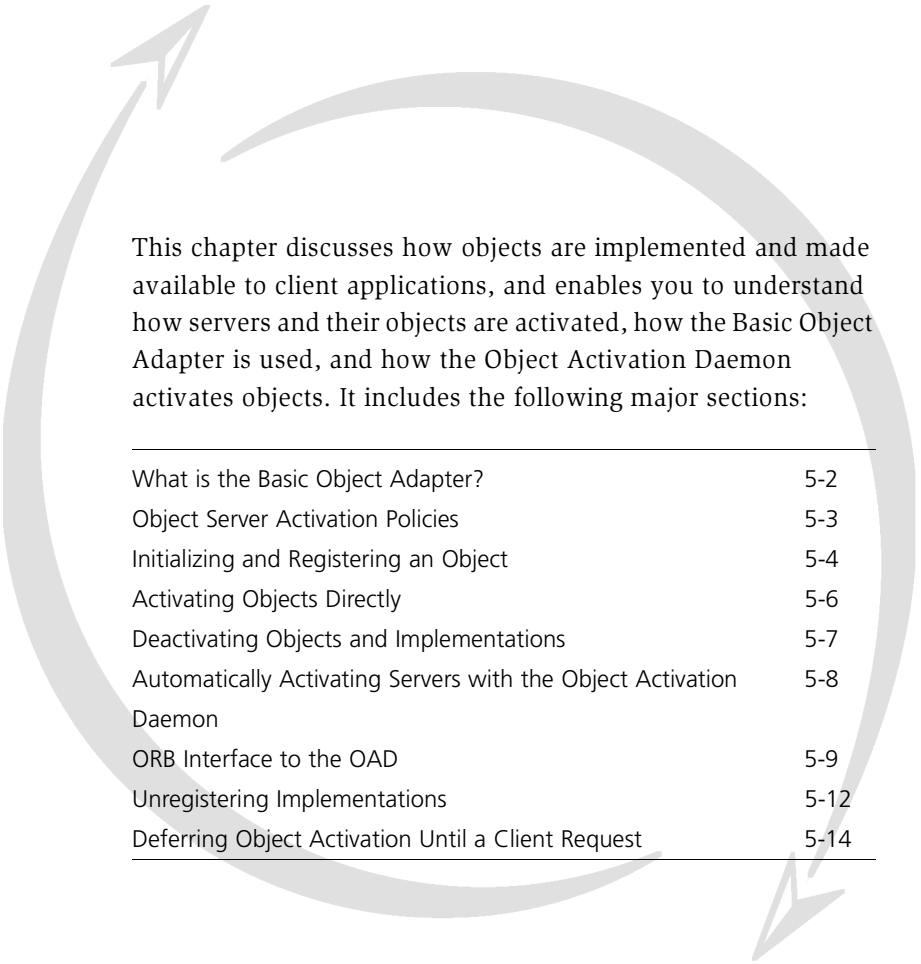
CHAPTER 5

ACTIVATING OBJECTS AND IMPLEMENTATIONS



This chapter discusses how objects are implemented and made available to client applications, and enables you to understand how servers and their objects are activated, how the Basic Object Adapter is used, and how the Object Activation Daemon activates objects. It includes the following major sections:

What is the Basic Object Adapter?	5-2
Object Server Activation Policies	5-3
Initializing and Registering an Object	5-4
Activating Objects Directly	5-6
Deactivating Objects and Implementations	5-7
Automatically Activating Servers with the Object Activation Daemon	5-8
ORB Interface to the OAD	5-9
Unregistering Implementations	5-12
Deferring Object Activation Until a Client Request	5-14

WHAT IS THE BASIC OBJECT ADAPTER?

VisiBroker's Basic Object Adapter, or BOA, provides several important functions to client applications and the object implementations they use, including:

- Providing several policies for activating object implementations, and determining how client applications can access these implementations.
- Registering object implementations with VisiBroker's Smart Agent.
- Installing and registering the object with the Implementation Repository, and activating the object upon client request, with the Object Activation Daemon.
- Storing information about object implementations residing on a server with the Implementation Repository.

It is important to realize that an object implementation may reside in the same process as its client application or it may reside in a separate process called a server. Servers may contain and offer a single object or multiple objects. Furthermore, servers may be activated by the BOA on demand or they may be started by some entity external to the BOA.

Support for Multiple Adapters

In previous versions of VisiBroker there was one adapter type, now there are several adapter types. The object adapter types that are initialized with `BOA_init()` include one of the following:

- TPool - Thread pooling
- TSession - Thread per session
- SSLTPool - SSL with thread pooling (optional feature)
- SSLTSession - SSL with thread per session (optional feature)

You use `BOA_init()` to set the object adapter type and its properties for your application. There are two versions of `BOA_init()`:

- If you use the `BOA_init()` method *with no arguments*, you accept the default thread policy which is thread pooling.
- If you use `BOA_init()` *with arguments*, you can set the adapter type and its properties. If you don't want to set any properties, you can pass a null instead.

Each time `BOA_init()` is called, it returns an instance of the object adapter type specified. It always returns the same instance of the object adapter. If you call `BOA_init()` with no arguments, it will return you an instance of TPool. If you call `BOA_init()` with a TPool argument and properties, it modifies the existing TPool. If you repeatedly call `BOA_init()` with the same type, it updates the properties, if they are different than the previous invocation of `BOA_init()`.

If you run BOA_init *with no arguments* and then invoke your application using a runtime parameter with different settings than `BOA_init()`, BOA_init is overriden by the runtime arguments. However, if you use `BOA_init()` *with arguments* and then use a runtime parameter, the BOA_init takes precedence. The runtime parameter is never even checked.

NOTE *For information of BOA options that you can set, see the VisiBroker for Java Reference Manual.*

OBJECT SERVER ACTIVATION POLICIES

VisiBroker implements activation policies that describe the way an object implementation is started and the manner in which it may be accessed by a client application. These activation policies apply only to objects with a global scope, not locally scoped objects.

- shared server
- unshared server
- server-per-method

SHARED SERVER POLICY

When the shared server policy is specified, the server can be accessed by more than one client. Multiple clients of a given object share the same implementation. Only one implementation will be activated by an OAD at a time.

UNSHARED SERVER POLICY

Unshared servers are accessed by one client only. Only one client of a given implementation will ever be bound to the activated server. If multiple clients wish to bind to the same object implementation, a separate server is activated for each client application. A server exits when its client application disconnects or exits.

SERVER-PER-METHOD POLICY

This activation policy requires a server process to be started for each method that is invoked. After the method has been completed, the server will exit. Subsequent method invocations on the same object will require a new server process to be started.

INITIALIZING AND REGISTERING AN OBJECT

An object implementation provides the state and processing activities for an ORB object. An ORB object is created when its implementation class is instantiated in Java by an implementation process or server. An object implementation uses the Basic Object Adapter, or BOA, to activate its ORB objects so that they can be used by client applications.

Transient Object References

Object references that are only available during the lifetime of the process that created them are called *transient object references* and they are not registered with the Smart Agent or the Object Activation Daemon (OAD), and, consequently, they are not visible to the Smart Agent or to the OAD. Only those entities that possess an explicit object reference to a transient object reference may invoke its methods. You can create a transient object reference by simply not specifying an object name when you instantiate the object, as shown in Code Sample 5-1.

CODE

```
// Server.java

public class Server {

  public static void main(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Initialize the BOA.
    org.omg.CORBA.BOA boa = orb.BOA_init();
    // Create the account manager object.
    Bank.AccountManager manager =
      new AccountManagerImpl();
    // Export the newly created object.
    boa.obj_is_ready(manager);
    System.out.println(manager + " is ready.");
    // Wait for incoming requests
    boa.impl_is_ready();
  }
}
```

Code Sample 5-1 Creating a transient object.

Persistent Object References

You create a *persistent object reference* when you instantiate an object and specify an object name. Persistent object references remain valid beyond the lifetime of the process that create them. These object references have a global scope and are registered with the

osagent and used by client applications. Persistent object references are registered with the osagent when the BOA.obj_is_ready method is invoked.

You can use persistent object references to implement long-running servers that provide long-term tasks. Light-weight and domain-specific, transient object references take a relatively short amount of time to instantiate. Persistent object references take longer to instantiate. Consequently, the ideal mix is to have some persistent object references and many more transient object references used by your application. Code Sample 5-2 shows the creation of a persistent object reference.

CODE

```
// Server.java

public class Server {
  public static void main(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
    // Initialize the BOA.
    org.omg.CORBA.BOA boa = orb.BOA_init();
    // Create the account manager object.
    Bank.AccountManager manager =
      new AccountManagerImpl;("BankManager");
    // Export the newly create object.
    boa.obj_is_ready(manager);
    System.out.println(manager + " is ready.");
    // Wait for incoming requests
    boa.impl_is_ready();
  }
}
```

Code Sample 5-2 Creating a persistent object.

Checking for Persistent Object References

The Object._is_persistent method allows your client application to determine whether an object reference is persistent or transient. It is important to know whether or not a reference is persistent because some methods for manipulating object references will fail if the reference is transient. The _is_persistent method returns true if the reference is persistent and false if it is transient.

For information about all of the available methods, see the *VisiBroker for Java Reference Manual*.

Registering Objects with the BOA

Once a server has instantiated the ORB objects that it offers, the BOA must be notified when the objects have been initialized. Lastly, the BOA is notified when the server is ready to receive requests from client applications.

The `obj_is_ready` method notifies the BOA that a particular ORB object is ready to receive requests from client applications. If your server offers more than one ORB object, it must invoke the `obj_is_ready` method for each object, passing the object reference as an argument.

If a persistent object reference is passed to the `obj_is_ready` method, the BOA will register the object with VisiBroker's osagent. If the reference is transient, no such registration will occur.

NOTE *If* `obj_is_ready` *has not been invoked for an object by its server and a client attempts to bind to the object, the exception* `NO_IMPLEMENT` *is raised.*

Once all of the objects have been instantiated and all the invocations of the `obj_is_ready` method have been made, the server may invoke the `impl_is_ready` method to enter an event loop and await client requests. Servers that are activated through a GUI (graphical user interface) or are activated by other means, do not need to use `impl_is_ready`.

ACTIVATING OBJECTS DIRECTLY

In the Bank example introduced in Chapter 2, the `AccountManager` object was activated directly by the server. Direct activation of an object involves an object server instantiating all the Java implementation classes, invoking the `boa.obj_is_ready` method for each object, and then invoking `boa.impl_is_ready` to begin receiving requests. Code Sample 5-3 shows how this processing would occur for a server offering two `AccountManager` objects; one with the object name of "Chicago" and the other named "Dallas". Once the objects have been instantiated and activated, the server invokes `boa.impl_is_ready` to begin receiving client requests.

NOTE *The* `boa.obj_is_ready` *must be called for each object offered by the implementation. If you attempt to pass an object and you haven't called* `boa.obj_is_ready` *for that object, then it will do it for you with a default BOA.*

CODE

```
// Server.java

public class Server {

  public static void main(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Initialize the BOA.
    org.omg.CORBA.BOA boa = orb.BOA_init();
    // Create the account manager object for Chicago.
    Bank.AccountManager chicago =
      new AccountManagerImpl("Chicago");
    // Create the account manager object for Dallas.
    Bank.AccountManager dallas =
      new AccountManagerImpl("Dallas");
    // Export the newly created objects.
    boa.obj_is_ready(chicago);
    boa.obj_is_ready(dallas);
    System.out.println(chicago + " is ready.");
    System.out.println(dallas + " is ready.");
    // Wait for incoming requests
    boa.impl_is_ready();
  }
}
```

Code Sample 5-3 Server activating two objects and the implementation.

DEACTIVATING OBJECTS AND IMPLEMENTATIONS

An object implementation that is started manually can be considered deactivated when the server that implements the object exits. At that time, VisiBroker automatically unregisters the objects within that implementation from the list maintained by the osagent's directory service.

For implementations that called `boa.obj_is_ready`, use the `boa.deactivate_obj` method to deactivate them explicitly. The method `boa.deactivate_obj` must be called to allow Java garbage collection to eventually reclaim that object. Once this method is called, the implementation will not be available to service client requests.

NOTE *When using* `boa.deactivate_obj`*, this method only deactivates the particular object upon which it is invoked. To exit from a Java server, use* `ORB.shutdown()` *to shut the server down;* `ORB.shutdown()` *is a Java call.*

The implementation can only be re-activated if it is restarted or if it again calls the `boa.obj_is_ready` method.

AUTOMATICALLY ACTIVATING SERVERS WITH THE OBJECT ACTIVATION DAEMON

The Object Activation Daemon (OAD) is Visigenic's implementation of the Implementation Repository. The Implementation Repository provides a run-time repository of information about the classes a server supports, the objects that are instantiated, and their IDs. In addition to the services provided by a typical Implementation Repository, the OAD is used to automatically activate an implementation when a client requests a bind to the object. Object implementations can be registered using a command-line utility or the ORB interface to the OAD described in "ORB Interface to the OAD" on page 5-9. In each case, the interface name, object name, the activation policy, and the executable program representing the implementation must be specified.

NOTE *You can use the VisiBroker for Java OAD to instantiate servers generated with VisiBroker for Java (any release) and VisiBroker for C++ release 3.0 . To instantiate a 2.x C++ server, use system property VisiCpp2Compat with the OAD executable. For example: oadj -VBJprop VisiCpp2Compat.*

The OAD is a separate process that can register servers and only needs to be started on those hosts where object servers are to be activated on demand. See the *VisiBroker for Java Reference Manual* or the *VisiBroker for Java Installation and Administration Guide* for information on starting the OAD.

Locating the Implementation Repository Data

All object implementations registered with the OAD are stored in an implementation repository, maintained by the OAD. By default, the implementation repository data are stored in a file named `impl_rep`. This file's path name is dependent on where VisiBroker was installed on your system. If VisiBroker was installed in `/usr/local/vbroker/`, the path to this file would be `/usr/local/vbroker/adm/impl_dir/impl_rep`. These defaults can be overridden using the OAD environment variables, described in the *VisiBroker for Java Reference Manual*.

The OAD has a single uniform command-line interface to the tools that enable you to register, unregister, or list objects. Options for the commands are documented in the *VisiBroker for Java Reference Manual*.

ORB INTERFACE TO THE OAD

The Object Activation Daemon is implemented as an ORB object. IDL Sample 5-1 shows the IDL interface specification for the OAD. You can create a client application that binds to the OAD and uses this interface to register, unregister, list implementation information, and query the status of objects that have been registered. The code sample below shows some of the list and query status methods you can use from the OAD interface.

IDL

```
module Activation
{
    enum State {
      ACTIVE,
      INACTIVE,
      WAITING_FOR_ACTIVATION
    };
...
    interface OAD {
...
      extension::CreationImplDef get_implementation(
        in CORBA::RepositoryId repId,
        in string object_name)
         raises ( NotRegistered);
      ImplementationStatus get_status(in CORBA::RepositoryId repId,
        in string object_name) raises ( NotRegistered);
      ImplStatusList get_status_interface(in CORBA::RepositoryId repId)
        raises (NotRegistered);
      ImplStatusList get_status_all();
    };
...
};
...
```

IDL Sample 5-1 Some of the query status methods in the OAD interface.

OAD Registration with `oadutil reg`

The `oadutil reg` command can be used to register an object implementation from the command line or from within a script. The required parameters are the interface name, object name and path name. If the activation policy is not specified, shared server is the default. If you do not specify a host name, the OAD will look for the object locally only. The following example shows how to register the Server from the Bank example:

EXAMPLE

```
oadutil reg -i Bank::AccountManager -o BankManager -java Server -p shared
```

NOTE *The OAD must be running before you can use any of the OAD commands.*

For complete information on using this command, see the *VisiBroker for Java Reference Manual*.

You may write an implementation and start it manually during the development and testing phases. When your implementation is ready to be deployed, you can simply use `oadutil reg` to register your implementation with the OAD.

NOTE *When registering an object implementation, use the same object name that is used when the implementation object is constructed.*

OAD Registration using `OAD::reg_implementation`

Instead of using the `oadutil reg` command manually or in a script, VisiBroker allows applications written in Java to use the `OAD::reg_implementation` method to register one or more objects with the activation daemon. Using this method results in an object implementation being registered with the OAD and the osagent. The OAD will store the information in the implementation repository, allowing the object implementation to be located and activated when a client attempts to bind to the object.

IDL

```
module Activation {
...
    typedef sequence<ObjectStatus> ObjectStatus List;
...
    typedef sequence<InplementationStatus> ImplStatusList;
...
    interface OAD {
      // Register an implemenation.
      Object reg_implementation(in extension::CreationImplDef impl)
        raises (DuplicateEntry, InvalidPath);
    }
}
```

IDL Sample 5-2 The `OAD::reg_implementation` *method*

The `ImplementationDef` class, as defined by the CORBA specification, does not supply all the information that the OAD needs to activate an object implementation. The `CreationImplDef` interface is derived from `ImplementationDef` and adds the properties the OAD requires. The properties added are `path_name`, `policy`, `args`, and `env`.

Methods for setting and querying their values are also provided. These additional properties are used by the OAD to activate an ORB object.

IDL

```
...
interface CreationImplDef : CORBA::ImplementationDef {
    attribute CORBA::RepositoryId     repository_id;
    attribute string                  object_name;
    attribute CORBA::ReferenceData    id;
    attribute string                  path_name;
    attribute CORBA::Policy           activation_policy;
    attribute CORBA::StringSequence   args;
    attribute CORBA::StringSequence   env;
};
```

IDL Sample 5-3 The `CreationImplDef` *interface*

The `path_name` property specifies the exact path name of the executable program that implements the object. The `activation_policy` property represents the server's activation policy. The `args` and `env` properties represent optional arguments and environment settings to be passed to the server.

Changing an ORB Implementation

The `OAD::change_implementation` method can be used to dynamically change an object's implementation. You can use this method to change the object's activation policy, path name, arguments and environment variables.

IDL

```
class OAD {
    ...
    void change_implementation(in extension::CreationImplDef old_impl,
        in extension::CreationImplDef new_impl)
        raises (NotRegisterd, InvalidPath, IsActive);

}
```

IDL Sample 5-4 The `change_implementation` *method.*

CAUTION *You cannot change information for a currently active implementation. Also, though you can change an object's implementation name and object name with this method, you should exercise caution. Doing so will prevent client applications from locating the object with the old name.*

UNREGISTERING IMPLEMENTATIONS

When the services offered by an object are no longer available or temporarily suspended, the object should be unregistered with the OAD. When an ORB object is unregistered, it is removed from the OAD's list of objects. The object is also removed from the osagent's directory service and from the implementation repository. Once an object is unregistered, client applications will no longer be able to locate or use it. In addition, you cannot use `OAD::change_implementation` method to change the object's implementation. As with the registration process, unregistering may be done with a command line or with an ORB object interface to the OAD.

Unregistering with `oadutil unreg`

The `oadutil unreg` command can be used to unregister an object implementation from the command line or from within a script. If the interface name is specified by itself, all objects instances associated with that interface name will be unregistered. You can specify both the interface and object name if you only wish to unregister a specific object within an interface; for example:

```
oadutil unreg –i Bank::AccountManager
```

For complete information on using this command, see the *VisiBroker for Java Reference Manual*.

Unregistering with the OAD Methods

An object's implementation can use any one of the methods or attribute in the OAD interface to unregister an ORB object:

- **`unreg_implementation`**`(in CORBA::RepositoryId repId, in string object_name)`
- **`unreg_interface`**`(in CORBA::RepositoryId repId)`
- **`unregister_all`**`()`
- `attribute boolean` **`destroy_on_unregister`**

`unreg_implementation()` Use this method when you want to unregister implementations using a specific repository id and object name. This method terminates all processes currently implementing the specified repository id and object name.

`unreg_interface()` Use this method when you want to unregister implementations by using a specific repository id only. This method terminates all processes currently implementing the specified repository id.

`unregister_all()` Use this method to unregister all implementations. Unless `destroyActive` is set to true, all active implementations continue to execute. For backward compatability, `unregister_all()` is *not* destructive; it is equivalent to invoking `unregister_all_destroy(false)`.

`destroy_on_unregister` Use this attribute to destroy any spawned processes on unregistration of the relevant implementation. The default value is 'false'.

IDL

```
module Activation {
...
    interface OAD {
      ...
      void unreg_implementation(in CORBA::RepositoryId repId, in string
                object_name)
        raises(NotRegistered);
          ...
    }
}
```

IDL Sample 5-5 The OAD *unregister methods.*

Listing Object Implementations

You can use the `oadutil list` command to list the contents of a particular implementation repository. For each implementation in the repository the `oadutil list` command lists all the object instance names, the path name of the executable program, the activation mode and the reference data. Any arguments or environment variables that are to be passed to the executable program are also listed. For complete details on using this command see the *VisiBroker for Java Reference Manual.*

DEFERRING OBJECT ACTIVATION UNTIL A CLIENT REQUEST

When you design your object implementation, you may want to defer the activation of ORB objects until a client requests them. By deferring object activation, you benefit from greater performance. If you have thousands of objects on a server, you can save system resources by only activating the objects when clients request them. You can defer the activation of multiple object implementations with a single `Activator`—doing so is called service activation. A service is an arbitrary grouping of objects, any of which can be activated.

Service activation can be used when a server needs to provide implementations for a large number of objects (commonly thousands of objects, possibly millions) but only a small number of implementations need to be active at any specific time. The server can supply a single `Activator` which will be notified whenever any of these subsidiary objects are needed. The server can also deactivate these objects when they are not in use.

For example, you might use service activation for a server that loads objects implementations whose state is stored in a database. The `Activator` is responsible for loading all objects of a given type or logical distinction. When ORB requests are made on the references to these objects, the `Activator` is notified and creates a new implementation whose state is loaded from the database. When the `Activator` determines that the object should no longer be in memory and if the object had been modified, it writes the object's state to the database and releases the implementation.

The following steps are required to implement a server that uses service activation:

1. Define a service name that describes all objects activated and deactivated by the `Activator`.
2. Implement the `Activator` which creates the object implementations on demand.

Example of Deferred Object Activation for a Server

In this example, an implementation of the `Activator` interface is created that is responsible for activating objects for the service named "DBService". The `DBService` service is responsible for objects that implement the `DBObject` interface.

References to objects created by this `Activator` contain enough information for the ORB to locate the `Activator` for the `DBService` service, and for the `Activator` to create these objects on demand.

The code for the remaining examples in this chapter can be found in the **odb** directory in the **java_examples** directory installed with the VisiBroker for Java product.

IMPLEMENTING A SERVICE-ACTIVATED OBJECT

As shown in Code Sample 5-4, there are implementations of two interfaces. The IDL for the two interfaces is shown in IDL Sample 5-6.

IDL

```
// db.idl

interface DBObject {
  string get_name();
};
typedef sequence<DBObject> DBObjectSequence;
interface DB {
  DBObject create_object(in string name);
};
```

IDL Sample 5-6 IDL for the two interfaces implemented and shown in Code Sample 5-4.

`DBImpl` is the implementation of the `DB` interface (a logical database). The `DBObject` is an implementation of interface `DBObject`. The `DBObject` *type* is stored in a database. The `DBImpl` *type* is what accesses those objects. DBImpl creates implementations of the database objects via the `create_object()` method. It creates an object, calls `obj_is_ready` on the object, registers the object with a given service name and reference data.

The way to distinguish between various activated objects is by assigning each object an identifier. In CORBA, the identifier is known as the reference data. The reference data is a data defined by you to uniquely identify an object in the current process. If you are retrieving objects from a database, the reference data might be the key in the database to an object or an object's state, or it might be an index to a table. What is in the reference data and what it means is entirely up to you.

CODE

```
class DBObjectImpl extends _DBObjectImplBase {
  private String _name;
  DBObjectImpl(String name) {
    _name = name;
  }
  public String get_name() {
    return _name;
  }
}
```

CONTINUES ON NEXT PAGE

CODE CONTINUED FROM PREVIOUS PAGE

```
class DBImpl extends _DBImplBase {
  DBImpl(String name) {
    super(name);
  }

public DBObject create_object(String name) {
    System.out.println("Creating: " + name);
    DBObject dbObject = new DBObjectImpl(name);
    int length = name,length();
    byte[] ref_data = new byte[length];
    name.getBytes(0, length-1, ref_data, 0);
    _boa().obj_is_ready(dbObject, "DBService", ref_data);
    return dbObject;
  }
}
```

Code Sample 5-4 Example of implementing service-activated objects.

IMPLEMENTING A SERVICE ACTIVATOR

The Activator interface is shown in Code Sample 5-5. The Activator interface offers two methods: `activate()` and `deactivate()`. As shown in Code Sample 5-6, the DBActivator class implements the Activator interface. The implementation of the Activator interface activates objects when told to do so by the runtime.

CODE

```
public interface com.visigenic.vbroker.extension.Activator {
  org.omg.CORBA.Object activate(org.omg.CORBA.ImplementationDef);
  void deactivate(org.omg.CORBA.Object, org.omg.CORBA.ImplementationDef);
}
```

Code Sample 5-5 The `Activator` *interface.*

When the ORB receives a client request for an object under the responsibility of the `Activator`, the ORB invokes the `activate()` method on the `Activator`. In this method, the ORB uniquely identifies the activated object implementation by passing the `Activator` an `ImplementationDef` parameter—from which the implementation can obtain `ref_data`, the unique identifier.

In this example, nothing is done with `deactivate()`. For an implementation with a large number of objects, you might use `deactivate()` to clean up state data when you've got a loaded cache of objects.

CODE

```
class DBActivator implements Activator {
  private static int _count;
  public org.omg.CORBA.Object activate(ImplementationDef impl) {
    System.out.println("Activator called " + ++_count + " times");
    byte[] ref_data = ((ActivationImplDef) impl).id();
    DBObjectImpl obj = new DBObjectImpl
        (new String(ref_data, ref_data.length-1));
    _boa.obj_is_ready(obj);
    return obj;
  }

  public void deactivate(org.omg.CORBA.Object obj, ImplementationDef impl) {
    // nothing to do here...
  }
}
```

Code Sample 5-6 Example of implementing the `Activator`.

INSTANTIATING THE SERVICE ACTIVATOR

As shown in Code Sample 5-7, the `DBActivator` activator is created and registered with the BOA, and is responsible for all objects that belong to the `DBService` service. All requests for objects of the `DBService` service are directed through the `DBActivator` activator. All objects activated by this activator have references that inform the ORB that they belong to the `DBService` service. When the objects are activated, the reference data is used to construct the object.

The client programs that send requests to the server are not aware that the server is using the `Activator`. When using the `Activator`, there is nothing special that you have to put in the client-side code.

CODE

```
// Server.java

import org.omg.CORBA.*;
import com.visigenic.vbroker.extension.*;
import db.*;
...
public class Server {
  public static void main(String[] args) {
    ORB orb = ORB.init(args,null);
    BOA boa = orb.BOA_init();
    DB db = new DBImpl("Database Manager");
    boa.obj_is_ready(db);
    boa.impl_is_ready("DBService", new DBActivator(boa));
  }
}
```

Code Sample 5-7 Example of instantiating the service activator.

It's important to remember that objects that are to be activated via the `Activator` have two important associations: service name and reference data. When activating an object, the service name provided in the `obj_is_ready()` call must match what you specify in `impl_is_ready()`. Reference data must match up via the link between `DBActivator` in `impl_is_ready()` and the `ImplementationDef` parameter as shown in Code Sample 5-6.