Activity Diagrams

Activity diagrams are constructed from a limited repertoire of shapes, connected with arrows. The most important shape types:

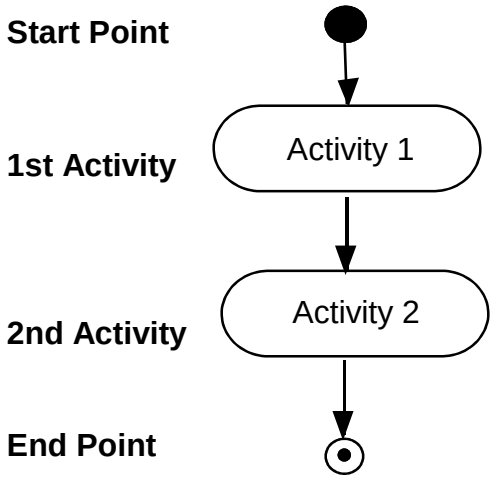
* rounded rectangles represent activities;
* diamonds represent decisions;
* bars represent the start (split) or end (join) of concurrent activities;
* a black circle represents the start (initial state) of the workflow;
* an encircled black circle represents the end (final state).

Arrows run from the start towards the end and represent the order in which activities happen.

Hence they can be regarded as a form of flowchart. Typical flowchart techniques lack constructs for expressing concurrency. However, the join and split symbols in activity diagrams only resolve this for simple cases; the meaning of the model is not clear when they are arbitrarily combined with decisions or loops.

First and foremost, an activity diagram is designed to be a simplified look at what happens during an operation or a process. It's an extension of the state diagram. The state diagram shows the states of an object and represents activities as arrows connecting the states. The activity diagram highlights the activities.

Each activity is represented by a rounded rectangle. The processing within an activity goes to completion and then an automatic transmission to the next activity occurs. An arrow represents the transition from one activity to the next. Also an activity diagram has a starting point represented by filled-in circle, and endpoint represented by a bull's eye.



Transition from one activity to another in the Activity Diagram

\Building an Activity Diagram

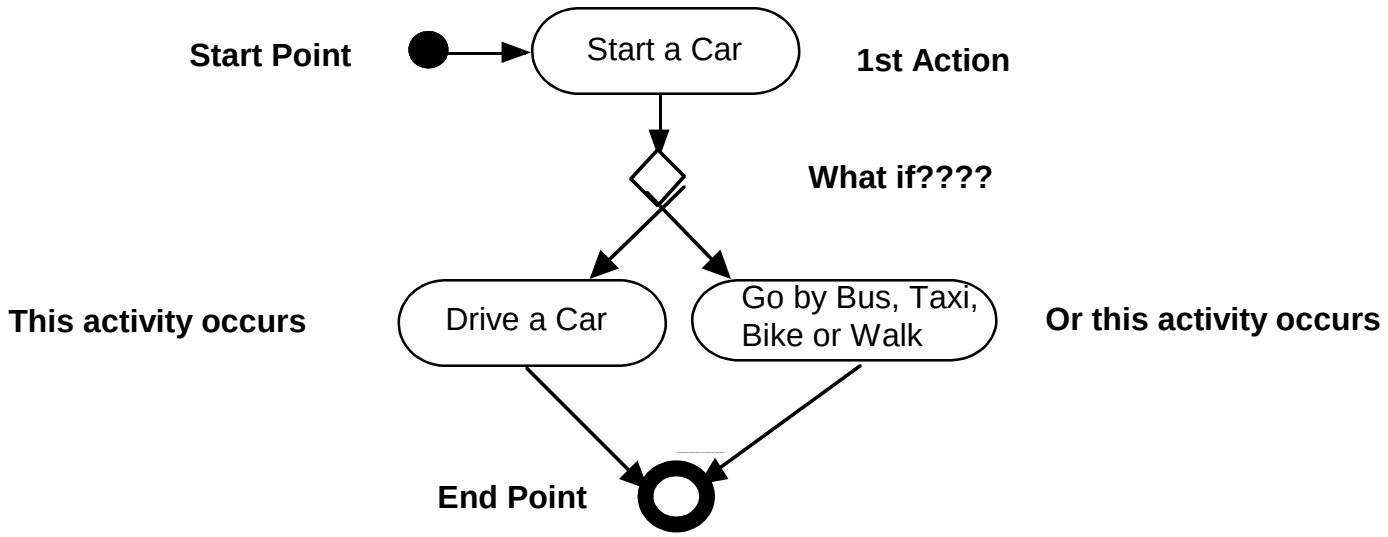
There are ways to represent decisions, concurrent paths, signals and swimlanes in the activity diagrams.

Decisions

A decision point can be represented in two ways. It's up to you which method you use in your activity diagrams.

One way is to show the possible paths coming directly out of an activity, the other is to have the activity transition to a small diamond and have the possible paths flow out of the diamond. Either way, you indicate the condition with a bracketed condition statement near the appropriate path.

Imagine that you have to go to the work. You get your car, put the key in the ignition, and there are two possible situations: your car will start or it won't. These possible cases will produce two other activities: drive a car, or go by a bus, taxi, bike etc.. This scenario is shown on this picture (pay attention to the two ways of indicating a decision):

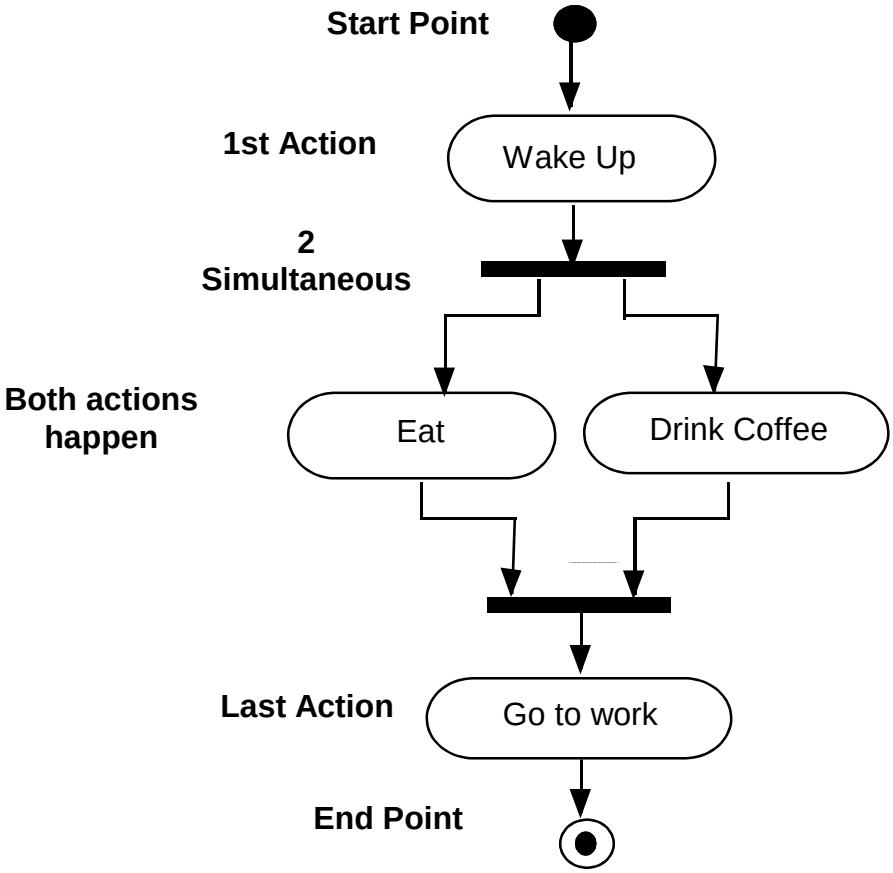


Activity diagram demo of two ways of indicating a decision

Concurrent paths

When you model activities, very frequently you'll have an occasion to separate a transition into two separate paths that run at the same time (concurrently), and then come together.

The split is represented by a solid bold line perpendicular to the transition and shows the paths coming out of the line. To represent the merge, show the paths pointing at another solid bold line:

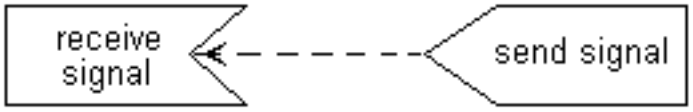


Activity diagram representing a transition split into two paths that run concurrently and then come together.

Signals

During a sequence of activities, it's possible to send a signal. When received, the signal causes an activity to take a place. This could be when attempting to read/write a file (messages are sent back by the OS - think of the Exception classes!)

The symbol for sending a signal is a convex pentagon, and the symbol for receiving a signal is a concave polygon.



Sending and receiving a signal

Swimlanes

The activity diagram adds the dimension of visualizing roles. To do that, you separate the diagram into parallel segments called swimlanes. Each swimlane shows the name of a role at the top, and represents the activities of each role. Transitions can take place from one swimlane to another.

It's possible to combine the activity diagram with the symbols from other diagrams and thus produce a hybrid diagram.

Activity Diagram - Example

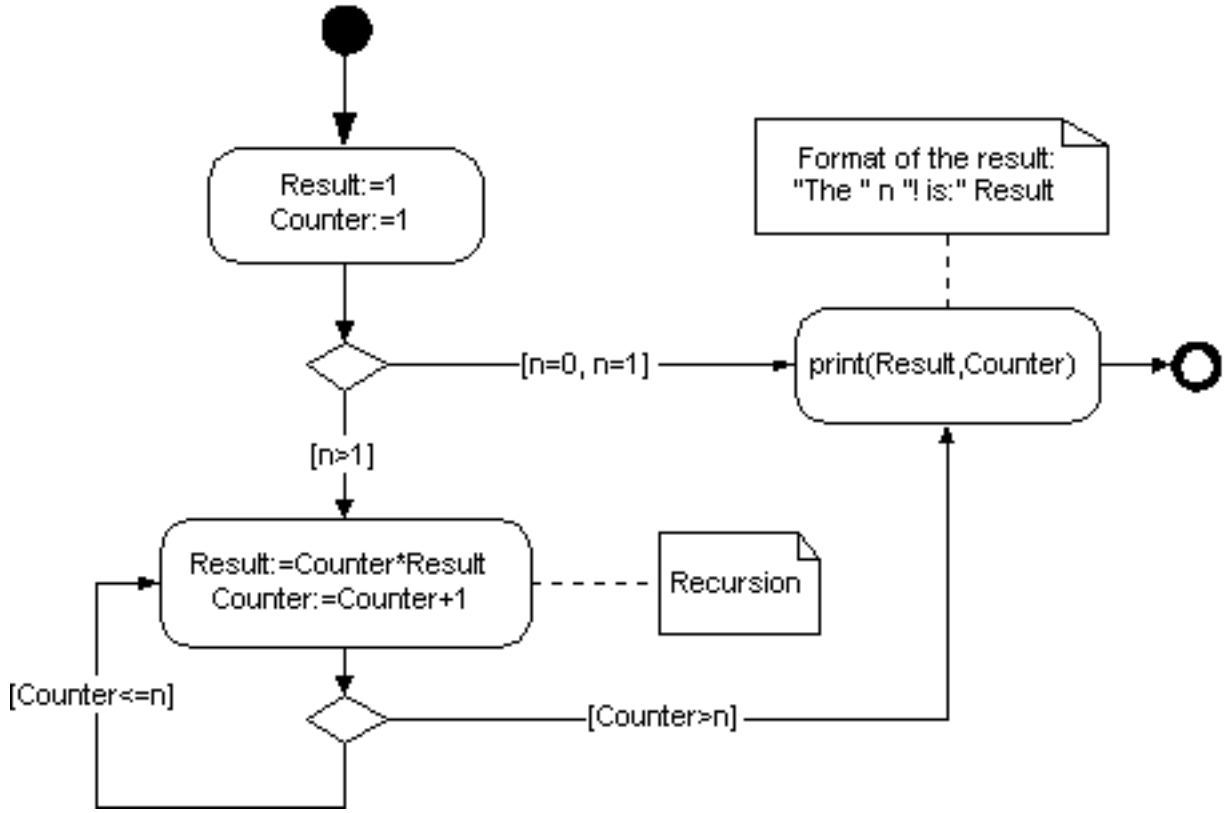
This example will deal with mathematics. Sometimes (when calculating combinations) you'll need to calculate a factorial of n - n!.

The formula for this is n! = n * (n-1)*(n-2)*...*2*1.

Let's dive into the problem. From the definition of factorials we have 0!=1 and 1!=1. For the rest of the numbers we must use the given formula. In computer programming, this problem is written with recursion. You might call the operation computeFact(n). You'll need a counter to keep track of whether or not the operation has reached the nth factorial, a variable that keeps a track of your computations, and two more to store the values of 0! and 1!.

3

The complete activity diagram is shown below:



If you're dealing with computer programming you will conclude that this activity diagram is very similar to a method that computes factorials.

More About Interfaces and Components

In previous lessons, you learned about diagrams that deal with conceptual entities. In next lesson, you're going to learn about a UML diagram that represent a real-world entity: the software component.

But what is a software component?

A software component is a physical part of a system. It resides in a computer, not in the mind of an analyst.

What's the relationship between a component and a class?

Think of a component as the software implementation of a class. The class represents an abstraction of a set of attributes and operations and one component can be the implementation of more than one class.

You model components and their relationships so that:

1. Clients can see the structure in the finished system
2. Developers have a structure to work toward
3. Technical writers who have to provide documentation and help files can understand what they're writing about

4. The system is maintainable and ready for reuse

When you deal with components, you have to deal with their interfaces. An interface is the object's "face" to the outside world, so that other objects can ask the object to execute its operations, which are hidden within **encapsulation**.

An interface is a set of operations that specifies something about a class's behaviour. It is a set of operations the class represents to other classes. For you as a modeller, this means that the way you represent an interface for a class is the same as the way you represent an interface for a component. As is the case with a class and its interface, the relation between a component and its interface is called a **realization**. See, Interfaces & Realizations.

You can replace one component with another if the new component conforms to the same interfaces as the old one. You can reuse a component in another system if the new system can access the reused component through that component's interfaces.

As you progress in your modelling career, you'll deal with three kinds of components:

1. Deployment components, which form the basis of executable systems (DLL's, executables, ActiveX controls, JavaBeans)
2. Work product components, from which deployment components are created (data files and source code files)
3. Execution components, created as result of a running system

Instead of representing a conceptual entity such as a class or a state, a component diagram represents a real-world item - a software component. Software components reside in computers, not in the minds of analysts.
A component is accessible through its interface. The relation between a component and its interface is called a realization. When one component access the services of another, it uses an import interface. The component that realizes the interface with those services provides an export interface.