The CORBA Component Model: Part 2, Defining Components with the IDL 3.x Types

Our previous column described the evolution of programming language and middleware technologies that have culminated in the CORBA Component Model (CCM), which is a key part of the CORBA 3.x specification. CCM is component middleware that addresses limitations with earlier generations of CORBA 2.x Distributed Object Computing (DOC) middleware.

By

Douglas C. Schmidt and Steve Vinoski

, C/C++ Users Journal
ápr. 01, 2004
URL:http://www.ddj.com/cpp/184403885

Some of the limitations with DOC middleware overcome by CCM include lack of standards for:

- Interface relationships other than inheritance.
- Common patterns of using POA policies.
- Integrating CORBA services (such as notification, lifecycle, and persistence) with distributed applications.
- Generic application server mechanisms.
- Software deployment and configuration tools.

In this column, we further our discussion by illustrating a hybrid publisher/subscriber and request/response distribution architecture that uses CCM features to implement our familiar stock-quoter example, which has been the driving application in our columns for years.

As discussed in the previous column, the CCM specification extends the CORBA object model to support the concept of components, and establishes standards for implementing, packaging, assembling, and deploying component implementations. From a client perspective, a CCM component is an extended CORBA object that encapsulates various interaction models via different interfaces and connection operations. From a server perspective, components are implementation units that can be installed and instantiated independently in standard application server runtime environments stipulated by the CCM specification.

In general, components are larger building blocks than objects, with more of their interactions managed by containers to simplify and automate key aspects of construction, composition, and configuration into applications. Components often need to collaborate with different types of components/applications, which may understand different interface types. Components can therefore express their intention to collaborate with other components/applications by defining ports, which expose different views of their capabilities to clients. There are several types of ports in CCM, including:

- Facets, which provide interfaces that implement point-to-point operations invoked from other components, and receptacles, which indicate a uses dependency on an operation interface provided

by another component. Facets are an instantiation of the Extension Interface pattern [POSA2], which allows each component to export multiple interfaces to prevent breaking of client code and bloating of interfaces when developers extend or modify component functionality. Receptacles define a way to connect a required interface to a component. Facets and receptacles are typically joined together by configuration and deployment tools. Their interconnection can be illustrated by the standard icon in Figure 1(a).

- Event sources and event sinks, which indicate a willingness to exchange typed event messages with one or more components. An event source is a named connection point for event production. There are two styles of event sources: publishers that can have multiple subscribers, and emitters that can have only one subscriber. An event sink is a named interface to which events can be pushed by event sources. An event sink can subscribe to multiple event sources. Event sources and event sinks are also typically joined together by configuration and deployment tools. Their interconnection can be illustrated by the standard icon in Figure 1(b).

In IDL 3.x (which defines component-oriented extensions to IDL 2.x), facets are designated by the *provides* keyword, receptacles are designated by the *uses* keyword, and event sinks are designated by the *consumes* keyword. The two styles of event sources (emitters and publishers) are designated by the *emits* and *publishes* keywords, respectively. Components can also include attribute ports, which specify named parameters that can be configured later via metadata specified in component property files. All the IDL 3.x features can be mapped to equivalent IDL 2.x features, as we illustrate in our stock-quoter example.

In general, when you develop a CCM application, you perform the following steps:

1. Define your interfaces using IDL 2.x features; that is, use the familiar CORBA types (such as *struct, sequence, long, Object, interface, raises*, and so on) to define your interfaces and exceptions.
2. Define your component types using IDL 3.x features; for example, use the new CCM keywords (such as *component, provides, uses, publishes, emits*, and *consumes*) to group the IDL 2.x types together to form components.
3. Use IDL 3.x features to manage the lifecycle of the component types; for example, use the new CCM keyword *home* to define factories that create and destroy component instances.
4. Implement your components; that is, using C++ or Java and the Component Implementation Definition Language (CIDL), which generates the component implementation executors and associated metadata.
5. Assemble your components; for instance, group related components together and characterize their metadata that describes the components present in the assembly.
6. Deploy your components and run your application; that is, move the component assemblies to the appropriate nodes in the distributed system and invoke operations on components to perform the application logic.

In this column, we show how to use CCM features (facets, for instance), receptacles, event sources, event sinks, and attributes defined using IDL 3.x to perform the first three steps just outlined for a component-based version of our stock-quoter example. In subsequent columns, we will examine how to perform the latter three steps that stitch all the components together and run them in application servers.

The Stock Quoter System Architecture

Many examples in our recent columns have focused on request/response communication, where operation requests flow from client to server and responses flow back from server to client. For example, we've often shown examples where a stock broker client obtains information about a particular stock name by invoking the *get_quote()* operation on a *Stock::Quoter* interface implemented by a remote server. While this "polling" approach is fairly common, it can also saturate the server and the network by making many requests, even if the value of the stock hasn't changed!

One way to avoid the problems with request/response polling is to employ a variant of the Publisher/Subscriber architectural pattern [POSA1]. In this pattern, publishers generate events that are transmitted to one or more subscribers, who can then take further action depending on the events they receive and their internal state. The overall flow of information in our latest incarnation of the stock-quoter system works as follows:

1. Stock broker clients subscribe with a stock distributor server to receive notification events whenever a stock value of interest to them changes.
2. The stock distributor server monitors a real-time stock feed database.
3. Whenever the value of a stock changes, the distributor publishes an event to interested stock brokers.
4. If stock brokers are interested in learning more details about a stock whose value has changed, they can invoke an operation on the stock distributor to receive more information.

By employing the Publisher/Subscriber pattern in our stock quoter example, we can alleviate the key drawbacks with polling-based request/response design—stock brokers only contact the stock distributor server via request/response operations when the value of a stock changes, rather than polling them repeatedly to see if the value has changed. Figure 2 illustrates how we can design this type of system using CCM. We define a *StockDistributor* component that publishes events to indicate that a particular stock's value has changed. This component will monitor the real-time stock database and, when the values of particular stocks change, will push a CCM *eventtype* containing the stock name via a CCM event source to the corresponding CCM event sink of one or more *StockBroker* components. The *StockBroker* components that consume this event will then examine the stock name stored in the event. If they are interested in the stock, they can invoke a request/response operation via their CCM receptacle on a CCM facet exported by the *StockDistributor* component to obtain more information about the stock.

The remainder of this column describes each component in Figure 2, focusing on how to program them using CCM features and IDL 3.x. To clarify the behavior of these components "under the hood," we also describe the equivalent IDL 2.x corresponding to the IDL 3.x types. An IDL 3.x compiler typically doesn't generate the IDL 2.x code directly (although it could). However, the C++ or Java mappings that it generates are equivalent to this IDL 2.x code, which can be helpful if you already understand IDL 2.x and you're learning CCM and IDL 3.x. It's important to note, however, that all the equivalent IDL 2.x code we show in this article is not written by application developers, which is one of the key benefits of CCM and CORBA 3.x relative to CORBA 2.x!

Step 1: Defining the Stock-Quoter Interfaces Using IDL 2.x Types

We'll start by defining some interfaces and other types using IDL 2.x features. All of our types (including the IDL 3.x types) will be defined in a module called *Stock*:

```
module Stock {
```

The remaining IDL 2.x and 3.x types are defined within the *Stock* module. When stock brokers want to learn more information about a particular stock whose value has changed recently, they can invoke the *get_stock_info()* operation on the following *StockQuoter* interface:

```
        interface StockQuoter {
                StockInfo get_stock_info (in string stock_name);
        };
```

This interface returns the following *StockInfo struct*:

```
struct StockInfo {
        string name;
                long high;
                long low;
                long last;
                // ...
        };
```

StockInfo contains information about the high and low trading values of the stock during the trading thus far today, along with the most recent value. It also includes the stock name so that each *StockInfo* instance is self-identifying and is thus easily trackable and usable in collections. The stock distributor itself runs as a daemon that can be started and stopped by a system administrator. The *Trigger* interface instructs the stock distributor to perform these control operations:

```
        interface Trigger {
                void start ();
                void stop ();
```

```
        }
```

When an administrator calls *start()*, the stock distributor begins to monitor the real-time stock database until the *stop()* operation is called.

Step 2: Defining the Stock-Quoter Components Using IDL 3.x Types

Now that we've illustrated the IDL 2.x types in our stock-quoter system, we'll show how they are combined using IDL 3.x component types. We start with the *eventtype* data type that components can use to communicate using CCM's publisher/subscriber event mechanism. Whenever a stock value changes, the stock distributor publishes the following *eventtype* containing the name of the stock:

```
        eventtype StockName {
public string name;
};
```

Internally, an IDL 3.x *eventtype* is implemented using an IDL 2.x *valuetype*. Unlike CORBA objects (which are passed by reference), instances of CORBA *eventtype* and *valuetype* are always passed by value. Like *structs*, they can contain state in the form of fields. Unlike *structs*, however, they can have user-defined operations and support inheritance. The equivalent IDL 2.x for the *eventtype* is:

```
valuetype StockName : Components::EventBase
{
        public string name;
};
interface StockNameConsumer : Components::EventConsumerBase {
        void push_StockName (in StockName the_stockname);
};
```

Note how equivalent IDL 2.x code maps the *StockName eventtype* to a *valuetype* that inherits from *Components::EventBase*, which is an abstract *valuetype* defined in the standard CCM *Components* module:

```
module Components
{
        // ...
        abstract valuetype EventBase {};

        interface EventConsumerBase {
                void push_event (in EventBase evt);
        };
};
```

This mapping gives you a greater range of options because it enables publishers to push names of stocks to subscribers as either:

- Concrete *StockName valuetypes* via *StockNameConsumer::push_StockName()*, which is statically typed (and hence less error-prone), but less flexible, or
- Generic *EventBase valuetypes* via *EventConsumerBase::push_event()*, which is more flexible, but dynamically typed (and hence potentially more error-prone).

Now that we've defined our *StockName eventtype*, we can combine it with the IDL 2.x *StockQuoter* interface defined earlier to create our first CCM component, called *StockBroker*, whose ports are in Figure 3. The IDL 3.x description for *StockBroker* is:

```
component StockBroker {
                consumes StockName notifier_in;
                uses StockQuoter quoter_info_in;
        };
```

Although the *StockBroker* component doesn't inherit from anything explicitly, its equivalent IDL 2.x code below shows how it inherits implicitly from *Components::CCMObject*:

```
interface StockBroker : Components::CCMObject {
        // ...
};
```

Components::CCMObject is the base interface for all component types in CCM. By inheriting from this interface, *StockBroker* obtains operations that manage its event sink and receptacle. It also inherits operations that enable discovery of its ports via navigation by component-aware clients. The *StockBroker* component contains two ports that correspond to the two roles it plays. First, it is a subscriber that consumes a *StockName eventtype* called *notifier_in* that's published by the *StockDistributor* when the value of a stock changes. Second, it is a user of the *StockQuoter* interface we defined earlier to provide additional information about a stock. This dependency is indicated explicitly in IDL 3.x via a CCM receptacle called *quoter_info_in* that indicates the uses relationship on the *StockQuoter* interface. The equivalent IDL 2.x for *StockBroker*'s IDL 3.x event sink designator:

```
consumes StockName notifier_in;
```

is

```
StockNameConsumer get_consumer_notifier_in ();
```

which defines a factory operation that returns an object reference to the *StockNameConsumer* equivalent IDL 2.x interface shown earlier. When our stock-quoter system is initialized, standard CCM deployment and configuration tools [D&C] will use this factory operation to connect publishers (such as the *StockDistributor* component) with *StockBroker* subscribers. *StockBroker*'s IDL 3.x receptacle designator

```
uses StockQuoter quoter_info_in;
```

maps to the following group of equivalent IDL 2.x types and operations:

```
void connect_quoter_info_in  (in StockQuoter c);
StockQuoter disconnect_quote_info_in ();
StockQuoter get_connection_quoter_info_in ();
```

These operations are also used by standard CCM deployment and configuration tools to connect the *quoter_info_in* receptacle with the *StockQuoter* facet, which is provided by the *StockDistributor* component whose ports are in Figure 4. The *StockDistributor* component has the following IDL 3.x description:

```
component StockDistributor supports Trigger {
        publishes StockName notifier_out;
        provides StockQuoter quoter_info_out;
        attribute long notification_rate;
};
```

This CCM component supports (also known as "inherits from") the *Trigger* interface defined earlier, which enables a system administrator application to *start()* and *stop()* instances of *StockDistributor*. The equivalent IDL 2.x for *StockDistributor* for supports is:

```
interface StockDistributor : Components::CCMObject, Trigger {
        // ...
};
```

The *supports* keyword is useful for components such as *StockDistributor* that have a "primary" interface, which alleviates the need to go through extra steps just to access the operations of that interface. If the interface were specified as a facet via the *provides* keyword instead, applications would have to call an operation to get a reference to the facet, and then invoke the desired operation. Instead, the *supports* keyword lets administrator applications invoke the *start()* and *stop()* operation directly on the component reference since the *Trigger* interface is a parent of *StockDistributor*. *StockDistributor* also publishes a *StockName eventtype* called *notifier_out* that is pushed to the *StockBroker* subscriber components when a stock value changes. The equivalent IDL 2.x for the

```
publishes StockName notifier_out;
```

IDL 3.x event source designator is defined as:

```
Components::Cookie subscribe_notifier_out (in Stock::StockNameConsumer c);
        Stock::StockNameConsumer unsubscribe_notifier_out (in Components::Cookie ck);
```

This pair of operations is used by standard CCM deployment and configuration tools to subscribe/unsubscribe components that consume the *StockName* events published by the *StockDistributor*. For example, assuming there were *stockBroker* and *stockDistributor* object references to the respective

StockBroker and *StockDistributor* components, these tools could connect automatically the stock distributor publisher to a stock broker subscriber using these steps:

```
Stock::StockNameConsumer_var consumer = stockBroker->get_consumer_notifier_in ();
stockDistributor->subscribe_notifier_out (consumer.in ());
```

The CCM deployment and configuration tools we've mentioned several times provide an important benefit to CCM and CORBA 3.x relative to the earlier CORBA 2.x Standard because they alleviate the need for developers of application components to perform connection "plumbing" programmatically. Moreover, CCM containers that provide the runtime environment for the components will mediate access to *CosNotification* event channels or other mechanisms used to deliver the events, further simplifying the task of application component developers. *StockDistributor* also defines a *StockQuoter* facet called *quoter_info_out*. The IDL 3.x facet designator

```
provides StockQuoter quoter_info_out;
```

in *StockDistributor* maps to the following equivalent IDL 2.x type:

```
StockQuoter provide_quoter_info_out ();
```

which defines a factory operation that returns object references that clients (such as *StockBroker* components) can use to obtain more information about a particular stock. As before, the CCM deployment and configuration tools can use this factory operation in conjunction with the *StockBroker*'s *connect_quoter_info_in()* receptacle operation to perform connection plumbing automatically, as follows:

```
Stock::StockQuoter_var quoter = stockDistributor->provide_quoter_info_out ();
stockBroker->connect_quoter_info_in (quoter.in ());
```

Finally, in addition to the inherited *Trigger* functions, system administrators can use the *notification_rate* attribute to control the rate at which the *StockDistributor* component checks the stock-quote database and pushes changes to *StockBroker* subscribers. Attributes in CCM are primarily used for component configuration; for example, to define optional behaviors, modality, resource hints, and so on. The mapping for attributes in IDL 3.x is the same as for attributes in IDL 2.x; that is, they are represented as a pair of accessor/mutator methods in C++. The only semantic difference is that they can now raise user-defined exceptions, which was not allowed in IDL 2.x.

Step 3: Use IDL 3.x Features to Manage the Lifecycle of the Component Types

Instances of the *StockBroker* and *StockDistributor* components must be created by the CCM runtime system. One well-recognized problem with CORBA 2.x was its lack of a standard way to manage component lifecycles. To rectify this problem, CCM integrates lifecycle management into component definitions via the concept of homes, which are factories that create and destroy component instances. In CCM, home is a new IDL keyword that's used to define a factory that manages one type of component. A component instance is managed by one home instance. Since a home has an interface, it's identified via an object reference. By default, a home has standard lifecycle operations; for instance, *create()*, through which users can define operations with arbitrary parameter lists if the defaults don't suffice. For our stock-quoter example we can use the defaults, so our homes are defined as follows:

```
        home StockBrokerHome manages StockBroker {};
home StockDistributorHome manages StockDistributor {};
```

The equivalent IDL 2.x mapping for these IDL 3.x homes are identical:

```
interface StockBrokerHomeImplicit : Components::KeylessCCMHome[jp17] {
        StockBroker create ();
};
interface StockDistributorHomeImplicit : Components::KeylessCCMHome {
        StockDistributor create ();
};
```

Homes also typically support operations to find components, and the CCM specification also provides a *HomeFinder* interface that an application can use to find a particular home. An application can get a *HomeFinder* by passing the string *"ComponentHomeFinder"* to the ORB's *resolve_initial_references()* operation. These operations reduce the bookkeeping that CORBA applications must do to manage their

objects and factories.

Concluding Remarks

This column illustrated the basics of using the IDL 3.x features specified by the CORBA Component Model (CCM) to define the key components and relationships of our stock-quoter example. The extra features of IDL 3.x make relationships between components and interfaces much clearer than IDL 2.x allows, though as you can see from our examples, there's an equivalent mapping from IDL 3.x to IDL 2.x semantics. IDL 3.x enables the definition of components that can manage multiple views (object instances, for instance), allows component dependencies and publishers/subscribers of events to be specified clearly, and supplies lifecycle management features.

In future columns, we'll examine how to implement our stock-quoter example components using the CCM Component Implementation Framework (CIF), as well as how to apply the new OMG Deployment and Configuration specification [D&C] to stitch all the components together and run them in component servers. As always, if you have any comments on this column or any previous one, please contact us at object_connect@cs.wustl.edu.

References

[CORBA3] "CORBA Components," OMG Document formal/2002-06-65, June 2002.

[D&C] "Deployment and Configuration of Component-based Distributed Applications Specification," OMG Document ptc/2003-07-08, July 2003.

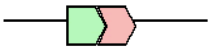
[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern Oriented Software Architecture: A System of Patterns*, Wiley & Sons, 1996.

[POSA2] D. Schmidt, H. Rohnert, M. Stal, and F. Buschmann: *Pattern Oriented Software Architecture: Concurrent and Networked Objects*, Wiley & Sons, 2000.

[Schmidt04] D. Schmidt and S. Vinoski. "The CORBA Component Model, Part 1: Evolving Towards Component Middleware," *C/C++ Users Journal*, February 2004 (http://www.cuj.com/documents/s=9039/cujexp0402vinoski/).

Figure 1: Standard icons.

(a)



(b)

Figure 2: Architecture of the stock-quoter system.

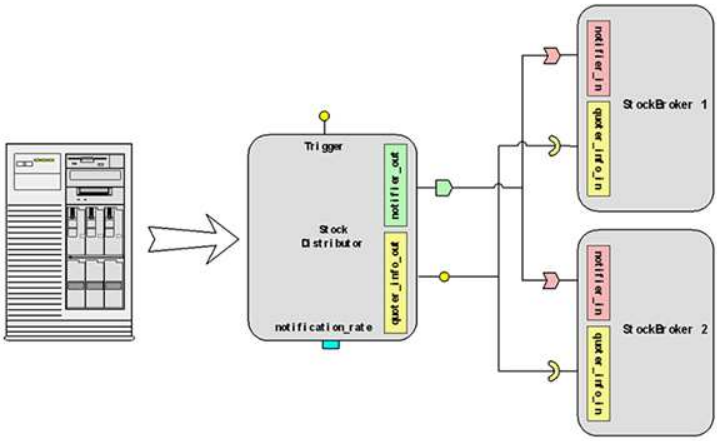


Figure 3: StockBroker component ports.

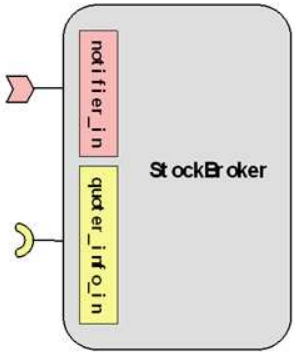
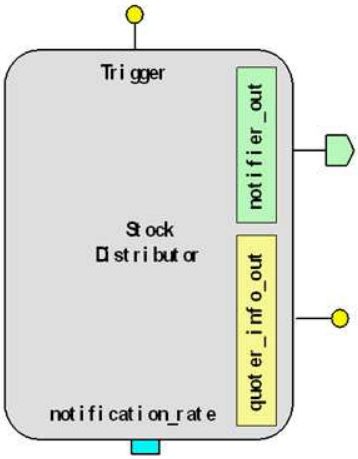


Figure 4: StockDistributor component ports.



Copyright © 2006 CMP Media LLC