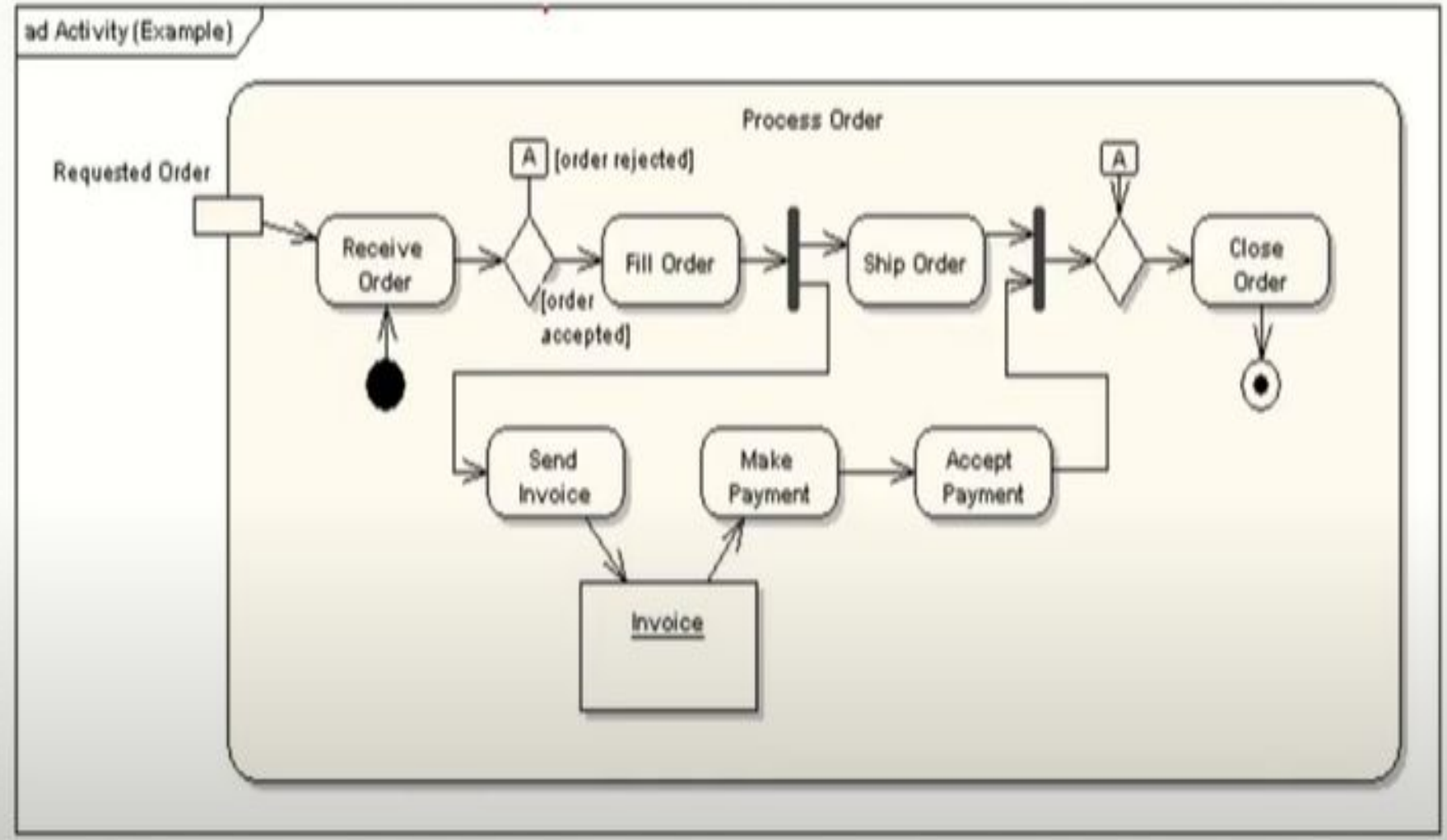
Figure 28.5. The expansion of an activity.



Basic Components in an Activity Diagram

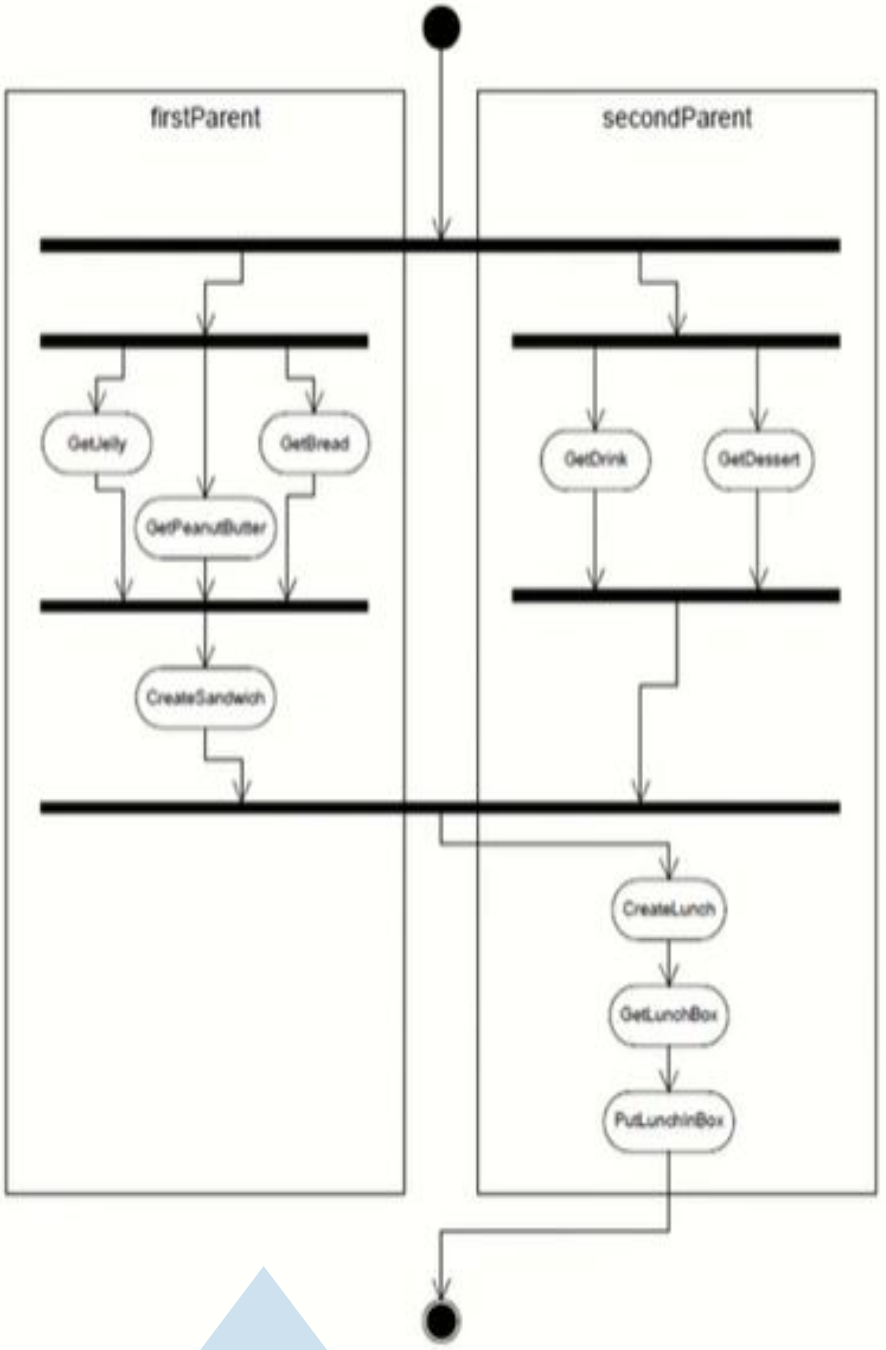
- Difference between Join and Merge
  - A join is different from a merge in that the join synchronizes two inflows and produces a single outflow. The outflow from a join cannot execute until all inflows have been received
  - A merge passes any control flows straight through it. When all incoming flow reach this point then processing continues

Example of activity Diagram

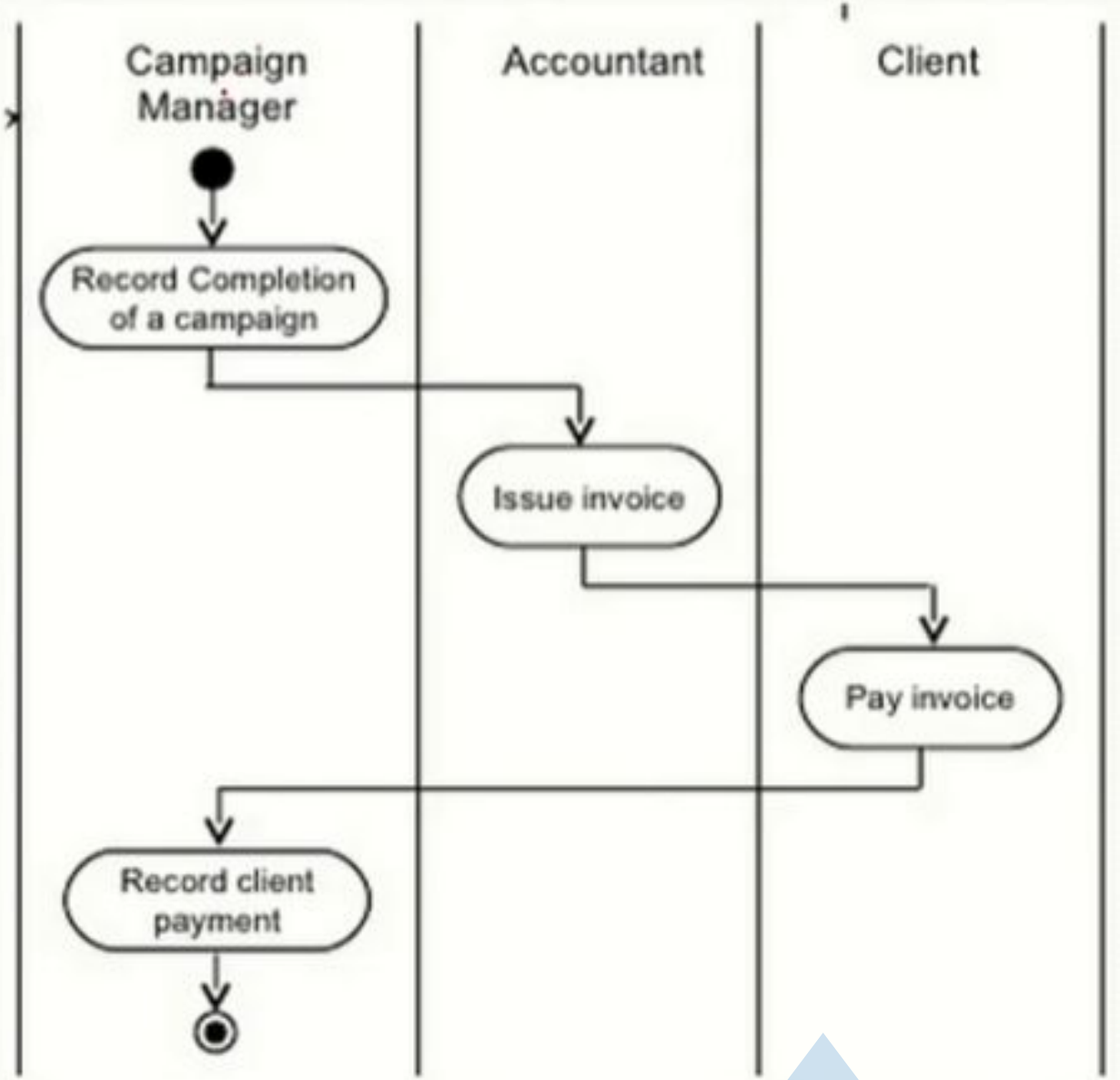


Swimlanes

- Used to assign responsibility to objects or individuals who actually perform the activity
- Represents a separation of roles among objects
- Can be drawn horizontally or vertically



Example of Swimlanes



Swimlanes

- A swimlane delineates who does what in a process.
- To partition the activity states on an activity diagram into groups
  - each group representing the business organization responsible for those activities
  - each group is called a swimlane
- It provides clarity and accountability by placing process steps within the horizontal or vertical “swimlanes” of a particular employee, work group or department.

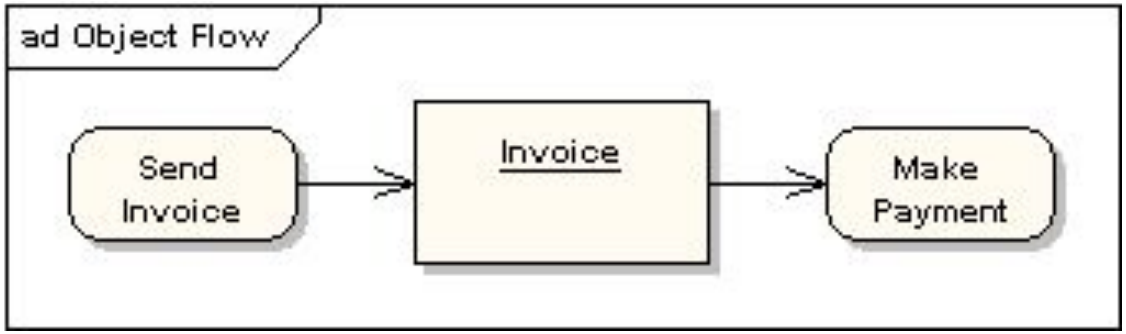
Swimlanes

- Each swimlane has a name unique within its diagram
- Each swimlane may represent some real-world entity
- Each swimlane may be implemented by one or more classes
- Every activity belongs to exactly one swimlane, but transitions may cross lanes

Some more features in Activity Diagrams

Object and Object Flow

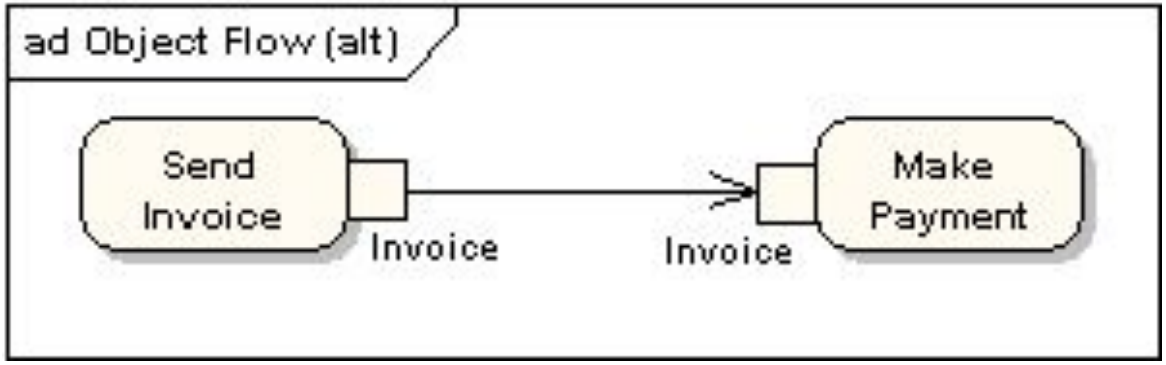
- There is no data flow in activity diagram. Object flow plays role of data flow as well.
- An object flow is a path along which objects can pass. An object is shown as a rectangle
- An object flow is shown as a connector with an arrowhead denoting the direction the object is being passed.



Input and Output Pin

- An object flow must have an object on at least one of its ends.

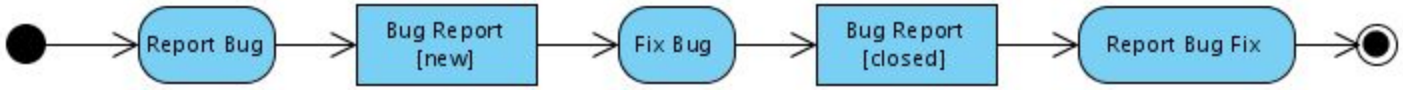
  A shorthand notation for the above diagram would be to use input and output pins



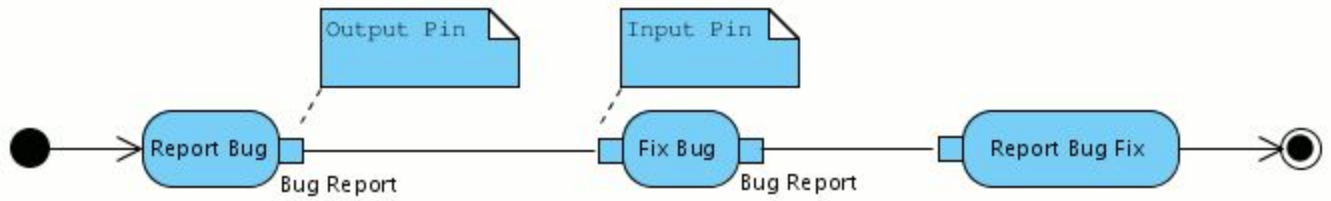
Passing Objects Between Actions

- Objects in the UML language hold information - state - that is passed between actions. Objects may represent class instances:

- Objects usually change state between actions:

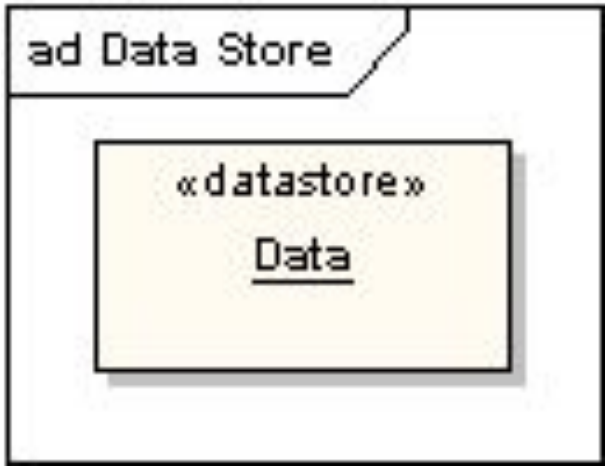


- An alternative notation shows action ***input pins*** and ***output pins***. These emphasize that the corresponding object is *required* while ***object nodes*** rather emphasize the *existance* of that object:



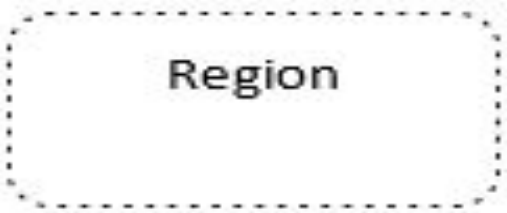
Data Store

- A data store is shown as an object with the «datastore» keyword



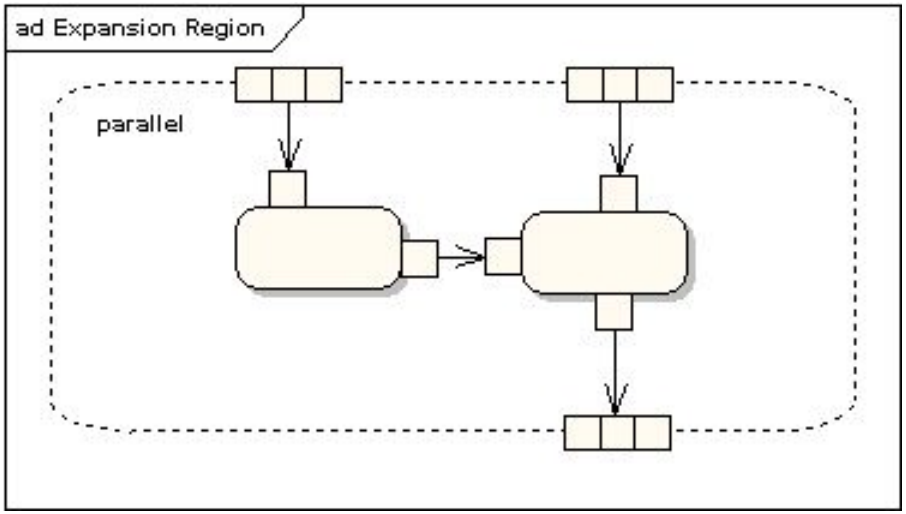
Expansion Region:

- On an Activity diagram, sometimes the output of an action can trigger multiple invocations of another action when this happens it's helpful to use an expansion region in your activity diagram.
- An expansion region shows a set of actions that occur once for each item in a collection so an expansion region contains some process that acts multiple times on the incoming data once for each element in the input collection it may be helpful to think of an expansion region as behaving like a for loop over the input collection.



Expansion Region

- Expansion regions and structured activities provide mechanisms to show the strategy for managing the synchronization issues in a real- time system.
- An expansion region is a structured activity region that executes multiple times for collection items.
- Input and output expansion nodes are drawn as a group of three / four boxes representing a multiple selection of items. The keyword iterative, parallel or stream is shown in the top left corner of the region



Interaction Types:

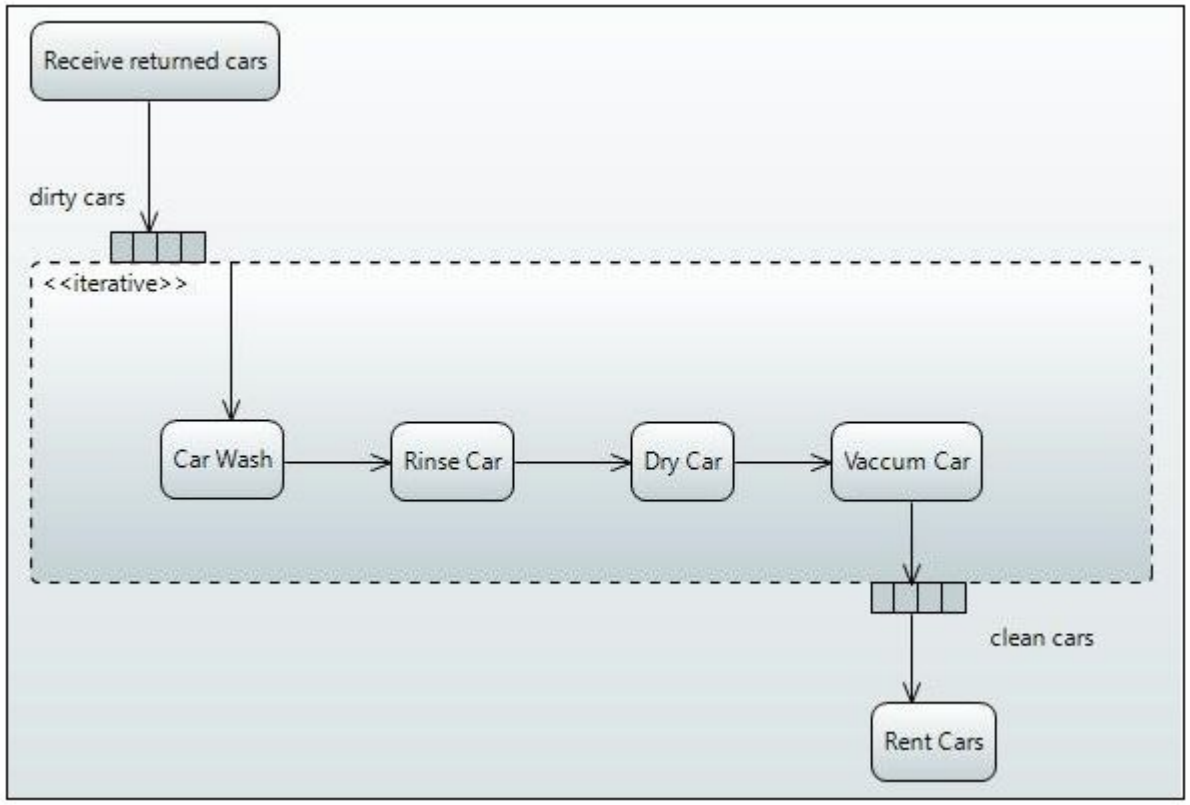
- Iterative: when your expansion region is iterative the executions happens in sequence (One item at a time)

- Parallel: when your expansion region is parallel it means your interactions are independent and can happen concurrently

- Streams: All the input elements enters the expansion region at the same time so they come in all at once and the expansion region can operate on the collections as the whole

Iterative:

- **Example:**
- Imagine a car rental agency and when cars are returned to this agency they collect a batch of cars and then they send those cars through their one lane car wash one at a time so our action that would trigger this multiple invocation of the carwash would be receive returned cars and once they've received batch of cars they're going to send them one at a time through the carwash and so that will be our expansion region

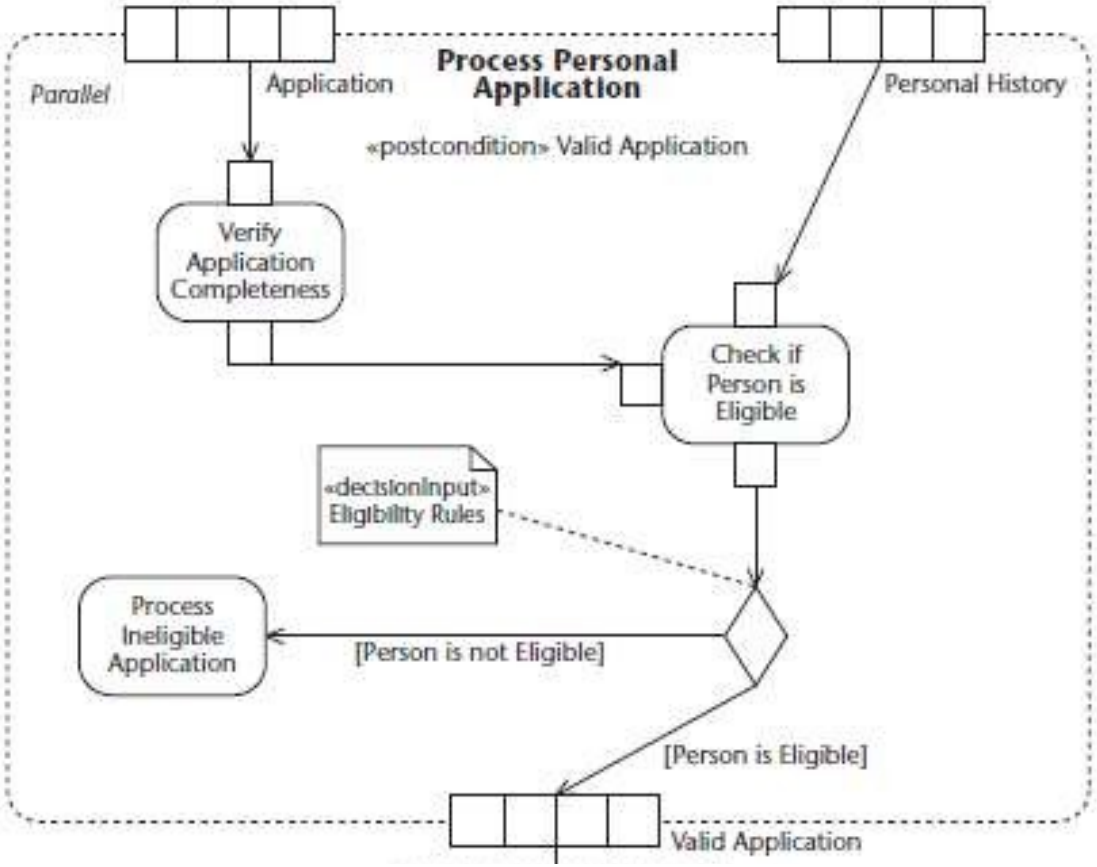

Receive returned cars
dirty cars
<<iterative>>
Car Wash
Rinse Car
Dry Car
Vaccum Car
clean cars
Rent Cars

Expansion Region:

Application-processing:

The organization must accept an electronic application, review the eligibility of the application, store the application, and then confirm receipt of a valid application.

Parallel


Parallel
Application
Process Personal Application
Personal History
«postcondition» Valid Application
Verify Application Completeness
Check if Person is Eligible
«decisionInput» Eligibility Rules
Process Ineligible Application
[Person is not Eligible]
[Person is Eligible]
Valid Application

Continue..

- The top section on the diagram shows an expansion region, a type of structured activity that handles the input and output of collections with multiple executions.
- The collection as two sets of four boxes on the top and one at the bottom showing the application, the personal information, and the verified applications.
- The italicized word in the upper-left corner shows that this expansion
- region allows multiple executions to the actions to occur in parallel.
- The action executions do not have to follow any ordering on the entering

collections.

- You can also use iterative to show the region only allows one set of actions to execute at a time or streaming to show execution in the order of the collection.
- The system also relies on information from a database about the person.
- When combined with the eligibility rules for the application, shown on the diagram as a <<decisionInput>>, the organization filters out ineligible applications and sends the application on for storage.

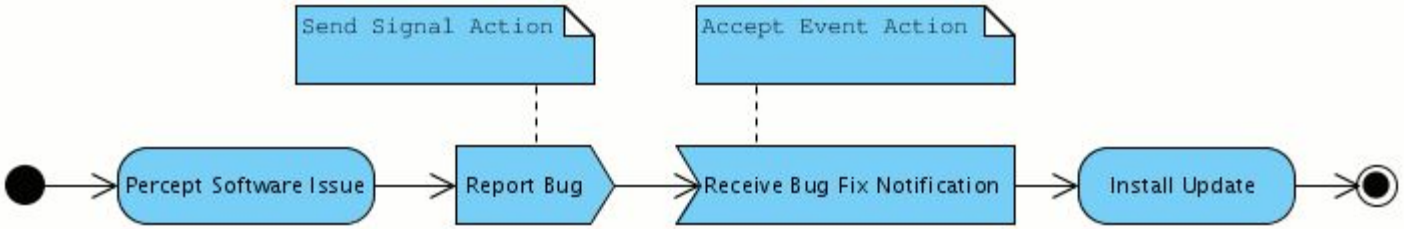
signals

They usually appear in pairs because the state can't change until the response is received
e.g., an authorization of payment is needed before the order is completed
Input/Receive signal : Initiate a particular action
Output/Send signal : waits until the response is received

Concave pentagon = receive signal
Context pentagon= send

Interacting with External Participants

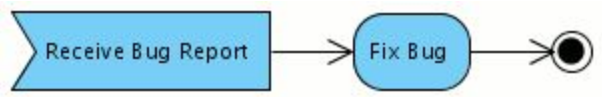
- External participants may be external processes, systems, or people, interacting with an activity.
- A ***receive signal*** "wakes up" an action that is "sleeping".
- A ***send signal*** is sent to an external participant.
- In the following diagram, both a ***send*** and ***receive signal action*** are used.



- Note that the activity flow gets interrupted - gets into a wait state - until the bug fix notification is received. (If there was no ***receive signal action***, however, the flow would just continue after executing the ***send signal action***.)

Interacting with External Participants

- An activity may also start with a ***receive signal node***:



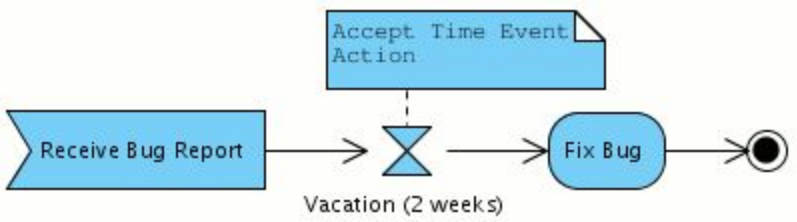

Call pizza place
Doorbell rings
Answer door
Pay
Eat

Time Event

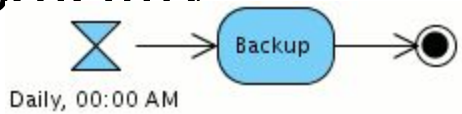
Stops the flow for a period(depicted as an hour class)

Time Events

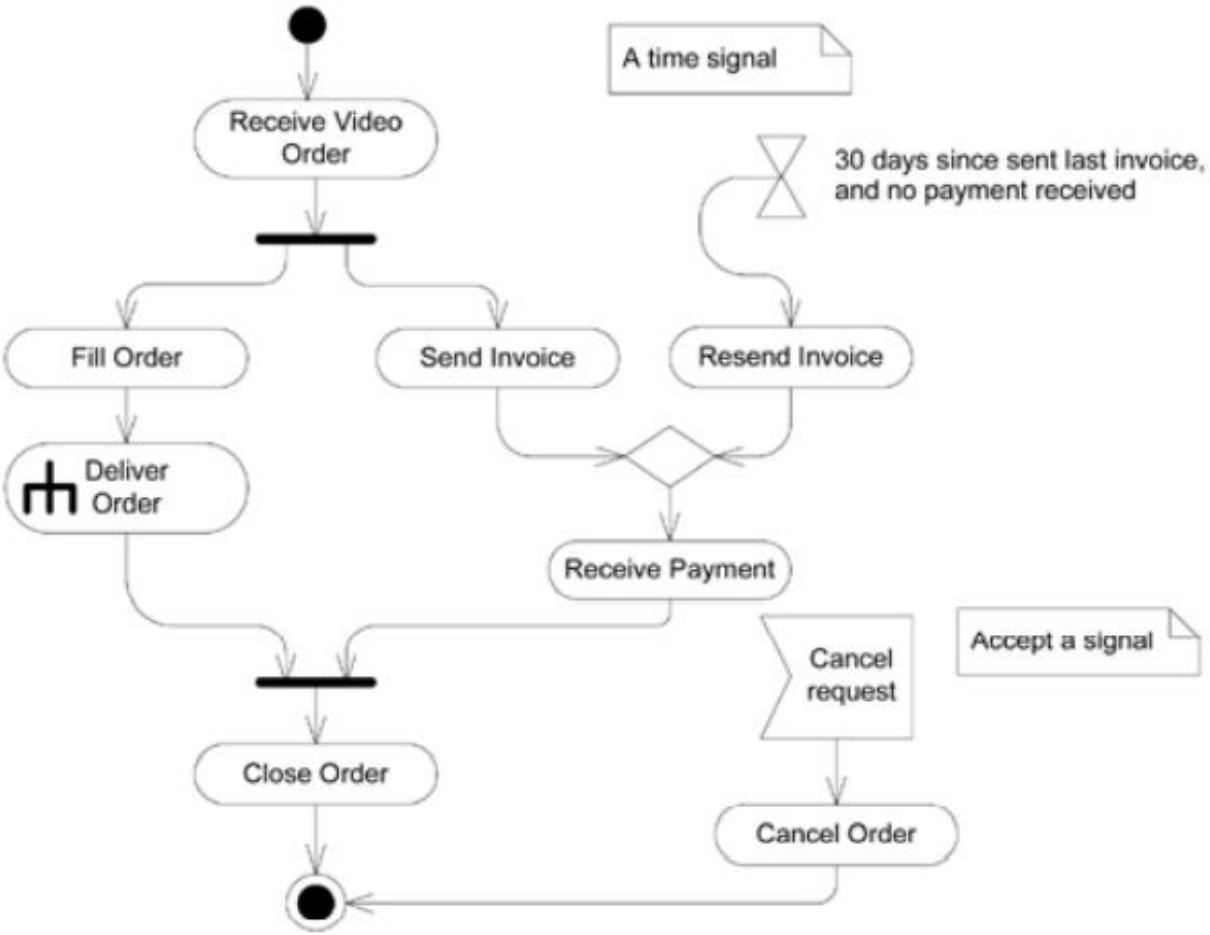
- A ***time event*** models a wait period:



- An activity starting with a time event is launched periodically:

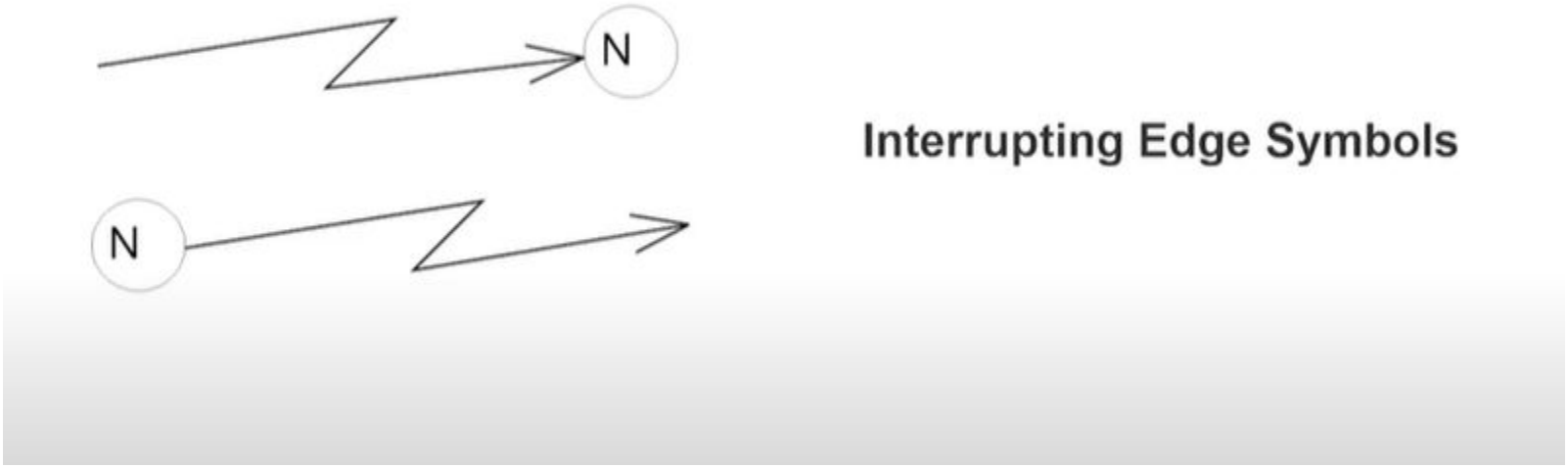


when you need to model events such as time triggering an action, or a cancellation request.



Interrupting an Activity

Interrupts a flow e.g., cancellation
Represented with Zigzag arrow
It looks like exception handler but in this context, we call it an interrupting edge



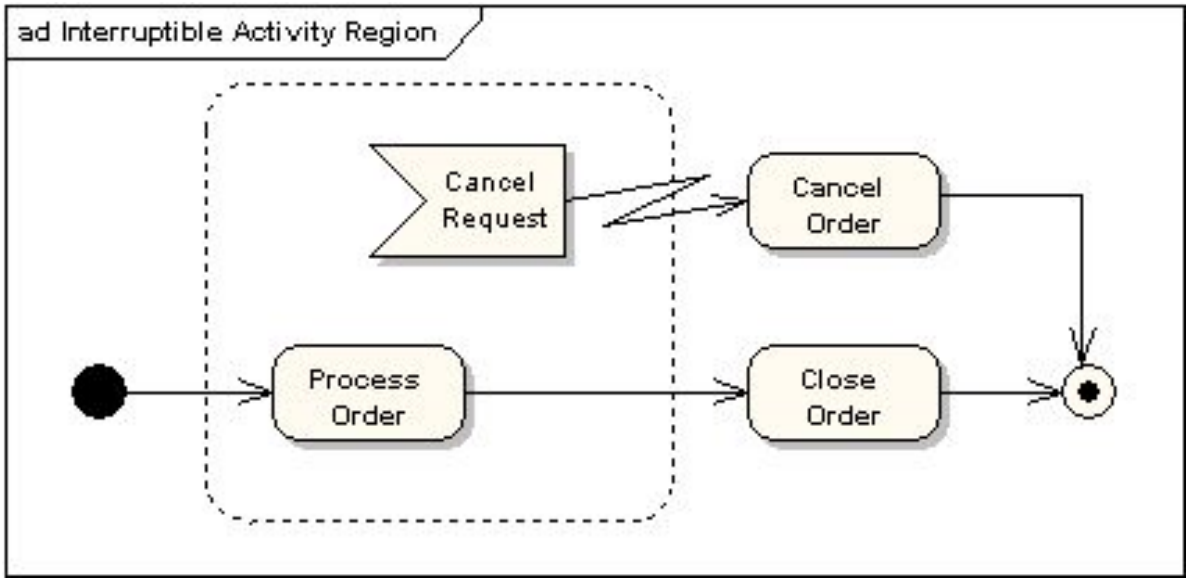
Interruptible Activity Region

- An interruptible activity region is represented by Activity box with the dashed lines
- The actions that are interrupted must be placed inside the interruptible activity region
- The event that could cause the interruption as also placed inside the interruptible activity region
- When the interruption happens, we bypass all the actions and moves towards the end

- As you diagram a processor activity you may find that an action or a group of actions can be interrupted by some event to show this use an interrupted activity region an interruptible activity region is represented by an activity box with a dashed line border.
-

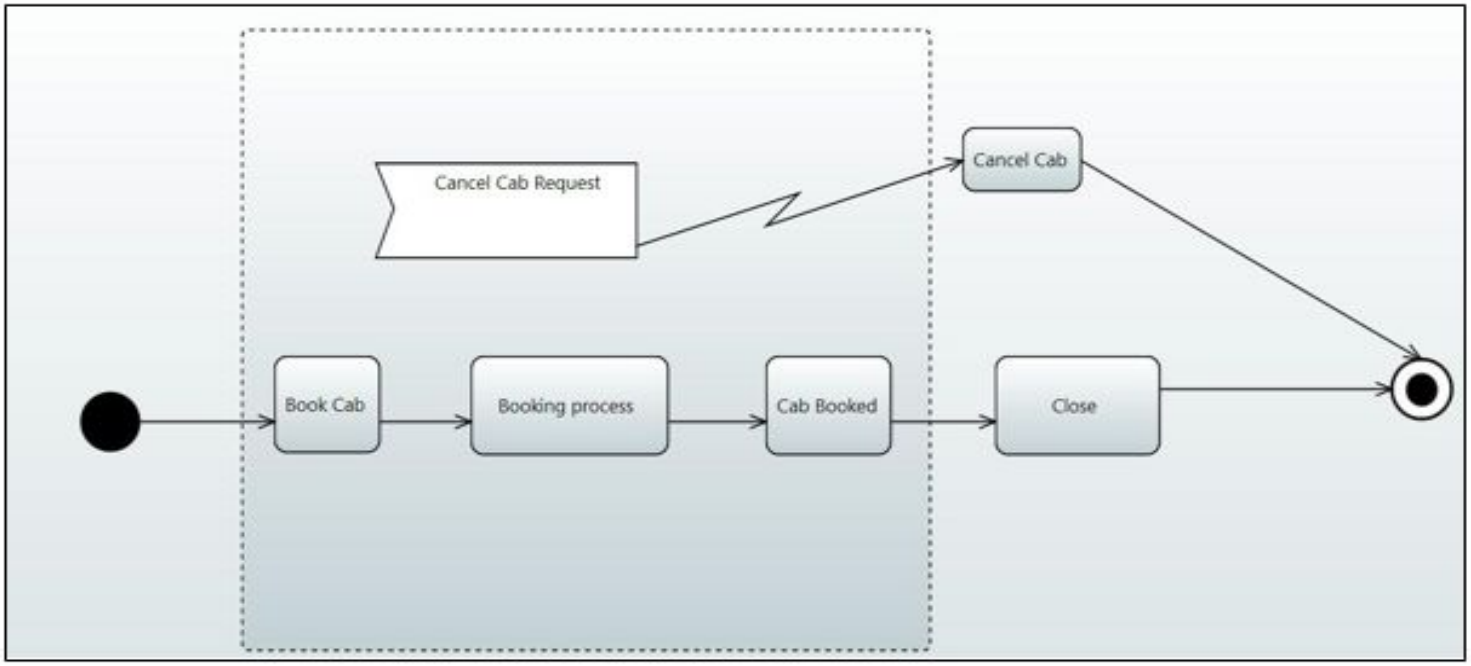
Interruptible Activity Region

- An interruptible activity region surrounds a group of actions that can be interrupted. In the very simple example below, 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



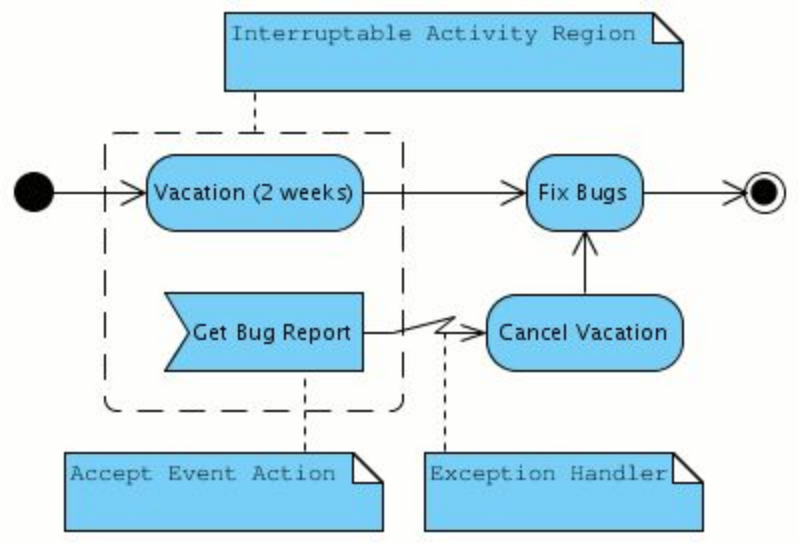
Example:

Booking of cab process is shown below this process will execute until completion, when cab is booked, search for nearest captain starts (Booking process), when search is complete cab is booked and process is finished but what happens if you cancel a request to cancel the cab, a cancellation request should interrupt these various actions that are going on bringing them to a halt. It can be represented by Interruptible region. The actions that could be interrupted are put inside this interruptible and the event that can cause the interruption and in the case of our diagram here that would be an input signal and that input signal would be a Cancel Cab Request



Interrupting an Activity

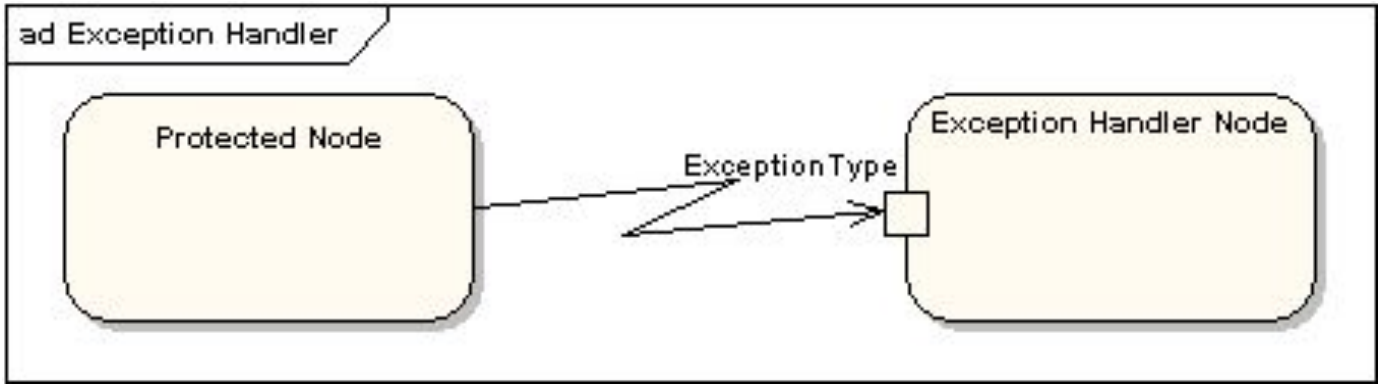
- Longer running processes can be interrupted by an event - an ***accept event action*** within an ***interruptable activity region:***

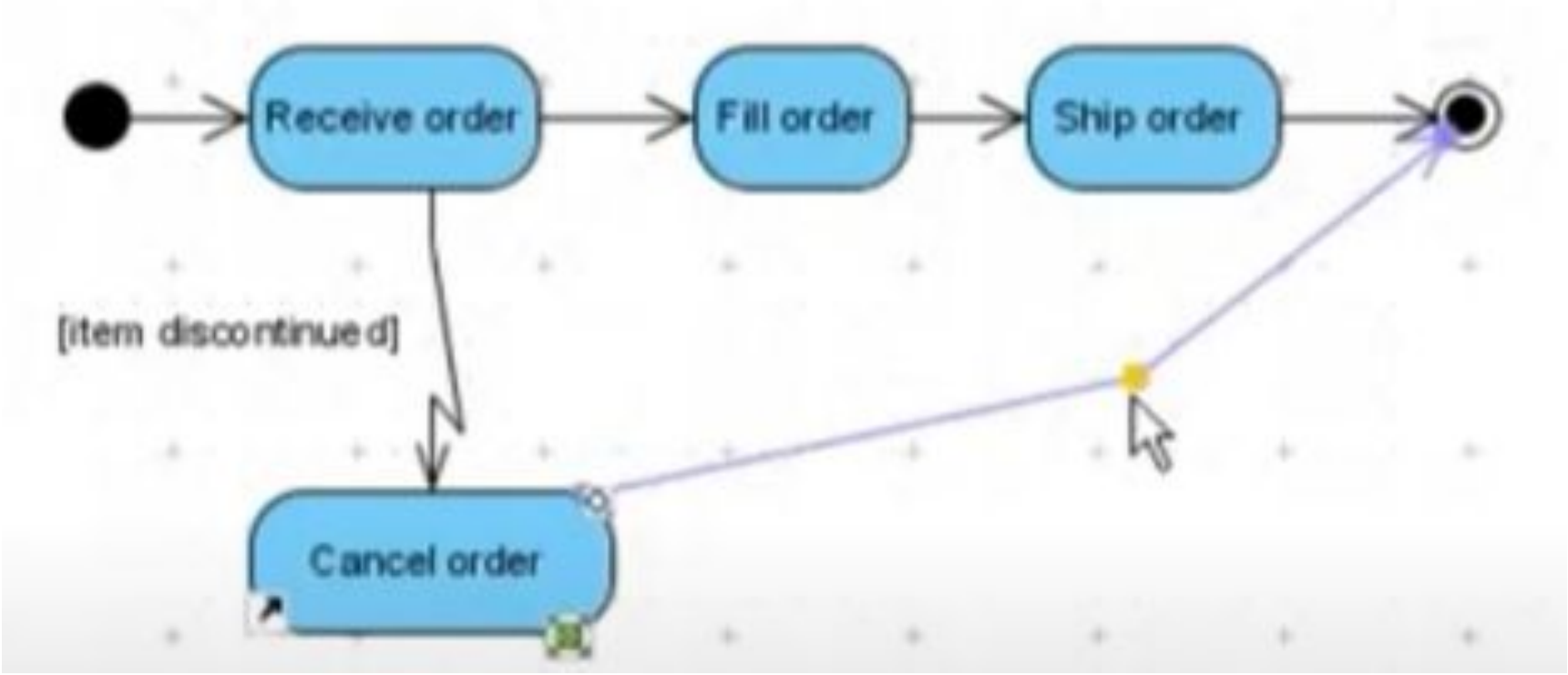


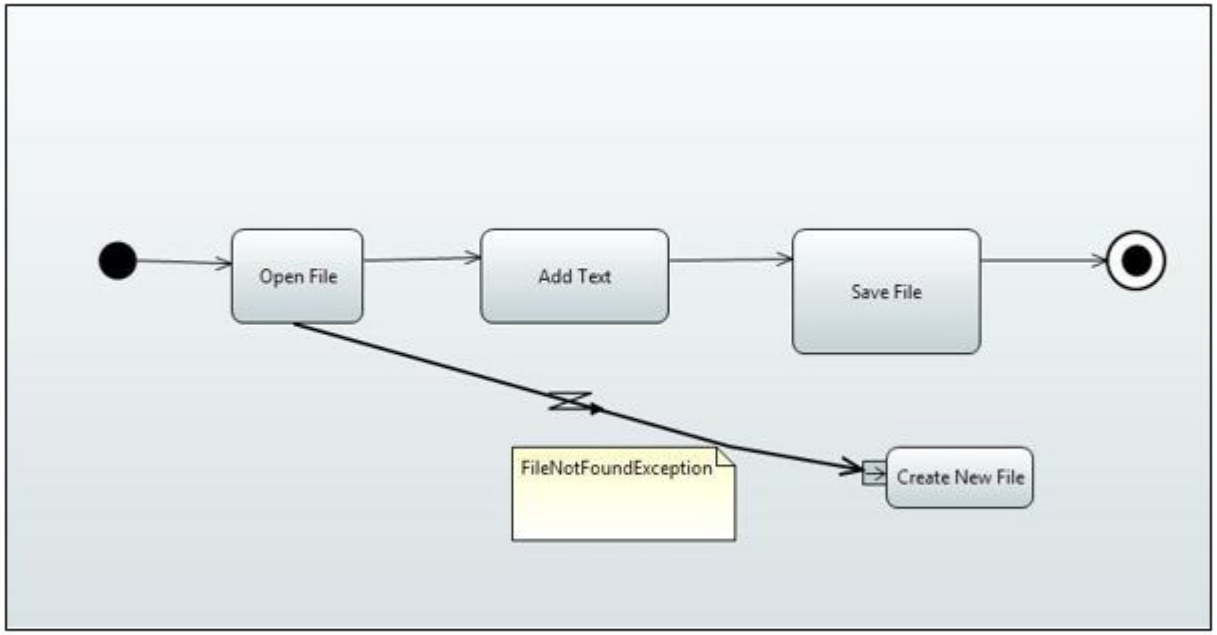
Exception Handling

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

The exceptions in the activity diagrams are the events or actions that occur during the execution of a program that interrupts the normal execution.

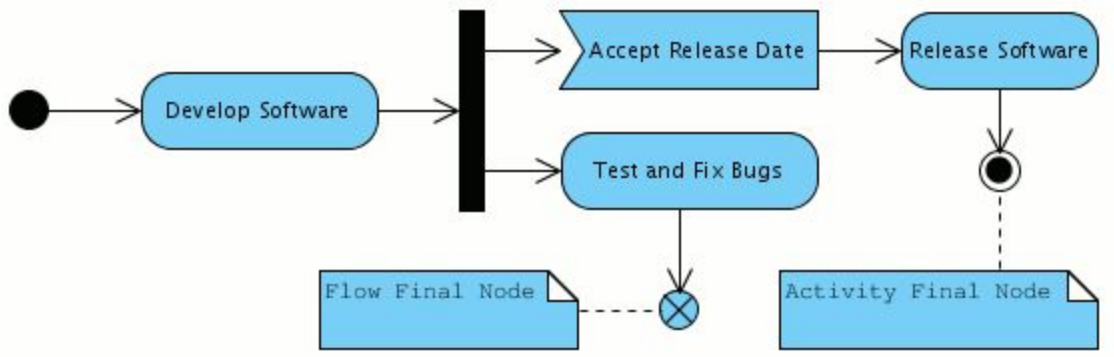



Receive order
Fill order
Ship order
[item discontinued]
Cancel order

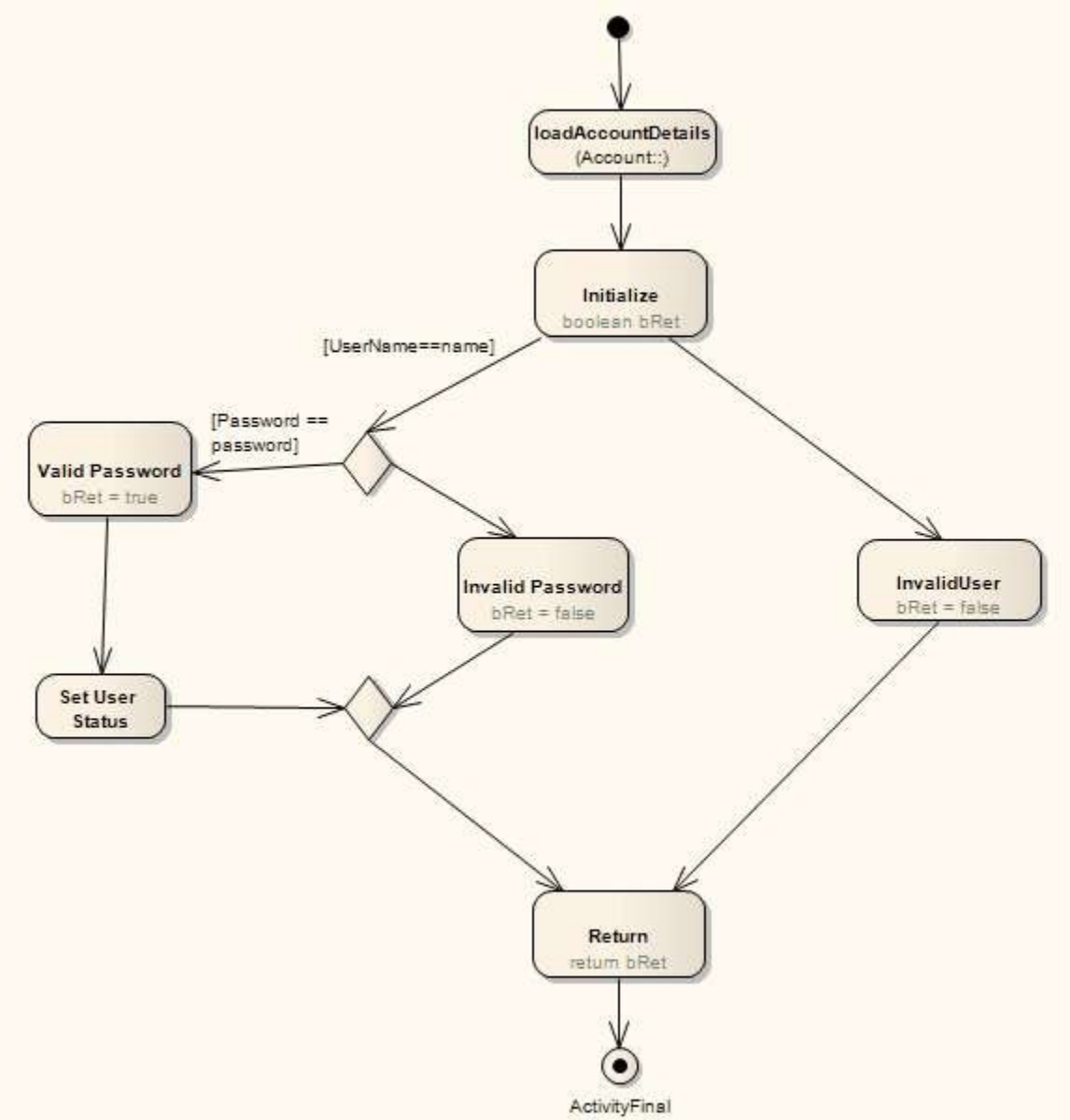

Open File
Add Text
Save File
FileNotFoundException
Create New File

Ending a Flow

- Reaching a ***flow final node*** ends a flow (not the activity). The following diagram models a scenario in which software is tested and bugs are fixed - until the release date is due:



Conditional Statements



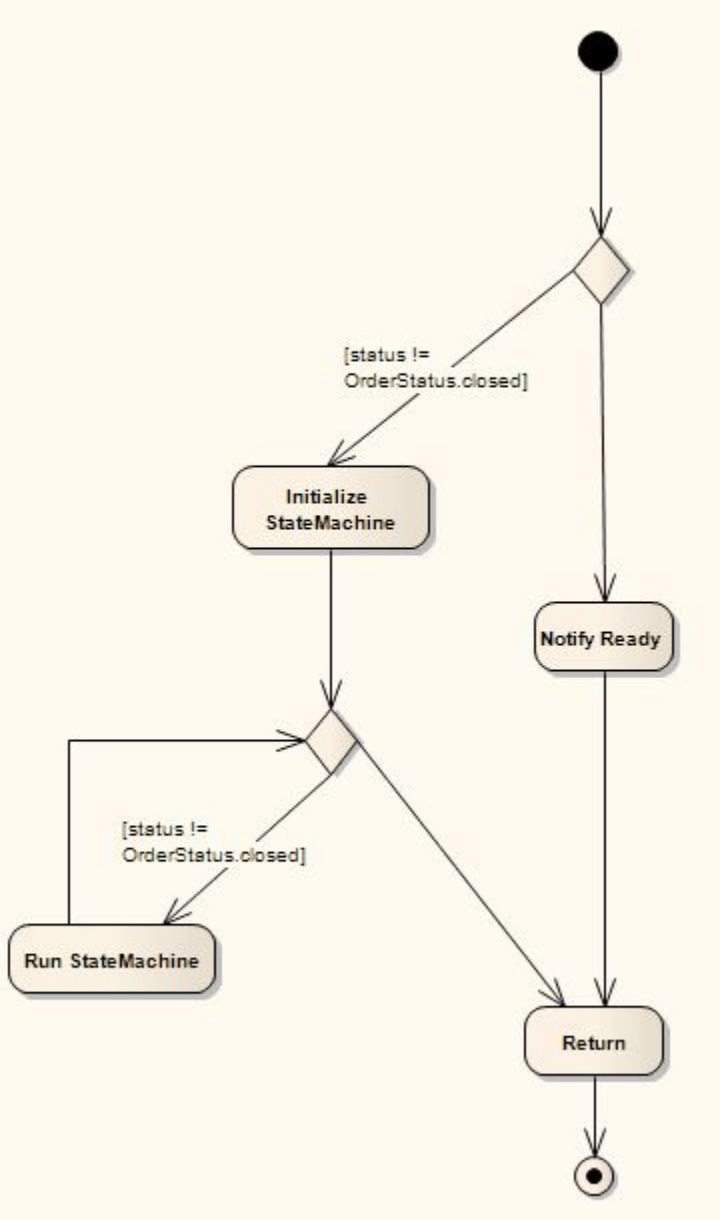
Conditional Statements

```
public boolean doValidateUser(String Password,String
UserName)
    {
        loadAccountDetails();
        boolean bRet;
        if (Username==name)
        {
            if (Password == password)
            {
                bRet = true;
                bValidUser =
                true;
            }
            else
            {
                bRet = false;
            }

        }
        else
        {
            bRet = false;
        }

        return bRet;
```

Loops


[status !=
OrderStatus.closed]
Initialize
StateMachine
Notify Ready
[status !=
OrderStatus.closed]
Run StateMachine
Return

Loops

```
public void doCheckForOutstandingOrders()
    {
        if (status != closed)
        {
            initializeStateMachine();
            while (status != closed)
            {
                runStateMachine();
            }
        }
        else
        {
            //No Outstanding orders;
        }
        return;
    }
```