Activation Policies

This chapter covers the following topics:

- Activation Policies on Windows Platforms
- Setting Activation Policies
- When to Use Which Activation Policy

Activation Policies on Windows Platforms

If a client makes a request to create an object of a certain class, it is DCOM's task to start a server process that provides the class and to direct the request to this process. For Natural classes, the responsible server process is a NaturalX server. DCOM recognizes different options that control when a new server process is started or when an object is created in a server process that is already running. For further information, see the section *Registration*. While registering a Natural class with the `REGISTER` command, you can control which activation options DCOM shall use for this class. NaturalX combines the different options supported by DCOM in the form of the following three activation policies:

- **ExternalMultiple**
  If a Natural class is registered with the activation policy "ExternalMultiple", and a client requests an object of that class, DCOM tries first to create the requested object in the current process. Remember that the client itself might at the same time be a NaturalX server and might provide the class itself. If the current process is not a server for the class, DCOM starts a new NaturalX server process and creates the object in that process. If a second object of the same class is created later, this object is also created in that server process. This means that the same server process can contain several objects of the class.

- **ExternalSingle**
  If a Natural class is registered with the activation policy "ExternalSingle", DCOM starts a new NaturalX server process each time an object of this class is created. One server process can contain only one object of the class.

- **InternalMultiple**
  If a Natural class is registered with the activation policy "InternalMultiple", DCOM always creates objects of this class in the current process. The same server process can contain several objects of the class.

The default activation policy is "ExternalMultiple". This default is defined with the Natural parameter `ACTPOLICY` and can be changed with the Configuration Utility.

Setting Activation Policies

The activation policy of a class can be set in three different ways, in the following order of precedence:

- Explicity as part of the `REGISTER` command.

- In the `DEFINE CLASS` statement.
- With the profile parameter `ACTPOLICY`.

When to Use Which Activation Policy

Non-trivial DCOM applications will mostly deal with "persistent" objects, i.e. objects stored in databases. For such applications, some considerations concerning database access, transaction handling and user isolation must be made. Consider the following scenario: clients A and B both create an object of a class that is provided by a certain NaturalX server process. Assume that the NaturalX server uses a database to load and store its objects. If both clients were served by the same server process, they would appear to the database as one single user. This would have the consequence that a transaction started by a method call from Client A can be committed or backed out by a method call from Client B. Such interferences are obviously to be avoided.

There are two approaches to avoid this interference: either the clients do not use persistent objects, or each of them is served by its own NaturalX server process. Both approaches have their advantages in different situations; for a class or application that does not access databases or other shared resources, it is useful to serve several clients with a single server process. For classes that access databases or other shared resources, it is necessary to isolate different clients in different server processes. Hence both approaches should be possible. Activation policies give an administrator the means to control the activation behavior for each class at registration time.

Example

This example illustrates how the various activation policies can be used. Let us consider parts of an imaginary travel agency application. The application contains the business classes `Trip`, `Skipper` and `RoutePlanner`. The `Trip` class represents a sailing trip to be planned; the `Skipper` class represents the skippers available to lead the trips. `RoutePlanner` is a class that determines an optimal route for a trip. Assume that the `Trip` and `Skipper` classes use a database to read and store their objects. The `RoutePlanner` class just performs some calculations on a given `Trip` object and does not use a database.

Since some of the business classes use transactional access to a database, and a transaction might span several method calls, each active client needs to be served with its own NaturalX server process. This can be done by defining an additional class `SagTours`, which represents an application session. This class can be used, for example, to keep general information about the session status, but the main task will be to create business objects on behalf of a client.

Class SagTours

```
* Represents a SagTours application session.
  *
  define class SagTours
    local using tour-ids
    id clsid-sagtours
  *
    interface Create  /* Used to create application objects. */
      id iid-sagtours-create
  *
      method newTrip  /* Creates a new Trip object. */
        is trip-n
        parameter
        1 trip handle of object by value result
```

```
      end-method


      method newSkipper   /* Creates a new Skipper object. */
        is skip-n
        parameter
        1 skipper handle of object by value result
      end-method
  *
  end-interface
  *
  end-class
  end
```

This class will be registered as "ExternalSingle". This means that each creation of a `SagTours` object starts a NaturalX server process for the client that requested the object. A client will create a `SagTours` object only once and will use its methods later to create the business objects it needs. In order to create a `Trip` object, the client will call the method `newTrip`, which is implemented as follows.

Method newTrip

```
* This method creates a new Trip object.
*
define data parameter
1 trip handle of object by value result
end-define
*
create object trip of class "Trip"
*
end
```

The `Trip` class itself will be registered as "InternalMultiple". This ensures that the `Trip` objects created by the method `newTrip` are created in the NaturalX server process just started for this client.

Now let us look at the class `RoutePlanner`.

Class RoutePlanner

```
* Plans optimal routes for sailing trips.
  *
  define class RoutePlanner
    local using tour-ids
    id clsid-planner
  *
    interface routing
      id iid-planner-routing
  *
      method plan   /* Plans a sailing trip. */
        is plan-n
        parameter
        1 trip handle of object by value
      end-method
  *
    end-interface
  *
  end-class
  end
```

Method plan

```
* This method plans a sailing trip.
*
define data parameter
1 trip handle of object by value
end-define
*
* Perform some operations on the given Trip object.
*
end
```

This class can be registered as "ExternalMultiple". In this case, all `RoutePlanner` objects created by different clients would be created in the same NaturalX server process. This does not do any harm if the methods of this class do not access databases, or if each database transaction is fully contained in a method (i.e. if each method subprogram ends with either a `BACKOUT TRANSACTION` statement or an `END TRANSACTION` statement).

Now let us look at a sample client program.

Sample Client Program

```
define data local
 sagTours handle of object
 trip handle of object
 planner handle of object
end-define
*
* Start the application session.
create object sagTours of "SagTours"
*
* Create a Trip object.
send "newTrip" to sagTours return trip
* Create a RoutePlanner object.
create object planner of "RoutePlanner"
* Plan the trip.
send "plan" to planner with trip
*
end
```

The client first creates a `SagTours` object. This starts a new NaturalX server process exclusively for this client. The client then uses the `SagTours` object to create a `Trip` object in the context of this application session. Note that the client creates the `RoutePlanner` object directly. This is possible because the class is registered as "ExternalMultiple", but it is not necessary: the `SagTours` class could also provide a method for the creation of `RoutePlanner` objects. Afterwards it lets the business objects do their jobs. The objects are automatically released at program end. The deletion of the `SagTours` object causes the NaturalX server to shut down.

Note:
This example shows only the NaturalX techniques needed to illustrate the usage of activation policies. A real-world application would require a lot more. The classes would use object data areas and they would surely have globally unique IDs assigned. Also parameter data areas would be used instead of inline parameter declarations.