CHAPTER 5

ACTIVATING OBJECTS AND IMPLEMENTATIONS

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

What is the Basic Object Adaptor?	5-2
Object Server Activation Policies	5-2
Initializing and Registering an Object	5-3
Activating Objects Directly	5-5
Automatically Activating Servers with the Object Activation Daemon	5-6
The IDL Interface to the OAD	5-12
Deferring Object Activation Until a Client Request	5-14
Deactivating Objects and Implementations	5-24

WHAT IS THE BASIC OBJECT ADAPTOR?

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

- Providing several policies for activating object implementations, and determining how client programs 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 a client program 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.

NOTE *For information on BOA options that you can set, see the VisiBroker for C++ 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 program. These activation policies apply only to objects with a global scope, not locally scoped objects.

SHARED SERVER POLICY

When the shared server policy is specified, only one server is launched regardless of the number of clients—the clients share the server. Shared servers are the most common types of servers.

UNSHARED SERVER POLICY

Unshared servers are processes that are used at most by one client. A client program causes this type of server to be activated. Once that client exits, the unshared server 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 ORB object is created when its implementation class is instantiated in C++ by an implementation process or server. An object server must then use the BOA to activate the object implementations it offers so that these implementations can receive client operation requests. Object implementations may be activated with either a *local* or a *global* scope.

Globally Scoped Objects

Objects that have a global scope are registered with the Smart Agent, described in Chapter 10, "The Smart Agent Architecture," which allows client programs to obtain references to them by invoking the `_bind()` method. References to globally scoped objects are called *persistent object references* because they remain valid beyond the lifetime of the process that obtained the reference. Globally scoped object references can be converted to a string and then saved to a file or passed to other programs. Objects with a global scope are used to implement servers that provide long-term tasks or to implement servers that are to be activated by the OAD.

NOTE *An object with a global scope is registered with the Smart Agent when the `boa::obj_is_ready()` method is invoked. Unless overridden, the default scope of any named object will be global.*

Locally Scoped Objects

Objects that are not given an object name when they are instantiated have a local scope. References to these objects are called transient object references. Only those clients that are explicitly passed a transient object reference may invoke methods on that object. You may want to specify a local scope if you want to implement lightweight objects that will not be generally available to other client programs. A common use for locally scoped objects is for call-back objects.

Code Sample 5-1 shows you how you could modify the application introduced in Chapter 3, "Quick Start for Development with VisiBroker for C++," so that the named `Account` object will not be registered with the Smart Agent.

NOTE *The desired scope must be set before the object is actually created.*

CODE
```
int main(int argc, char* const* argv) {
      try {
          // Initialize the ORB and BOA.
          CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
          CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

          // Set local registration scope
          boa->scope(CORBA::BOA::SCOPE_LOCAL);

          // Create a new account object.
          AccountImpl account("Jack B. Quick");

          // Since AccountImpl object is transient, it won't get
          // registered with the Smart Agent
          boa->obj_is_ready(&account);
          cout << account << " is ready." << endl;
          ...
```

Code Sample 5-1 Creating a named object with a local scope.

Handling Locally Scoped Object References

Clients can only access locally scoped objects when they are passed a reference to the object as an argument or return value. Object references with a local scope are only valid as long as the server that implements the object remains running.

Checking for Globally Scoped Object References

Knowing the scope of an object reference can be useful if you plan to stringify it and save it to a file. Code Sample 5-2 shows a method your client program can use to determine whether an object reference has a local or global scope. The `_is_persistent()` method returns `0` if the object has a local scope and a non-zero value if the scope is global.

CODE
```
class Object {
      ...
      CORBA::Boolean _is_persistent() const;
      ...
};
```

Code Sample 5-2 The method for checking for persistent object implementations.

NOTE *Persistent object references may fail-over to a different object implementation unless the `BindOptions` are set to not allow re-bind operation to occur. See "Enabling Re-Binds" on page 4-8.*

Registering Objects with the BOA

Once a server has instantiated an ORB object that it plans to offer, the BOA must be notified that the object has been initialized. The BOA is notified when the server is ready to receive requests from client programs.

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

If the object passed to the `obj_is_ready()` method represents a globally scoped object, the BOA will register the object with VisiBroker's Smart Agent. If the object is locally scoped, no such registration will occur.

Once a server has instantiated and activated all its objects, the server can invoke the `impl_is_ready()` method or enter another event loop. For information on integrating VisiBroker with other event loops, see Chapter 19, "Event Loop Integration."

NOTE *Invoking the `impl_is_ready()` method will block the caller until the `CORBA::ORB::shutdown()` method is invoked or until the server is terminated.*

ACTIVATING OBJECTS DIRECTLY

In the account example introduced in Chapter 3, "Quick Start for Development with VisiBroker for C++," the `AccountImpl` object was activated directly by the server. Direct activation of an object involves instantiating all the C++ 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 `AccountImpl` objects—one with the object name of Jack B. Quick and the other named Jane Doe. Once the objects have been instantiated and activated, the server invokes `BOA::impl_is_ready()` to prevent the `main` routine from returning.

NOTE *The `BOA::obj_is_ready()` must be called for each object offered by the implementation. If a client attempts to invoke the `_bind()` method on an object before the server has invoked `obj_is_ready()`, a `NO_IMPLEMENT` exception will be raised in the client.*

CODE
```
...
int main(int argc, char* const* argv) {
      try {
          // Initialize the ORB and BOA.
          CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
          CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

          // Create an account object for Jack B. Quick
          AccountImpl quick("Jack B. Quick");
          // Export the newly created Quick account.
          boa->obj_is_ready(&quick);
          cout << quick << " is ready." << endl;

          // Create an account object for Jane Doe
          AccountImpl doe("Jane Doe");
          // Export the newly created Quick account.
          boa->obj_is_ready(&doe);
          cout << doe << " is ready." << endl;

          // Wait for incoming requests
          boa->impl_is_ready();
      }
      ...
}
```

Code Sample 5-3 Server activating two objects directly.

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 references the object. You can register an object implementation with the OAD to provide this automatic activation behavior for your objects.

Object implementations can be registered using a command-line interface (`oadutil`) or programmatically with the `BOA::create()` method. There is also an ORB interface to the OAD, described in "The IDL Interface to the OAD" on page 5-12. In each case, the repository id, object name, the activation policy, and the executable program representing the implementation must be specified.

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

Locating the Implementation Repository Data

Activation information for all object implementations registered with the OAD are stored in the implementation repository. By default, the implementation repository data is stored in a file named impl_rep. This file's path name is dependent on the value of the `VBROKER_ADM` variable. 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 C++ Installation and Administration Guide*.

Registering Objects with oadutil

The `oadutil` 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 to the executable that starts the implementation. If the activation policy is not specified, the shared server policy will be used by default. 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` to register your implementation with the OAD. For more information, see the *VisiBroker for C++ Installation and Administration Guide*.

NOTE *When registering an object implementation, use the same object name that is used when the implementation object is constructed. Only named objects may be registered with the OAD.*

Registering Objects using BOA::create()

In addition to using the `oadutil` command manually or in a script, VisiBroker allows applications written in C++ to use the `BOA::create()` method to register object implementations with the OAD and the Smart Agent. The OAD stores the information in the implementation repository, allowing the object implementation to be located and activated when a client attempts to bind to the object.

CODE
```
class CORBA { ...
      typedef OctetSequence ReferenceData;
      ...
      class BOA {
          virtual CORBA::Object_ptr create(
              const ReferenceData& ref_data,
              CORBA::InterfaceDef_ptr inf_ptr,
              CORBA::ImplementationDef_ptr impl_ptr) = 0;
          ...};
};
```

Code Sample 5-4 The `BOA::create()` method and its parameters.

Distinguishing Between Multiple Instances of an Object

Your implementation can use the `ref_data` parameter to distinguish between multiple instances of the same object. The value of the reference data is chosen by the implementation at object creation time and remains constant during the lifetime of the object. The `ReferenceData` typedef is portable across platforms and ORBs.

NOTE *VisiBroker does not use the `inf_ptr`, which is defined by the CORBA specification to identify the interface of the object being created. Applications created with VisiBroker should always specify a `NULL` value for this parameter.*

Naming Objects Using the ImplementationDef Class

The `impl_ptr` parameter supplies the information that the BOA needs to register an ORB object. The `ImplementationDef` class defines the interface's repository ID, object name, and reference id properties used by the BOA. Code Sample 5-5 shows the methods for querying and setting these properties.

CODE
```
class CORBA {
    ...
    class ImplementationDef
    {
      public:
      ...
          const char *repository_id() const;
          void  repository_id(const char *val);
          const char *object_name() const;
          void  object_name(const char *val);
          CORBA::ReferenceData_ptr id() const;
          void  id(const CORBA::ReferenceData_ptr& data);
          ...
      };
};
```

Code Sample 5-5 The `ImplementationDef` class.

The `repository_id` attribute represents the name specified in the object's IDL specification. The `object_name` property is the name of this object, provided by the implementor or the person installing the object. The `id` property is chosen by the implementation and has no meaning to the BOA or the OAD.

Setting Activation Properties Using the CreationImplDef Class

The `ImplementationDef` class does not supply all the information that the OAD needs to activate an object implementation. The `CreationImplDef` class is derived from `ImplementationDef` and adds the properties the OAD requires to activate an ORB object—`path_name`, `activation_policy`, `args`, and `env`. Methods for setting and querying their values are also provided. Code Sample 5-6 shows the `CreationImplDef` class.

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, discussed in "Object Server Activation Policies" on page 5-2. The `args` and `env` properties represent command line arguments and environment settings for the server.

CODE

```
class CORBA {
    ...
    enum Policy {
      SHARED_SERVER,
      UNSHARED_SERVER,
      SERVER_PER_METHOD
    };
};
class extension {
    ...
    class CreationImplDef: public CORBA::ImplementationDef
    {
      public:
          CreationImplDef(const char *repository_id,
                const char *object_name,
                const CORBA::ReferenceData& id,
                const char *path_name,
                const CORBA::StringSequence& args,
                const CORBA::StringSequence& env);
          static CORBA::CreationImplDef_ptr _narrow(
                CORBA::ImplementationDef_ptr ptr);
          CORBA::Policy activation_policy() const;
          void  activation_policy(CORBA::Policy p);
          const char *path_name() const;
          void  path_name(const char *val);
          CORBA::StringSequence *args() const;
          void  *args(const CORBA::StringSequence& val);
          CORBA::StringSequence *env() const;
          void  env(const CORBA::StringSequence& val);
          ...
        };
    };
```

Code Sample 5-6 The `CreationImplDef` class.

Example of Object Creation and Registration

Code Sample 5-7 shows how to use the `CreationImplDef` class and the `BOA::create()` method to create an ORB object and register it with the OAD. These methods would be used in a separate, administrative process, not in the object implementation itself.

CODE

```
int main(int argc, char* const* argv) {
{
    CORBA::Object_ptr obj;

    // Initialize the ORB and BOA
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::BOA_var boa = orb->BOA_init(argc, argv);

    // Optional reference data
    CORBA::ReferenceData id;

    extension::CreationImplDef impl_def("IDL:Account:1.0", "Jack B. Quick", id,
                "/usr/home/dir/acctsrv",
                NULL /* no args */, NULL /* no envs */);
    obj = boa->create(id, NULL, &impl_def);
    if (! CORBA::is_nil(obj))
          cout << "ORB object created successfully" << endl;
          exit (1);
    }
    ...
```

Code Sample 5-7 Creating an ORB object and registering with the OAD.

NOTE *If the `impl_def` parameter passed to `BOA::create()` cannot be narrowed to a `CreationImplDef` reference, the create will fail and a `CORBA::BAD_PARAM` exception will be raised.*

Dynamically Changing an ORB Implementation

Code Sample 5-8 shows the `BOA::change_implementation()` method which 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.

If the `impl` parameter cannot be narrowed to a `CreationImplDef`, this method will fail and a `CORBA::BAD_PARAM` exception will be raised.

CODE
```
class BOA {
    ...
    virtual void change_implementation(
        const CORBA::Object &obj,
        const CORBA::ImplementationDef& impl);
      ...
};
```

Code Sample 5-8 The `change_implementation()` method.

CAUTION *Although you can change an object's implementation name and object name with the `change_implementation()` method, you should exercise caution. Doing so will prevent client programs from locating the object with the old name.*

Arguments Passed by the OAD

When the OAD starts an object implementation it passes all of the arguments that were specified when the implementation was registered with the OAD.

Un-Registering Objects

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 implementation repository. The object is also removed from the Smart Agent's dictionary. Once an object is unregistered, client programs will no longer be able to locate or use it. In addition, you will be unable to use the `BOA::change_implementation()` method to change the object's implementation. As with the registration process, un-registering may be done either at the command line or programmatically. There is also an ORB object interface to the OAD, described in "The IDL Interface to the OAD" on page 5-12.

Un-Registering Objects Using the oadutil Tool

The `oadutil` tool can be used to un-register an object implementation. You can invoke this tool from the command line or from within a script. If a particular interface name is unregistered, 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 un-register a specific object within an interface. For complete information on using this command, see the *VisiBroker for C++ Installation and Administration Guide*.

Un-Registering Objects Using the BOA::dispose() Method

An object's implementation can use the `BOA::dispose()` method to un-register an ORB object's information from the OAD's implementation repository. In addition, the `BOA:deactivate_obj()` method is automatically invoked to deactivate the object. Any connections that existed between any client program and the object will be terminated.

CODE
```
class CORBA {
      class BOA {
          ...
          virtual void dispose(CORBA::Object_ptr);
          ...
      };
};
```

Code Sample 5-9 The `BOA::dispose()` method.

Displaying the Contents of the Implementation Repository

You can use the `oadutil` tool to list the contents of a particular implementation repository. For each implementation in the repository the `oadutil` tool 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 the `oadutil` command see the *VisiBroker for C++ Installation and Administration Guide.*

THE IDL INTERFACE TO THE OAD

The OAD is implemented as an ORB object, allowing you to create a client program that binds to the OAD and uses its interface to query the status of objects that have been registered. IDL Sample 5-1 shows the IDL interface specification for the OAD.

IDL
```
module Activation
{
    enum state {
      ACTIVE,
      INACTIVE,
      WAITING_FOR_ACTIVATION
    };
```

CONTINUES ON NEXT PAGE

IDL CONTINUED FROM PREVIOUS PAGE

```
struct ObjectStatus {
  long unique_id;
  State activation_state;
  Object objRef;
};
typedef sequence<ObjectStatus> ObjectStatusList;
struct ImplementationStatus {
  extension::CreationImplDef   impl;
  ObjectStatusList  status;
};
typedef sequence<ImplementationStatus> ImplStatusList;

exception DuplicateEntry {};
exception InvalidPath {};
exception NotRegistered {};
exception FailedToExecute {};
exception NotResponding {};
exception IsActive {};
exception Busy {};

interface OAD {
  Object reg_implementation(in extension::CreationImplDef impl)
           raises (DuplicateEntry, InvalidPath);
  extension::CreationImplDef get_implementation(
           in CORBA::RepositoryId repId,
           in string object_name)
           raises ( NotRegistered);
  void change_implementation(in extension::CreationImplDef old_info,
           in extension::CreationImplDef new_info)
           raises (NotRegistered,InvalidPath,IsActive);
  attribute boolean destroy_on_unregister;
  void unreg_implementation(in CORBA::RepositoryId repId,
           in string object_name)
           raises ( NotRegistered );
  void unreg_interface(in CORBA::RepositoryId repId)
           raises ( NotRegistered );
  void unregister_all();
  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 The OAD interface specification.

DEFERRING OBJECT ACTIVATION UNTIL A CLIENT REQUEST

When you design your object implementation, you may want to defer the activation of one or more ORB objects until a client requests them. By deferring object activation, you benefit from shorter server initialization time and more conservative resource initialization. There are two ways to defer object activation:

- Deferring Activation for a Single Object. If you have an object that requires a lot of system resources, you can defer its activation until necessary. An `Activator` can be configured to start a particular object upon client request. In this scenario, the `Activator` masquerades as the object, pretending that the object is activated, until a client request. At this time, the object is actually activated. See "Deferring Activation for a Single Object" on page 5-16.

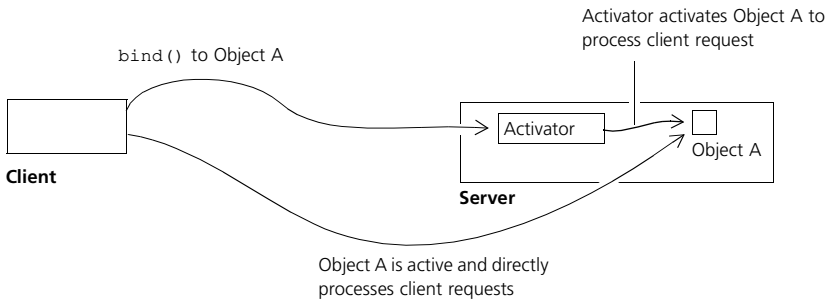

Figure 5-1 Deferring activation for a single object.

- Deferring Activation for a Service. When a server offers a collection of objects that provides clients with a related set of processing, these objects are referred to collectively as a *service* (for example, an object location service). If you have a very large number of objects in a server, you can save system resources by only activating the objects when clients request them. An `Activator` can be configured to start all objects of a certain service upon request. See "Deferring Object Activation Using Service Activators" on page 5-19.

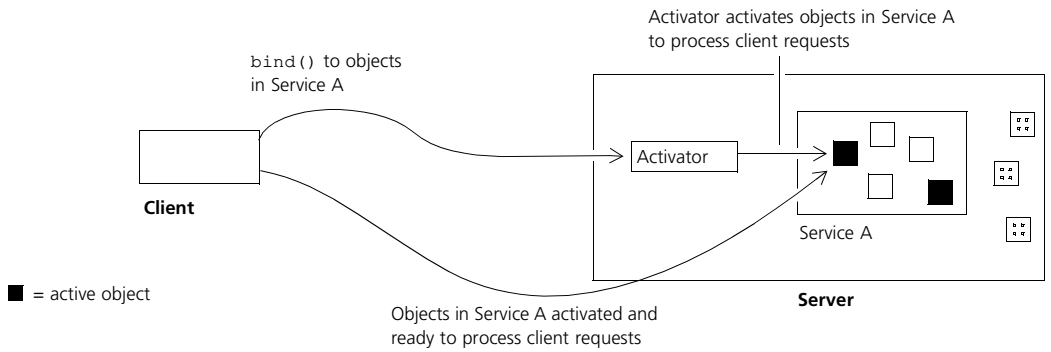

Figure 5-2 Deferring activation for a service.

The Activator Class

Code Sample 5-10 shows the `Activator` class, which provides methods invoked by the BOA to activate and deactivate an ORB object.

CODE
```
class Activator {
      public:
            virtual CORBA::Object_ptr activate(CORBA::ImplementationDef impl)=0;
            virtual void deactivate(Object_ptr, ImplementationDef_ptr impl);
};
```

Code Sample 5-10 The `Activator` class.

Deriving your own class from the `Activator` class lets you override the `activate()` and `deactivate()` methods that the ORB will use for the `AccountImpl` object. This lets you delay the instantiation of the `AccountImpl` object until the BOA receives a request for that object. It also lets you provide clean-up processing when the BOA deactivates the object. Code Sample 5-11 shows how to create an `Activator` for the `AccountImpl` class.

CODE
```
class extension { ...
    class AccountImplActivator : public extension::Activator {
        public:
            virtual CORBA::Object_ptr activate(
                CORBA::ImplementationDef_ptr impl);
            virtual void deactivate(CORBA::Object_ptr,
                CORBA::ImplementationDef_ptr impl);
    };
```

CONTINUES ON NEXT PAGE

CODE CONTINUED FROM PREVIOUS PAGE

```
    CORBA::Object_ptr AccountImplActivator::activate(
            CORBA::ImplementationDef_ptr impl) {
      // When the BOA needs to activate us, instantiate the AccountImpl object.
    return CORBA::Object::_duplicate(new AccountImpl(impl->object_name()));
    }
    void AccountImplActivator::deactivate(CORBA::Object_ptr obj,
            CORBA::ImplementationDef_ptr impl) {
      // When the BOA deactivates us, release the Account object.
      obj->release();
    }
}
```

Code Sample 5-11 Deriving an `AccountImplActivator` class, implementing the `activate()` and `deactivate()` methods.

Deferring Activation for a Single Object

To defer activation for a single object, the `BOA::obj_is_ready()` and `BOA::impl_is_ready()` methods may be used with the `ActivationImplDef` class to instantiate objects upon receipt of a client request.

CODE

```
class CORBA {
      class BOA {
          ...
          virtual void obj_is_ready(Object_ptr,
                ImplementationDef_ptr impl=NULL) = 0;
          virtual void impl_is_ready(ImplementationDef_ptr impl=NULL) = 0;
          ...
      };
};
```

Code Sample 5-12 The `BOA::obj_is_ready()` and `BOA::impl_is_ready()` methods.

In previous examples, the `obj_is_ready()` method was only passed an object reference. The `impl_is_ready()` method was invoked with no parameters at all. It is possible to pass an `ActivationImplDef` pointer to the `obj_is_ready()` method, which can be used to override the activation and deactivation methods used by the BOA. Code Sample 5-13 on page 5–17 shows the `ActivationImplDef` class, which specifies an `Activator` reference and provides methods for setting and retrieving that reference. Note that this class is derived from `ImplementationDef`.

CODE
```
class extension {
    ...
    class ActivationImplDef: public CORBA::ImplementationDef{
      public:
            ActivationImplDef(const char *repository_id,
                const char *object_name,
                const ReferenceData& id,
                Activator_ptr act);
            Activator_ptr activator_obj();
            void activator_obj(Activator_ptr val);
            ...
    };
};
```

Code Sample 5-13 The `ActivationImplDef` class for deferring activation of a single object.

Example of Deferred Object Activation for a Single Object

Code Sample 5-14 shows how to use the `ActivationImplDef` class, along with the `AccountImplActivator` class, to defer the activation of the `AccountImpl` object until a client request is received. The instantiation of the `AccountImpl` object no longer appears in the `main` routine. Instead, the `AccountImpl` object will be instantiated when the BOA receives a client request and invokes the `activate()` method.

In this example, the invocation of `BOA::obj_is_ready()` is passed a `NULL` object pointer as well as an `ActivationImplDef` reference. The creation of the `AccountImpl` object named Jack B. Quick will now be deferred until the first client request for that object is received.

CODE
```
void main(int argc, char * const * argv)
{
    // Initialize ORB and Basic Object Adaptor (BOA)
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::BOA_var boa = orb->BOA_init(argc, argv);

    CORBA::ReferenceData id;
    extension::ActivationImplDef impl("IDL:Account:1.0", "Jack B. Quick", id,
            (extension::Activator_ptr) new AccountImplActivator);

    // obj_is_ready is passed an ActivationImplDef object to override
    // the activation of the AccountImpl object.
    boa->obj_is_ready(CORBA::Object::_nil(), &impl);

    // activate other objects
    ...
```

CONTINUES ON NEXT PAGE

CODE CONTINUED FROM PREVIOUS PAGE

```
    // Begin event loop of receiving requests
    boa->impl_is_ready();
    return(1);
};
```

Code Sample 5-14 Using the `ActivationImplDef` class with the `BOA::obj_is_ready()` method.

If an implementation has only one object, the separate steps of invoking the `obj_is_ready()` method followed by the `impl_is_ready()` method can be combined into a single invocation of the `impl_is_ready()` method. This will invoke the `obj_is_ready()` method on the `ActivationImplDef` object and then enter the blocking event loop.

Code Sample 5-15 shows this use of the single invocation of the `impl_is_ready()` method with an `ActivationImplDef` object. Since the `impl_is_ready()` method can accept only one object's implementation, this approach cannot be used if multiple objects reside in the same implementation.

NOTE *If the second parameter to the `impl_is_ready()` method is set to 1, the method will block. If the second parameter is set to 0, the method will not block.*

CODE

```
void main(int argc, char * const * argv)
{
    // Initialize ORB and Basic Object Adaptor (BOA)
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::BOA_var boa = orb->BOA_init(argc, argv);

    CORBA::ReferenceData id;
    extension::ActivationImplDef impl("IDL:Account:1.0", "Jack B. Quick", id,
            (extension::Activator_ptr) new AccountImplActivator);

    // impl_is_ready is passed an ActivationImplDef object to override
    // the activation of the AccountImpl object.
    boa->impl_is_ready(&impl, 1);

    return(1);
};
```

Code Sample 5-15 The use of a single `impl_is_ready()` method.

Deferring Object Activation Using Service Activators

The use of an `Activator` and the `ActivationImplDef` class as described in "Deferring Activation for a Single Object" on page 5-16 allows the deferred activation of a single object implementation until a client requests that implementation. You can also defer the activation of multiple object implementations with a single `Activator`—doing so is called service activation.

Service activation can be used when a server needs to provide implementations for a large number of objects (commonly thousands of objects) 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, it writes the object's state to the database and releases the implementation.

Assuming the objects that will make up the service have already been created, 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. Provide implementations for the interface which are service objects, rather than persistent objects. This is done when the object constructs itself as an activable part of a service.
3. Implement the `Activator` which creates the object implementations on demand. In the implementation, you derive an Activator class from `CORBA_Activator`, overriding the `activate()` and `deactivate()` methods.
4. Register the service name and the Activator class with the BOA.

Example of Deferred Object Activation for a Service

The following sections describe the odb example for service activation which is located in the examples directory of your VisiBroker installation. The examples directory contains the following files:

NAME	DESCRIPTION
`db.idl`	IDL for DB and DBObject interfaces.
`server.C` or `server.cpp`	Creates objects using service activators, returns IORs for the objects, and deactivates the objects.
`server_r.C` or `server_r.cpp`	Multithreaded server program.
`creator.C` or `creator.cpp`	Calls the DB interface to create 100 objects and stores the resulting stringified object references in a file (objref.out).
`client.C` or `client.cpp`	Reads the stringified object references to the objects from a file and makes calls on them, causing the activators in the server to create the objects.
`Makefile`	When `make` or `nmake` (on Windows) is invoked in the `odb` subdirectory, builds the following client and server programs: ■ `server` or `server.exe` ■ `creator` or `creator.exe` ■ `client` or `client.exe`

Table 5-1 Files in the odb example for service activation.

The odb example shows how an arbitrary number of objects can be created by a single service. The service alone is registered with the BOA, instead of each individual object, with the reference data for each object stored as part of the IOR. This facilitates object-oriented database (OODB) integration, since you can store object keys as part of an object reference. When a client calls for an object that has not yet been created, the BOA calls a user-defined `Activator`. The application can then load the appropriate object from persistent storage.

In this example, an `Activator` is created that is responsible for activating and deactivating objects for the service named "DBService". References to objects created by this `Activator` contain enough information for the ORB to relocate the `Activator` for the DBService service, and for the `Activator` to recreate these objects on demand.

The DBService service is responsible for objects that implement the `DBObject` interface. An interface (contained in `db.idl`) is provided to enable manual creation of these objects.

THE DB.IDL INTERFACE

The `db.idl` interface enables manual creation of objects that implement the `DBObject` interface.

IDL
```
interface DBObject {
    string get_name();
};

typedef sequence<DBObject> DBObjectSequence;

interface DB {
    DBObject create_object(in string name);
};
```

IDL Sample 5-2 The `db.idl` interface.

The `DBObject` interface represents an object created by the `DB` interface, and can be treated as a service object.

`DBObjectSequence` is a sequence of `DBObject`s. The server uses this sequence to keep track of currently active objects.

The `DB` interface creates one or more `DBObject`s using the `create_object` operation. The objects created by the `DB` interface can be grouped together as a service.

IMPLEMENTING A SERVICE-ACTIVATED OBJECT

The idl2cpp compiler generates two kinds of constructors for the skeleton class `_sk_DBObject` from db.idl. The first constructor is for use by manually-instantiated objects; the second constructor enables an object to become part of a service. As shown in Code Sample 5-16 below, the implementation of `DBObject` constructs its base `_sk_DBObject()` method using the service constructor, rather than the `object_name` constructor typically used for manually-instantiated objects. By invoking this type of constructor, the `DBObject` constructs itself as a part of a service called `DBService`.

CODE
```
class DBObjectImpl: public _sk_DBObject
{
    private:
        CORBA::String_var_name;
```

CONTINUES ON NEXT PAGE

CODE CONTINUED FROM PREVIOUS PAGE

```
    public:
      DBObjectImpl(const char *nm, const CORBA::ReferenceData& data)
      : _sk_DBObject("DBService", data), _name(nm) {}
    ...
};
```

Code Sample 5-16 Example of implementing a service-activated object.

Note that the base constructor requires a service name as well as an opaque `CORBA::ReferenceData` value—the `Activator` uses these parameters to uniquely identify this object when it must be activated due to client requests. The reference data used to distinguish among multiple instances in this example consists of the range of numbers from 0 to 99.

IMPLEMENTING A SERVICE ACTIVATOR

Normally, an object is activated when a server instantiates the C++ classes implementing the object, and then calls `BOA::obj_is_ready` followed by `BOA::impl_is_ready`. To defer activation of objects, it is necessary to gain control of the `activate()` method that the BOA invokes during object activation. You obtain this control by deriving a new class from `CORBA_Activator` and overriding the `activate()` method, using the overridden `activate()` method to instantiate C++ classes specific to the object.

In the odb example, the `DBActivator` class derives from `CORBA_Activator`, and overrides the `activate()` and `deactivate()` methods. The `DBObject` is constructed in the `activate()` method.

CODE

```
class DBActivator: public CORBA_Activator
{
    virtual CORBA::Object_ptr activate(CORBA::ImplementationDef_ptr impl);
    virtual void deactivate(CORBA::Object_ptr,
        CORBA::ImplementationDef_ptr impl );
public:
    DBActivator(CORBA::BOA_ptr boa) : _boa(boa) {}

private:
    CORBA::BOA_ptr _boa;
```

Code Sample 5-17 `};` *Example of overriding* `activate()` *and* `deactivate()`.

As shown in Code Sample 5-18 on page 5–23, the `DB_Activator` class creates an operation based on its `CORBA::ReferenceData` parameter. When the BOA receives a client request for an object under the responsibility of the `Activator`, the BOA invokes the `activate()`

method on the `Activator`. When calling this method, the BOA uniquely identifies the activated object implementation by passing the `Activator` an `ImplementationDef` parameter—from which the implementation can obtain `CORBA::ReferenceData`, the requested object's unique identifier.

CODE
```
CORBA::Object_ptr DBActivator::activate(CORBA::ImplementationDef_ptr impl)
{
    CORBA::ReferenceData_var id(impl->id());
    cout << "Activate called for object=[" << (char*) id->data()
              << "]" << endl;
         DBObjectImpl *obj = new DBObjectImpl((char *)id->data(), id);
         _impls.length(_impls.length() + 1);
    _impls[_impls.length()-1] =  DBObject::_duplicate(obj);
    _boa->obj_is_ready(obj);
    return obj;
}
```

Code Sample 5-18 };Example of implementing a service activator.

INSTANTIATING THE SERVICE ACTIVATOR

As shown in Code Sample 5-19 below, the `DBActivator` service activator is created and registered with the BOA using the `BOA::impl_is_ready()` call in the main server program. The `DBActivator` service activator is responsible for all objects that belong to the DBService service. All requests for objects of the DBService service are directed through the `DBActivator` service activator. All objects activated by this service activator have references that inform the ORB that they belong to the DBService service.

CODE
```
int main(int argc, char **argv)
{
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
    CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

    MyDB db("Database Manager");
    boa->obj_is_ready(&db);

         DBObjectImplReaper reaper;
         reaper.start();

         cout << "Server is ready to receive requests" << endl;

    boa->impl_is_ready("DBService", new DBActivator(boa));
    return(0);
}
```

Code Sample 5-19 Example of instantiating the service activator.

Note that the call to `BOA::impl_is ready()` is a variation on the usual call to `BOA::impl_is_ready()`—it takes two arguments:

- a service name.
- an instance of an Activator class that will be used by the BOA to activate objects belonging to the service.

USING A SERVICE ACTIVATOR TO ACTIVATE AN OBJECT

Whenever an object is constructed, `BOA::obj_is_ready()` must be explicitly invoked in `DBActivator::activate()`. There are two calls to `BOA::obj_is_ready()` in the server program. One call occurs when the server creates a service object and returns an IOR to the creator program.

CODE
```
DBObject_ptr create_object(const char *name) {
      char ref_data[100];
      memset(ref_data,'\0',100);
      sprintf(ref_data, "%s", name);
      CORBA::ReferenceData id(100, 100, (CORBA::Octet *)ref_data);
      DBObjectImpl *obj = new DBObjectImpl(name, id);
      _boa()->obj_is_ready(obj);
                  _impls.length(_impls.length() + 1);
      _impls[_impls.length()-1] =  DBObject::_duplicate(obj);
      return obj;
    }
```

Code Sample 5-20 First server call to `BOA::obj_is_ready`.

The second occurrence of `BOA::obj_is_ready()` is in `DBActivator::activate()`, and this needs to be explicitly called. Refer to Code Sample 5-18 on page 5–23 to see this second call in context.

DEACTIVATING OBJECTS AND IMPLEMENTATIONS

The correct approach to deactivating objects depends on how the instance is activated. Instances can be activated manually through C++ instantiation, by the OAD or by the BOA.

Exiting a Server Process

A server that is started manually can be considered deactivated when the server exits. A server may use the `CORBA::ORB::shutdown()` method to cause a previously invoked `BOA::impl_is_ready()` method to unblock. Once this method is invoked, the BOA will prevent any new remote operations from being invoked on any objects contained by the

server. Once all existing operation requests have returned, the original blocking invocation of the `BOA::impl_is_ready()` method will return and the server's main program can safely exit.

CAUTION *You cannot invoke the `BOA::impl_is_ready()` method after the `CORBA::ORB::shutdown()` method has been invoked.*

Deactivating Objects Registered with the BOA

The `BOA::deactivate_obj()` method is provided to deactivate objects that were activated by the BOA. After this method is invoked, the object will be removed from the Smart Agent's list of objects offered by that implementation. Code Sample 5-21 shows the definition of the `deactivate_obj()` method.

CODE
```
class CORBA {
      class BOA {
          ...
          virtual void deactivate_obj(Object_ptr);
          ...
      };
};
```

Code Sample 5-21 The `BOA::deactivate_obj()` method.

CAUTION *You should not invoke `delete()` on an object since other references to the object may still exist. Instead, invoke `BOA:deactivate_obj()` method and then the `release()` method.*

Deactivating an Instance of an Object

In addition, an object that was created by instantiating its C++ class, will be unregistered when it is destroyed. You can invoke the `CORBA::release()` method to indicate that you no longer wish to hold your reference to the instance. This method will invoke the object's destructor if no other references to the object exist, at which time it will be unregistered from the list of objects maintained by the Smart Agent.

Deactivating Implementations Started by the OAD

Implementations started by the OAD can be deactivated by using the `BOA::deactivate_impl()` method. Once this method is invoked, the implementation becomes unavailable to service client requests. The implementation can only be

re-activated if it is re-registered with the OAD as described in "Automatically Activating Servers with the Object Activation Daemon" on page 5-6.

Deactivating Service-Activated Object Implementations

The main use for service activation is to provide the illusion that a large number of objects are active within a server, but to have only a small number of these objects actually active at any given moment. To support this model, the server must be able to temporarily remove objects from use. The multithreaded `DBActivator` example program contains a reaper thread that deactivates all `DBObjectImpls` every 30 seconds. The `DBActivator` simply releases the object reference when the `deactivate()` method is invoked. If a new client request arrives for a deactivated object, the ORB informs the `Activator` that the object should be re-activated.

CODE

```
// static sequence of currently active Implementations
static VISMutex          _implMtx;
static DBObjectSequence _impls;

// updated DBActivator to store activated implementations
// in the global sequence.
class DBActivator: public CORBA_Activator {
    virtual CORBA::Object_ptr activate(CORBA::ImplementationDef_ptr impl) {
        CORBA::ReferenceData_var id(impl->id());
        DBObjectImpl *obj = new DBObjectImpl((char *)id->data(), id);
        VISMutex_var lock(_implMtx);
        _impls.length(_impls.length() + 1);
        _impls[_impls.length()-1] =  DBObject::_duplicate(obj);
        return obj;
    }
    virtual void deactivate(CORBA::Object_ptr,
                CORBA::ImplementationDef_ptr impl) {
          obj->_release();
        }
};
// Multi-threaded Reaper for destroying all activated
// objects every 30 seconds.
class DBObjectImplReaper : public VISThread {
public:
     // Reaper methods
    virtual void start() {
      run();
    }
    virtual CORBA::Boolean startTimer() {
      vsleep(30);
      return 1;
```

CONTINUES ON NEXT PAGE}

CODE CONTINUED FROM PREVIOUS PAGE

```
    virtual void begin() {
        while (startTimer()) {
            doOneReaping();
        }
    }
protected:
    virtual void doOneReaping() {
        VISMutex_var lock(_implMtx);
        for (CORBA::ULong i=0; i < _impls.length(); i++) {
            // assigning nil into each element will release
            // the reference stored in the _var
            _impls[i] = DBObject::_nil();
        }
        _impls.length(0);
      }
};
```

Code Sample 5-22 Example of deactivating service-activated object implementations.

Note that sections of code where multiple threads could be accessing the _impl data structure are guarded by using `VISMutex` to provide mutual exclusion.

PART II

ADVANCED VISIBROKER DEVELOPMENT

This part of the guide includes these chapters:

Chapter 6	Handling Exceptions	6-1
Chapter 7	Managing Threads and Connections	7-1
Chapter 8	Using the IDL to C++ Compiler	8-1
Chapter 9	Parameter Passing Rules for the IDL to C++ Mapping	9-1
Chapter 10	The Smart Agent Architecture	10-1
Chapter 11	Using the Tie Mechanism: An Alternative to Inheritance	11-1
Chapter 12	Using Interface Repositories	12-1
Chapter 13	Using the Dynamic Invocation Interface	13-1
Chapter 14	Dynamically Creating Object Implementations	14-1
Chapter 15	Instrumenting and Modifying the ORB with Interceptors	15-1
Chapter 16	Customizing Remote Object Invocations Using Smart Stubs	16-1
Chapter 17	Enabling Object Persistence Using the Object Database Activator	17-1

Chapter 18	Discovering Object Instances Using the Location Service	18-1
Chapter 19	Event Loop Integration	19-1
Chapter 20	Dynamically Managed Types	20-1
Chapter 21	Using Object Wrappers	21-1
Chapter 22	Using the ORB Management Interface	22-1