CHAPTER 6

Object Location

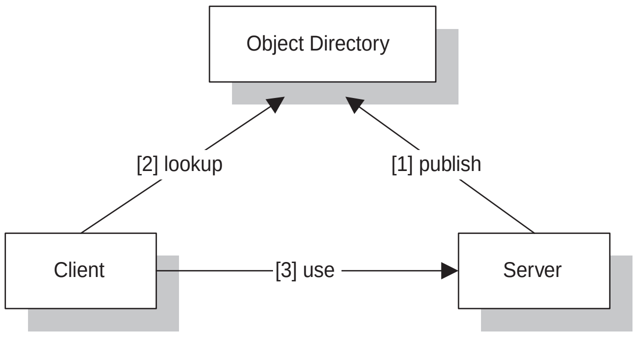
Every CORBA system must, somehow, answer the same question: how do client components obtain object references? There are many possible ways to answer this question, and each approach has its own set of strengths and weaknesses. Solutions can be simple to use but less flexible and scalable, or more powerful but more complex. Likewise, solutions can be proprietary to a specific ORB implementation or CORBA-compliant. The ideal approach would allow a client to easily obtain a reference to an arbitrary object on an arbitrary host, be CORBA-compliant, and be flexible enough to scale up to large CORBA systems.

In this chapter, we first introduce a model of how clients obtain references to objects in servers. Then, we discuss the CORBA-compliant approaches to obtaining object references, as well as touching on some ORB-specific mechanisms. Each of these mechanisms has its own set of strengths and weaknesses, and is best applied in different situations.

A Model for Locating Objects

There are three steps involved in making objects available to clients, as shown in Figure 6.1. This abstract model sets the stage for our discussion on the practical mechanisms that can be used to locate objects.

Figure 6.1 Exporting and locating objects



This model is made up of several pieces. *Server* and *client* are our familiar application components, which implement and use our business objects, respectively. The third component is the *object directory*. This piece is responsible for storing object references, along with some descriptive data that can optionally be associated with the reference. For example, the CORBA naming service is an object directory which stores object references, associating them with a name. Likewise, the CORBA trading object service is an object directory which stores object references, associating them with a set of properties. Even something as simple as an object reference string stored in a configuration file follows this model.

To locate objects, our systems perform the following steps. First, the server publishes a number of objects to the directory, providing some attributes that identify the object in a meaningful way. Next, clients look up objects in the directory. Clients provide a set of desired attributes to the directory so that it can return a set of matching objects. Once a client has obtained one or more objects from the directory, it can begin using them.

The details of how a server publishes objects, and how clients look up objects is, of course, specific to a particular object directory. Likewise, the number and type of the attributes associated with an object is specific to each concrete object directory implementation.

What is an Object Reference?

What exactly is it that these directories are storing? Our servers are not actually publishing objects to the directory, rather they are publishing *object references*. A reference contains the IDL type of the object, as well as enough information for an ORB to be able to find the target object in its server process on its host. When a client obtains an object reference from the directory, the ORB turns this into a local programming language object—a proxy—which the client application code uses to make invocations on the remote target object.

CORBA Object Location Services

The CORBA specification introduces several instances of object directories. The CORBA naming service and the CORBA trading object service are the most commonly used, and provide different levels of flexibility and complexity for object publication and lookup.

The CORBA naming service stores a name with each object reference. The naming service also provides for a hierarchical naming space structure, which allows us to logically organize our objects in whatever way makes sense for our business domain.

With the CORBA trading object service, each object (known as an *offer* in trader terminology) can have multiple properties of any type. The trading service provides a flexible mechanism for clients to look up objects based on any subset of these properties.

Both of these services provide us with the ability to associate some additional information with our published object references. The ability to attach application-specific information to an object reference is the primary value of these services. It is a layer of abstraction which allows us to locate objects based on information that is important to us, rather than information that is important to the ORB.

The CORBA Naming Service

The CORBA naming service is a simple example of an object directory. It stores objects references in a hierarchical structure much like a Unix-style file system, so many of the concepts are familiar to us. Each object reference has a *name* associated with it. A name consists of two string fields, *id* and *kind*. Conceptually, these correspond to a file-system filename and extension. The hierarchy is made up of *naming contexts*, which can contain object references as well as other naming contexts. In this sense, they correspond to directories in a file system. A naming context can store multiple object references, which must be differentiated by either the `id` or `kind` fields of the name structure. A simple naming service hierarchy is shown in Figure 6.2.

Our root naming context object contains two elements, both of which are naming contexts. Each of these elements has a name associated with it; one is “StockWatch,” and the other is “PortfolioManager.” (In this example, we only use the `id` part of the name, and don’t use the `kind` field). The “StockWatch” naming context contains two elements, “NASDAQ” and “NYSE,” both of which are application object references that our server has published.

Notice that the naming contexts shown in Figure 6.2 have the T-bar notation, which indicates that they are CORBA objects. This is, in fact, how our application components use the naming service—they make invocations on naming context objects. Naming contexts support a number of operations, only two of which are important for this discussion. Pictorially, this is shown in Figure 6.3. The IDL for these methods is shown below. (For a full introduction to the naming service IDL, see the CORBA specification, or your naming service programmer’s guide).

Figure 6.2 Sample Naming Service Hierarchy

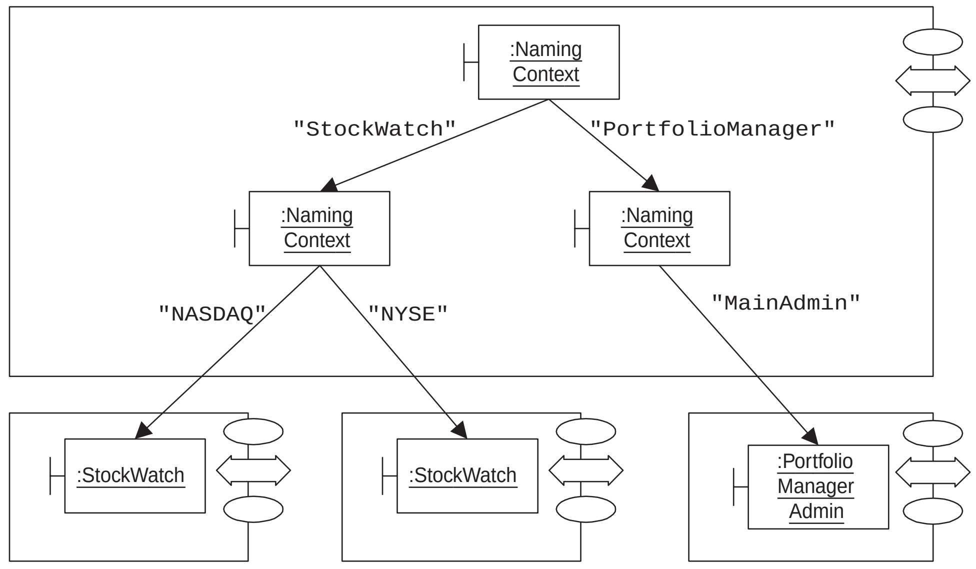
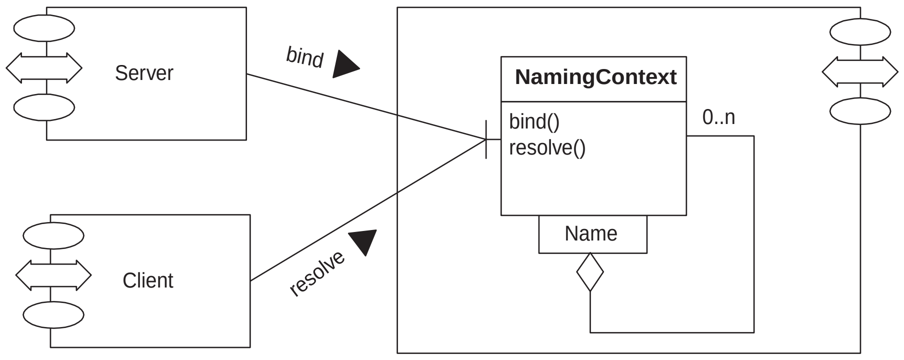


Figure 6.3 Naming Service Class Diagram



```
// IDL Fragment : CORBA Naming Service
// Simplified IDL : exceptions and typedefs omitted
interface NamingContext
{
    void bind(in Name n, in Object obj);

    Object resolve (in Name n);

// other methods not shown...
}
```

As required by our model, naming contexts support operations that allow our servers to publish objects, and allow our clients to look up objects. These two methods are discussed next.

Servers: Bind Objects

Our servers use the `bind()` method to publish an object. The server invokes this method on a naming context object, and supplies the `Name` structure associated with the object, as well as the object itself. Note that we are actually associating two pieces of identifying information with each object that we export. First is the name itself, made up of the id and kind fields as mentioned previously. The second piece of information is the object's place in the naming service hierarchy. It is these two pieces of information that we can use to control how our name space is structured. This will be covered shortly, in "Designing a Name Hierarchy."

Notice that our application object is passed to the `bind()` method as an `Object`. This generic type permits the naming service to be used to store any application object. (Recall that `Object` is an implicit base class for all CORBA objects defined in IDL.)

Clients: Resolve Objects

Clients look up objects by invoking the `resolve()` method on a particular naming context object. The client provides the naming service with enough information to uniquely identify an object. The desired name is passed in as an input parameter, while the naming context on which this method is invoked determines the place in the hierarchy from which the service will perform the lookup. If a matching object is found, it is returned as a generic `Object`. The client application simply narrows this to the appropriate application object, then uses it.

Designing a Name Hierarchy

In making use of the naming service, the first thing to focus on is the structure of the hierarchy itself. We can make it as deep or wide as necessary, depending on our business requirements. We can also choose how to use the `id` and `kind` fields of our entries in the naming service. Applying these two aspects lets us arrange our hierarchy in any number of ways. Generally, we will use the hierarchical structure to separate logically distinct objects, while related objects will be grouped together and distinguished by their unique names.

Let's briefly explore an example. Imagine that each StockWatch server application implements two interfaces. One is our familiar `StockWatch` interface, which implements our business logic. The other is `ServerManager`, which is a management and instrumentation interface. This is used by a system administration application to observe and control the server processes as they are running.

One possible structure for our name hierarchy is shown in Figure 6.4. In order to simplify our name hierarchy diagrams, we will avoid using the formal notation. Instead, we'll use the simpler format shown below. Each context and object is simply denoted by its name, in an `id.kind` string format. The leaf nodes of the hierarchy are the object references.

Here, we have chosen to have a flat hierarchy, with two object references with `id` *NASDAQ* in the *StockWatch* naming context. These references are differentiated by their `kind` field. Likewise, there are two *NYSE* references in the naming context, one reference for each of the interfaces supported by our server process.

Alternatively, we could have chosen not to use the `kind` field at all. In this case, we would have a deeper name hierarchy, with a naming context for each of the stock exchanges. Each of these contexts would contain the two exported objects. This is shown in Figure 6.5.

Figure 6.4 Flat Name Hierarchy

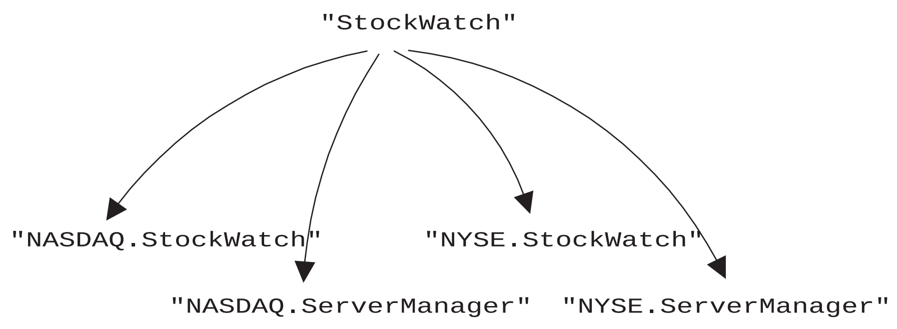
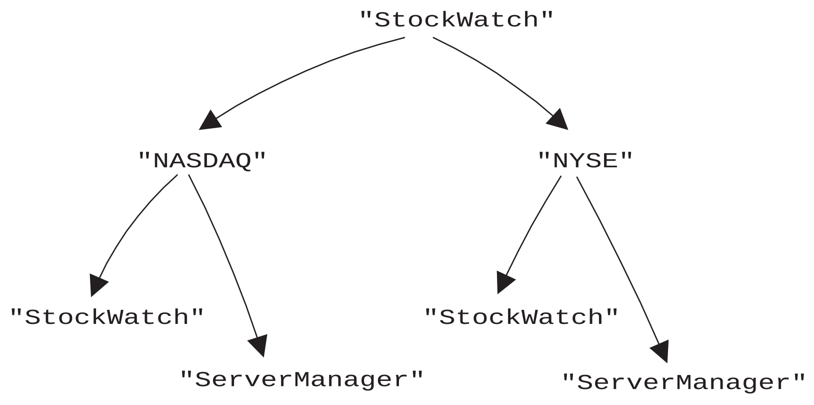


Figure 6.5 Deep name hierarchy



When designing a name hierarchy, there are a number of factors to be considered. First, consider the complexity of your applications, and the domain that this name hierarchy will cover. Obviously, a hierarchy for a single departmental application can be much simpler than one for a collection of applications spanning the enterprise. Also think about the intended usage of the hierarchy. Is the structure of the hierarchy going to be exposed to end users through an application? Or, will the hierarchy only be accessed by the application program and administrators? This factor will often determine whether a hierarchy is made up of many contexts with descriptive names, or fewer contexts with names in a standard format.

For instance, two potential hierarchies are shown in Figures 6.6 and 6.7. The first is well-suited for browsing by end users. We could easily imagine an application that browses the hierarchy and lets users choose the printer they wish to use. Compare this to the second hierarchy, which is much more compact. This structure is better used for applications that hide the structure from end users. The application may be configured once by an administrator to use a particular printer, so that these names are not visible to end users. The structure in Figure 6.7 is simpler, so that it can be used by simpler administration and application code.

Another factor to consider is the number of objects published by server applications. In general, there are two types of objects that we can choose to export. First, we could export the actual business objects that our applications use. The printer objects shown in Figure 6.6 are examples of this. Second, we could choose not to publish our business objects, but to instead publish factory objects. These factories acts as entry points into our components. They support methods that applications use to obtain the business objects. The `PortfolioManager` object is a good example of an entry point. Our client applications will look up a `PortfolioManager` object, and use this to obtain a `Portfolio` object, which implements our business functions. Later, in Chapter 18, “Consequences For The Engineering Process,” we discuss CORBA components, and entry points into their services.

Figure 6.6 Descriptive naming hierarchy

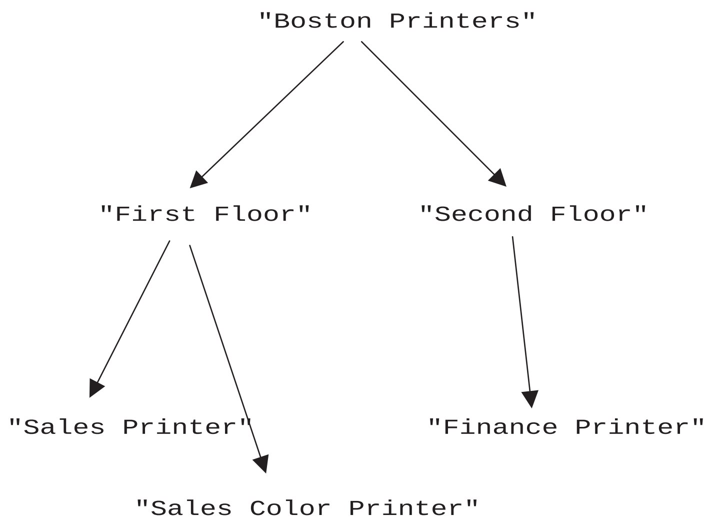
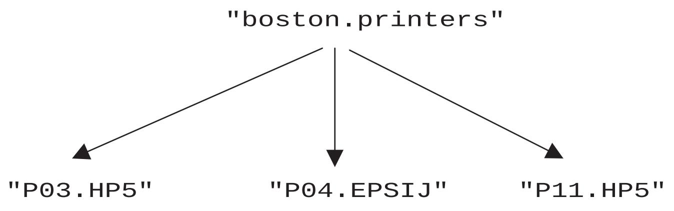


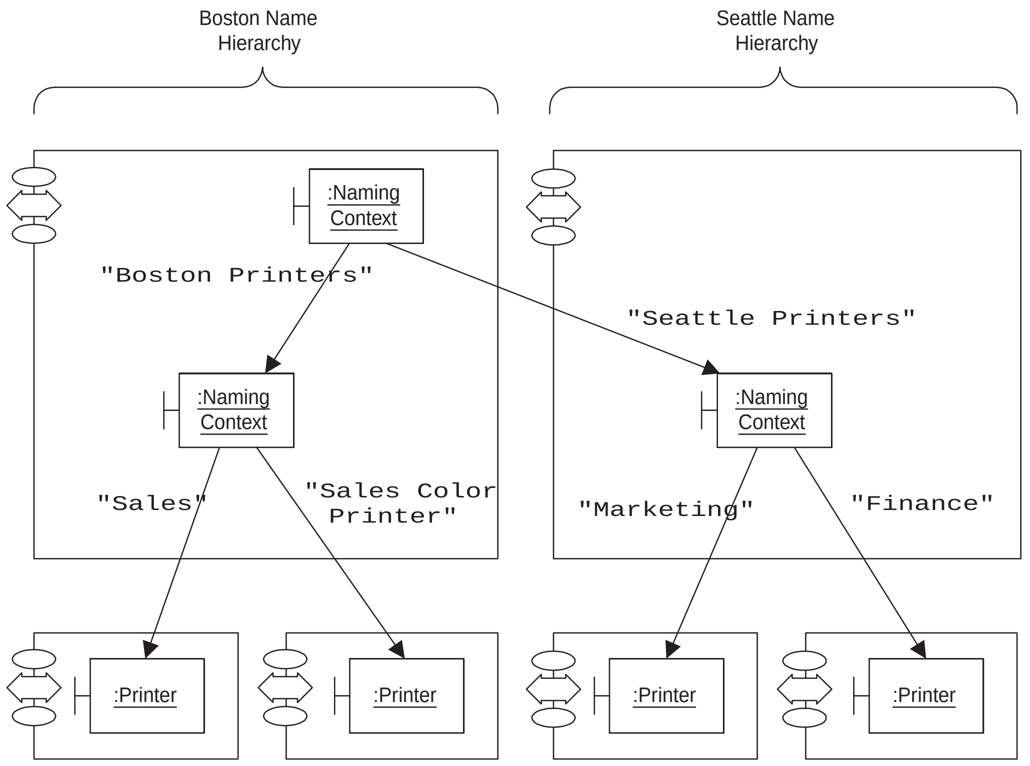
Figure 6.7 Compact naming hierarchy



One additional factor to be considered is the quality of service required by a naming structure. This is especially important if you intend to use the structure to store large numbers of objects, or very complex hierarchies. Once you have designed your hierarchy, evaluate the capabilities of your naming service. Consider the persistent storage mechanisms supported—does it use a file-based system, or is it connected to an industrial-strength database? Evaluate its performance with the intended numbers of contexts and objects. Does it meet your performance requirements? Consider its robustness—does it support replication of its data store, for instance? Make sure that the product you choose meets your needs.

When designing a name hierarchy, also consider the naming service's ability to federate (that is, link) name hierarchies together. Recall that naming contexts are simply objects, implemented in a particular CORBA server. When you insert a naming context into a hierarchy, you actually supply

Figure 6.8 Federated name hierarchies



a reference to the new naming context. Typically, this new context is implemented in the same server as the containing context. However, because these are simply CORBA object references, the naming context can in fact be implemented in another naming service process. An example of federation is shown in Figure 6.8.

This example shows our Boston naming hierarchy linked to another hierarchy in our Seattle office. These hierarchies are stored by two separate naming service implementations, running on separate hosts. By federating them, we can provide transparent access to multiple hierarchies. In this example, our Boston-based end users can easily use a printer in the Seattle office, by simply navigating to the naming context named “Seattle Printers.”

Federating naming hierarchies allows us to provide global access to objects, while still maintaining local control of these objects. As long as these two offices agree on the structure and format of the namespace, we’ll be able to easily and transparently share services across the organization.

The CORBA Trading Object Service

The CORBA trading object service provides us with a powerful, flexible means of publishing and looking up our objects. When our servers publish objects, they associate any number of properties of any type with the object. When clients perform a lookup, they specify a set of desired properties. The trader evaluates this lookup query, and returns a set of matching objects.

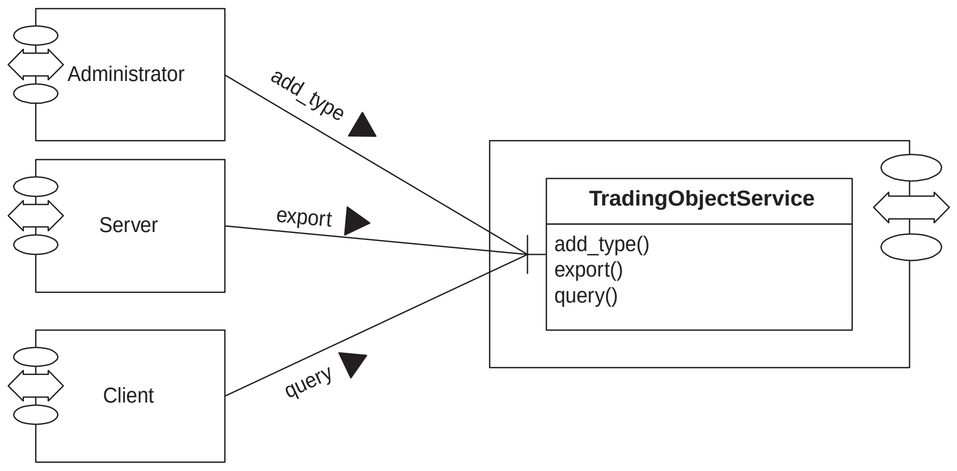
Unlike a name hierarchy, the trader's object directory is not structured in any formal way. Instead, the trading service is based on the concept of a *service type*. A service type contains an IDL interface identifier, as well as some additional data defining the attributes that can be associated with this type. Before our servers can actually publish any objects, we must properly define the number and type of attributes that can be associated with objects of this type. Once we have defined a service type, our servers publish objects of this type. In trader terminology, this is known as *exporting a service offer*. Servers export an object reference, along with attributes that identify this particular service offer. These offers must follow the format described by a previously defined service type. The collection of offers exported by servers to a trader makes up its *trading space.*

Clients look up objects in the trading space by performing a *query*. Clients specify a service type, as well as a set of desired properties and constraints. The trading service evaluates the supplied criteria and returns an ordered set of matching objects.

The trading object service IDL is complex, and we do not fully explain it here. Instead, we will work with an abstraction of the IDL, which is simpler to explain and conceptually provides the same set of operations. For a fully detailed explanation, see the CORBA specification or your trading object service programming guide.

Figure 6.9 illustrates our application components' use of the trading service. The simplified IDL shown below contains the essential arguments for each of the methods implemented by the trader.

Figure 6.9 Trading Object Service Diagram



```
// pseudo-IDL
interface TradingService
{
    // define a new service type
    serviceTypeID add_type(in string serviceTypeName,
                           in string IDLInterfaceIdentifier,
                           in propertyDefinitionSeq definedProperties);
    // publish an offer to the trader
    offerId export(in Object reference,
                   in string serviceTypeName,
                   in propertySeq properties);

    // look up one or more matching offers
    void query (in string serviceTypeName,
                in string constraint,
                in specifiedProperties propertiesToReturn,
                out offerSeq offers);
};
```

Administrator: Define Service Types

When using the trading service, the first step is to define the service types that will be supported. Each service type corresponds to a single IDL interface[1] and defines the set of attributes that can be used to describe this service offer. Let's examine a simple example. This uses the simple service type notation from the OMG CORBA trading object service specification. (Note that this is not IDL, but is just a convenient notation for specifying service types.)

```
service PrinterService {
    interface Printer;
    mandatory property string           location;
    mandatory property sequence<string>      supportedFormats;
    mandatory property long                  pagesPerMinute;
};
```

This notation is useful for compactly describing service offers. It contains the IDL interface identifier as well as any number of property definitions for this service type. In this simple example, we are defining a service type called `PrinterService`, which applies to objects that implement the `Printer` interface. We have defined three properties associated with this service type, `location`, `supportedFormats`, and `pagesPerMinute`. All of these properties are specified as being `mandatory`, which means that when a server exports an offer of this service type, it must provide values for all of these properties. The defined properties can be of any IDL data type. Our example uses a string, a sequence of strings, and a long.

Let's consider a second service type example, which is more complex.

[1] The trading service also supports inheritance of service types, which is useful for reflecting the inheritance of the corresponding IDL interfaces.

```
service CompanyResearchService {
    interface Company;
    mandatory property string          companyName;
    mandatory property sequence<string>       sectors;
            property float             currentStockPrice;
};
```

Here, we have defined a service type called `CompanyResearchService`. This service type is used to describe objects that implement the IDL interface `Company`. This service type has defined a number of properties, of varying IDL types. One of these properties is not defined as `mandatory`, which means that offers exported by servers are not required to provide a value for this property. So far, all the properties we have mentioned are *static properties*. A static property has a value provided for it at the time the server exports its offer. For instance, when our server exports a `PrinterService` offer, it will provide values for the `location` and `supportedFormats` properties.

The trading service also supports *dynamic properties*. Rather than supplying a value when the offer is exported, our server instead supplies a callback object. Only when a client performs a query will the trader obtain a value for this property. It does so by making an invocation on this callback object. The server that exported this offer must determine the property's value current value, and return it.

Dynamic properties are very useful when we have attributes that we decide are important for our clients to be able to query on, but happen to frequently change in value. In this example, the `currentStockPrice` property is a dynamic property. Our client application needs to be able to find companies based on their stock price. Since a company's stock price changes frequently, dynamic properties allow us to perform this query, in a standard, efficient manner.[2]

Servers: Export Offers

Once our administrator has defined the service types that will be used by our applications, our servers can export their objects to the trading service. Servers accomplish this by invoking the `export()` method on an object in the trading service, supplying an object, a string identifying the service type, and a sequence of properties.

The object is, of course, an object that implements a particular IDL interface. In particular, it must implement the interface associated with the specified service type.[3] The server also provides a sequence of properties, which are name-value structures. Each property name must match a property name defined in the specified service type. The server must also supply a value for each of the properties. This value is either a concrete value (for static properties), or a callback object (for dynamic properties).

The trader, of course, imposes some consistency checks on offers. In particular, it verifies that the exported object is of an appropriate type, and that values have been supplied for all the mandatory properties.

[2] Interestingly, dynamic properties are not specified as such at the time of service type definition. Instead, this is specified when each offer is exported by a server. Any property can be a dynamic property if the exporting server chooses to implement it as such. Therefore, a single property could have a static value supplied for it in one offer, and support dynamic evaluation in another offer.

[3] The exported object can also be of a type that inherits from the service type's interface.

Clients: Query for Matching Offers

This final step is to have our clients look up objects in the trading service. Clients of the trader invoke a `query()` method on an object in the trader, supplying a number of parameters. First, the client specifies the desired service type. This must match a service type previously defined by the administrator. Recall that a service type defines a set of property names and data types. When servers export offers, they supply values for some (or all) of these properties. Our clients query this set of exported offers by supplying a set of constraints. The trading service evaluates this query and finds a set of offers that match the specified constraints. These offers are returned to the client.

Unlike the name service (which can only return a single object reference as the result of a lookup), the trading service is designed to return multiple offers from a single query. These offers contain not just the object reference, but also some (or all) of the offer's properties. Rather than having the trader automatically returning all of an offer's (potentially many) properties, clients have complete control over the set of properties that are returned with the matching offers.

Let's examine the parameters to `query()` in more detail. The constraint string specified by the client is conceptually similar to a SQL WHERE clause. It specifies the required values for properties against which all offers of a particular service type are to be filtered. This string follows a format specified by the OMG, as part of the trading object service specification. Let's take a look at an sample constraint string, from a case where our client is querying for a `PrinterService` offer:

```
(location == 'first floor') and (pagesPerMinute > 10)
```

This constraint string searches for offers that are on the first floor, and can print more than 10 pages per minute.

Here's another sample constraint string, where our client is querying on the `CompanyResearchService` offers:

```
('airline' in sectors) and (currentStockPrice < 100.00)
```

This query searches for companies in the airline sector, with a current stock price of less than 100.00.

This constraint language provides a flexible mechanism that allows our clients to specify an arbitrarily complex set of criteria for the trader to use to filter the offers.

The client also specifies a list of properties which should be returned by the trader. Recall that the trader returns a sequence of matching offers to the client. This allows the client to further qualify the matching offers before it begins using one. For example, information about the matching offers could be presented to a user, who would then choose the appropriate target. Or, the application could look at the returned values and perform some additional processing on them before determining which offer to use. Rather than having the trader automatically return the values for all of an offer's properties, the client supplies a `specifiedProperties` type. This union contains an enumerated type indicating whether the client wants all of an offer's properties, none of the properties, or some of the properties. If the client wants some of the properties, it specifies which ones by passing in a sequence of property name strings.

The trader returns a sequence of offers to the client. Each offer contains the object reference, as well as a (possibly empty) sequence of name-value pairs, for the requested properties.

Designing a Trading Space

Designing a trading space is primarily a matter of defining service types. Evaluate the ways in which your client applications need to be able to find objects, and define service types made up of those properties. Then, your servers simply export offers, and your clients query the trading service for matches. The trading service does the heavy lifting.

One factor to consider is, of course, the logical and geographical separation of your components. Applications may best be serviced by traders that are close to them, logically or physically. However, a single trader can only return matches from the set of offers that have been directly exported to that trader instance. To overcome this limitation, the trading service supports *linked traders*. When traders are linked together, they can forward client queries to other traders, to find additional matching offers. This *link follow behavior* is determined by both the trader's configuration and some additional options supplied by the client when it performs a query. By linking traders together (also known as *federating*), we form a graph of linked traders. By federating traders across domains, we can provide a wide variety of services to our clients.

Federation depends, of course, on a standardized set of service types. All the linked traders must agree on the precise naming conventions for service types, IDL interfaces, and property names. Today, such an agreement is only likely within a particular organization. However, as the commercial acceptance of CORBA becomes even more widespread, we envision standardized trading spaces defined for entire industry domains.

Other Ways to Locate Objects

In addition to using CORBA services to locate objects, there are other ways to accomplish this as well. We first mention using object reference strings, which is a CORBA-compliant approach that fits into our object location model. Then, we discuss factory pattern. Factories are simply objects that return other objects. This pattern is commonly used throughout CORBA, and is very effective. Next, we cover ORB-specific approaches. These proprietary mechanisms are useful, but are often limited in ways that the CORBA services are not. Finally, we discuss the issue of bootstrapping, which is a CORBA-compliant way for an object to obtain a reference to one of the CORBA services.

Using Object Reference Strings

Another CORBA-compliant way of obtaining object references is to use object reference strings. The CORBA specification requires that ORBs support the `COBRA::ORB::object_to_string()` method. This method *stringifies* the supplied object reference, converting it into a standard string format. A client application invokes the `CORBA::ORB::string_to_object()` method, and the ORB converts this string back into an object reference.

This approach follows our model for locating objects, although the object directory in this case is not a service, but rather just a container chosen by the application designer. Our server publishes its objects by stringifying them, and writing the string to some output location. Logically, these stringified object references are stored in an object directory. In practice, these tend to be stored in client-side configuration files or shell scripts. Client applications look up objects by retrieving the string from its storage location and converting it back into an object reference.

Of course, our client and server programmers must agree on how these object reference strings are managed. When a server writes an object reference to a file, for instance, client developers must know what this file is called, and what object is denoted by the reference contained in it. They must then transfer this string to the client application, which must use it appropriately. This administrative coordination is comparable to that required when using the naming or trading services. In both those instances, client and server developers must agree on the structure and names of entities as well.

Using Factory Objects

A commonly used pattern in CORBA systems is that of the factory object. A factory is any object that returns a reference to another object as the result of a method invocation. We've already seen an example of a factory object—the `PortfolioManager` object in our Portfolio Manager. It's important to note that factory objects are not just used for creating new CORBA objects, but can also be used to return references to existing CORBA objects. This is demonstrated in the `PortfolioManager` IDL, part of which is shown below.

```
// IDL fragment from PortfolioManager
interface PortfolioManager
{
    // This method creates and returns a new Portfolio
    // object
    Portfolio newPortfolio(in string id,
                           in string password);

    // This method returns a reference to an
    // existing Portfolio object
    Portfolio login(in string id,
                    in string password);
}
```

Factory objects are extremely useful, especially when there are a large number of objects that clients can use. Rather than publishing references to all the servant objects, the server can publish just a few factory objects, which the client then uses to obtain references to the remaining objects that it needs. Note that factories are also useful for reducing the number of objects that have to be active in a server process at one time. Rather than eagerly instantiating all possible objects, factories allow the server programmer to defer the instantiation of an object until clients explicitly request it. Both of these benefits make factories important for most large-scale CORBA systems.

In the IDL shown above, the factory returns an object of a specific type, a `Portfolio`. A common alternative to this is to have the factory return an object of the generic type `Object`. This is the approach taken, for example, by the naming service IDL. It allows a factory to return object references of arbitrary types, at the cost of requiring the caller to perform a `_narrow()` on the returned reference.

ORB-Specific Approaches

In addition to supporting the compliant methods specified by CORBA, commercial ORBs will typically also support a proprietary mechanism for locating objects. These mechanisms tend to be very simple to use, and are appealing from that perspective.

For instance, ORBs from both IONA Technologies and Inprise implement proprietary `_bind()` methods, which are generated by the IDL compiler for each interface. The details of these two mechanisms are quite different, but they both suffer from the same limitation—a lack of abstraction.

In both cases, client programmers must provide information that uniquely identifies the target object within an ORB domain. In particular, this information is closely tied to the object instance and system configuration. For example, the client may have to specify information such as the host on which the servant object is running, or the ORB's internal identifier for the object. There is no facility for associating any higher-level information about an object. Clients can only look up objects using this low-level information.

Nonetheless, these proprietary approaches have achieved widespread use. In many cases, this is due to their simplicity. They are very easy to use, and this tended to make them the first choice of object location mechanisms. Over time, we expect that use of these mechanisms will diminish. New applications will make use of the more flexible approaches discussed here, and ORB vendors will deprecate these proprietary methods.

Bootstrapping

The naming and trading services are implemented, naturally, by servant objects that provide a particular interface. Our components (both clients and servers) act as clients when communicating with these services. Just like any CORBA client, however, we need to obtain a reference to these objects before we can begin using them. So, how can we get these references? The recommended way to obtain a reference is to use the naming or trading service. Clearly, we need some other way to bootstrap our programs so that they can obtain an initial reference to an object in one of these services.

One CORBA-compliant way to bootstrap is through the `CORBA::ORB::resolve_initial_references()` method. This call takes as input a single string parameter, and returns an object reference. This string identifies the service from which the caller wants an object reference. Legal values for this string include "NameService" and "TradingService." These strings are specified by the OMG, and all ORBs must return an object reference for the corresponding service.

Another approach would be to obtain an object reference for the desired service, in string form. This stringified object reference could either be output by the service itself or generated by an IOR creation tool. Once our clients have obtained this string (read in from a configuration file, for instance), they can call `CORBA::ORB::string_to_object()` to create a reference for the object.

No matter which bootstrapping mechanism we use, once our applications obtain an initial reference to the repository, they can simply begin using it to publish or lookup application objects.

Selecting An Object Location Mechanism

Choosing which object location mechanism to use is an important decision. Selecting an inappropriate tool can result in an inflexible, difficult-to-maintain system. Although the object location mechanisms differ in their complexity, none of them are truly difficult to use. Once the administrative infrastructure has been defined, all the approaches are straightforward to use (perhaps aided by some client-side wrapper classes customized for your particular environment).

Our primary consideration should be the complexity of our environment. How many objects are going to be published to the object directory? How complex are the criteria that our clients will be using to look up objects? What qualities of service do we require? It's important to make certain that we have a good understanding of how our clients will be looking up objects.

In general, we recommend avoiding the use of ORB-specific mechanisms, for the reasons mentioned earlier. They rely on proprietary mechanisms, have no facility for associating higher-level information with an object, and cannot interoperate with other ORBs.

For very simple, static environments, the use of object reference strings is appropriate. This approach does not rely on the availability of any additional service. Clients just use the object reference string and connect to the appropriate server. Often, clients will retrieve a single object reference string from a configuration file, then use this object as a factory to gain access to the business objects needed.

Of course, if the servant objects are ever moved, our clients will have to be reconfigured with a new object reference string. With large numbers of clients, or geographically distributed clients, this can be difficult and expensive. If this limitation is acceptable, then we can use object reference strings effectively.

Comparing the Naming and Trading Services

For more complex environments, the naming or trading services are more appropriate. They do require some forethought, but provide us with a great deal more functionality than the other approaches. The naming and trading services are most beneficial in slightly differing situations.

The trading space differs from a name hierarchy in that it is multidimensional. Every service type can have any number of properties, and each property defined within a service type is orthogonal to all the others. In addition, new properties can be added to a service type without affecting existing offers. Compare this to the name hierarchy, which is really just two-dimensional. Each object that our servers bind to a name hierarchy only has two pieces of information associated with it—its name and its place in the hierarchy.

Consider our printer objects. As we expressed in the `PrinterService` service type, our clients need to be able to locate printers based on their physical location as well as their print speed. In some cases, end users will want to print to the closest printer, while in other cases they want to find the fastest printer. As we have already seen, supporting these different types of lookup operations is straightforward when using the trader. Trying to duplicate this functionality with the name service, however, is difficult. We would end up with two parallel hierarchies, or a deeply nested hierarchy.

Now, consider what happens if we decide to add a third attribute on which clients can look up objects. Adding this to our service type in the trader is trivial. However, adding this to the name hierarchy requires a complete reworking of the hierarchy. In cases such as this, it quickly becomes apparent that the naming service is not a good choice.

In general, if our clients are only looking up objects based on a fixed set of criteria, then the naming service can usually model this well. If, on the other hand, our clients use varying sets of criteria to look up objects, then the trading service is often a better choice.

Selecting Objects For Publication

How should we decide which of our servant objects should be published to the object directory? The answer, of course, depends on how our system is architected, and how the clients will be using the servant objects. Some systems contain a relatively small, fixed set of servant objects that all clients utilize. In cases such as this, it often makes sense to publish all of these objects. Other systems contain a relatively small number of factory objects, which are used to create short-lived, transient objects that clients will use temporarily. In these cases, we should just publish the factory objects. The transient objects are returned to clients by a factory, and are typically dedicated to one client, containing client-specific state. Because of this, we don't want them to be generally available, so there's no reason to publish them. Also, since these objects don't exist until requested by clients, publishing and looking them up would simply be unnecessary overhead.

Now, let's consider a system with a large number of objects—our Portfolio Manager system, with thousands of `Portfolio` objects in our database. We need to decide how to structure our naming service hierarchy. One approach would be to create an entry for each Portfolio object, named by its unique account number. Alternatively, we could simply export our `PortfolioManager` object, and have clients use that to obtain a reference to a specific `Portfolio` object. This is the preferred approach, for a number of reasons. First, it saves us the trouble of having to instantiate each `Portfolio` object, and bind it to the naming service. Second, adding many entries to the naming service will increase the size of its database, and slow down its processing. Third, by just exporting the `PortfolioManager` factory, it is much easier to relocate our server, or to provide a group of objects for load balancing. We can temporarily (or permanently) relocate our server by simply replacing one entry in the naming service, rather than having to enumerate through thousands of objects.

When our system contains a large number of objects within a single server process, it is often better to export a small number of factory objects from that process, rather than exporting all the servant objects. If our system, however, has objects distributed among many server processes, then it can make sense to export all the servant objects.