Dr. Dobb's Portal
The World of Software Development

The CORBA Component Model Part 4: Implementing Components with CCM

Our previous columns presented the design of our stock quoter example using the CORBA Component Model (CCM) and described how the CCM container architecture and Component Implementation Framework (CIF) are designed. This column deepens our coverage of CCM by showing how the CIF can be used to implement the components in our stock quoter application using C++ and CCM and interact with component containers, which provide the runtime environment for one or more component implementations called executors, which in turn are where components are actually implemented by CCM server application developers.

By Steve Vinoski, C/C++ Users Journal
okt. 01, 2004
URL:http://www.ddj.com/cpp/184403889

The CORBA Component Model Part 4: Implementing Components with CCM

Object Interconnections

Bala Natarajan, Douglas C. Schmidt, and Steve Vinoski

Introduction

Our previous columns [Schmidt04b, Schmidt04c] presented the design of our stock quoter example using the *CORBA Component Model* (CCM) [CORBA3] and described how the CCM *container architecture* and *Component Implementation Framework* (CIF) are designed. This column deepens our coverage of CCM by showing how the CIF can be used to implement the *components* in our stock quoter application using C++ and CCM and interact with component containers, which provide the runtime environment for one or more component implementations called *executors*, which in turn are where components are actually implemented by CCM server application developers. Figure 1 illustrates the components in our stock quoter system example. The **StockDistributor** component monitors a real-time stock database. When the values of particular stocks change, it pushes a CCM **eventtype** that contains the stock's name via a CCM event source to the corresponding CCM event sink implemented by one or more **StockBroker** components. If these components are interested in the stock they can obtain more information about it by invoking a request/response operation via their CCM receptacle on a CCM facet exported by the **StockDistributor** component.

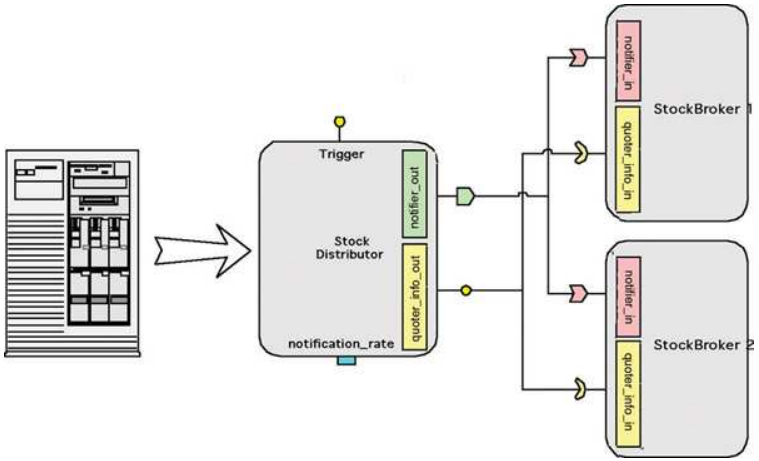

Figure 1: CCM Architecture of the Stock Quoter System

The remainder of this column shows how to implement the stock quoter application shown in Figure 1, focusing on how to implement the **StockBroker** and **StockDistributor** components using CCM features. In particular, the CCM container framework and CIF described in our previous column [Schmidt04c] provide the following important capabilities to developers of the stock quoter system:

- The CIF provides access to all the services available in containers using the **SessionContext** and **SessionComponent** context interfaces described in our previous column.
- Rather than having to write the introspection, navigation, and port connection management code manually, the CIF can generate this code automatically.
- Developer-specified metadata expressed in XML can be used to instruct CCM deployment mechanisms how to control the lifetime of these containers and the components they manage. The metadata is present in XML files called *descriptors*, which we'll cover in our next "Object Interconnections" column.

Implementing Components

This section shows how to use CCM containers and the CIF to implement the components in our stock quoter example. The example is illustrated in Figure 1, and is implemented using the following steps:

1. **Create component implementation definitions** by first defining components and their homes using IDL3.x keywords and then using the *Component Implementation Definition Language* (CIDL) to describe the structure and state of the component implementations, which is referred to as the *composition* of a component.
2. **Compile the component implementation definitions** by running them through a CIDL compiler, which generates implementation skeleton "glue" code (e.g., servants and executor interfaces) that reduces the amount of software written by server application developers.
3. **Implement component executors**, which ultimately perform the business logic requested by clients.

We discuss each of these steps in greater detail below.

Step 1: Create Component Implementation Definitions

As with the distributed object computing programming techniques supported in CORBA 2.x, application developers must first define interfaces for components that need to be developed and deployed. We start

by defining our components using IDL3.x keywords (such as **component**, **home**, **provides**, **uses**, **publishes**, **emits**, and **consumes**), as illustrated in [Schmidt04b]. We also use IDL 2.x keywords (such as **struct**, **sequence**, **long**, **Object**, **interface**, **raises**, etc.) to complete the definition.

Component developers need to define the structure and state of their components. The CCM approach is to use the **composition** declaration in CIDL that describes how to connect *component definitions* (which perform business logic) with *home definitions* (which are the factories that create components). Since a component can be instantiated by more than one home, compositions designate which home will manage which component. The general form of a CIDL composition is shown in Figure 2.

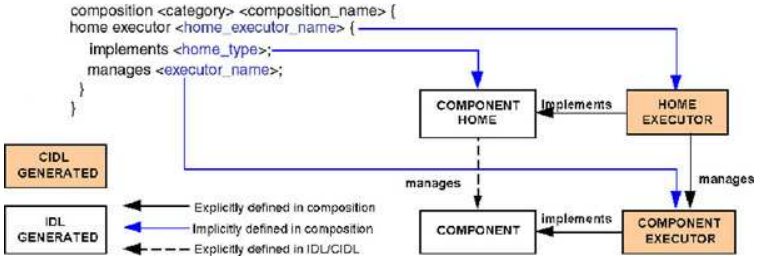

Figure 2: Overview of a CIDL Composition

This figure illustrates the relationships between the following entities defined in a composition:

- A composition declaration using the keyword **composition**. The *category* defines the type of component category used in this composition, which could be one of **service**, **session**, **entity**, and **process**, as described in [Schmidt:04c].
- The *composition_ name* provides a unique identifier that identifies this composition within the translation unit. The CIDL compiler uses the composition name as the namespace that contains all the generated implementations.
- The composition specifies a **home executor** definition that identifies the home executor within a scope. The **implements** definition declares the *home type*, which is the focal point of the composition. To implement application logic, component developers will subsequently extend interfaces with the name *home_executor_name*, as we show in Step 3 below.
- The keyword **manages** identifies the *executor_name* that provides the component executor definitions used by server application developers to extend and implement the business logic of the component within a component server. Step 3 below also shows how to implement homes and component executors.

Both Figure 2 and the explanation above reinforce that the focus of compositions is the *home* rather than the *component*. This design is essentially the Factory Method pattern [GoF], which localizes the creation of an object in a subclass. In CCM, the home is the primary point of contact for a client, and the home's interface and behavior impact the interaction between clients and the component.

CCM allows developers to define a single component and have multiple home definitions that manage the same component using IDL3.x keywords. By defining the composition, the component developer designates the relationship between a home and the component it manages. It is important to note, however, that the composition does not specify component types explicitly, but rather implicitly from the home type within the composition since each home can manage only one type of component.

The CIDL for our stock quoter example is placed in a file called **stock.cdl**, as shown below:

```
composition session StockDistributor_Impl {
       home executor StockDistributorHome_Exec  {
              implements StockDistributorHome;
              manages StockDistributor_Exec;
       }
```

```
};
composition session StockBroker_Impl {
       home executor StockBrokerHome_Exec  {
              implements StockBrokerHome;
              manages StockBroker_Exec;
       }
};
```

These two compositions convey the following information to a CIDL compiler:

- **StockDistributor_Impl** and **StockBroker_Impl** specify the name of two compositions in our stock quoter example. Our stock broker example uses the **session** component category, which is characterized by transient state, a nonpersistent identity, and a behavior implemented as operations defined on the facet of the component, as described in [Schmidt:04c].
- **StockDistributorHome_Exec** and **StockBrokerHome_Exec** are the home executors in our stock quoter example. Likewise, **StockDistributor_Exec** and **StockBroker_Exec** are the component executors managed by **StockDistributorHome_Exec** and **StockBrokerHome_Exec**, respectively. The component types **StockBroker** and **StockDistributor** are implicitly referred to by the composition through the specification of home types that are managed by the **StrockBrokerHome_Exec** and **StockDistributorHome_Exec** home executors. We show how to actually implement these home and component executors in Step 3 below.

The compositions described above show the mandatory parts of a composition specification. CIDL also allows server application developers to define the following optional specifications:

1. ***Storage specifications***, which provide ways to associate state information with the components. The binding of the state information with the components occurs through the *uses catalog* specification within the composition definition. The composition will refer to a *persistent state definition language* [PSDL] definition for associating state information.
2. ***Home operation delegation specification***, which maps operations defined on the component home to operations on either the abstract storage home or the component executor. The CIDL uses this description to generate implementations of operations on the home executor, and to generate operation declarations on the component executor. Operation delegations can be specified explicitly or implicitly in the composition.
3. ***Proxy home***, which implements the home interface for the component, but is not required to be collocated with the container that executes the component implementation managed by the home. In some configurations, proxy homes can provide implementations of home operations without contacting the container that executes the actual home and component implementation. The use of proxy homes is transparent to clients and largely transparent to component implementations. The runtime component of CIF is expected to maintain the relation between the proxy homes and the actual homes that they represent.
4. ***Segmented executors***, which are a set of distinct artifacts from the containers perspective, as opposed to *monolithic executors*, which are treated as a single entity by the container. The component executor in our **stock.cdl** file is an example of monolithic executors. Each segment in a segmented executor may have distinct abstract state declarations. The motivation is to allow a segmented executor with at least one facet to be activated individually, instead of activating the whole component.

We omit these optional composition features in our example since we don't need them to implement our stock broker application.

Step 2: Compile the Component Implementation Definitions to Generate Glue Code

Component applications that implement the business logic are developed by extending the language-specific interfaces generated from the composition definitions created in Step 1 above. These interfaces are generated by compiling the CIDL files using a CCM-compliant CIDL compiler. In addition to generating C++ code and IDL interfaces that are needed for component implementations, a CIDL

compiler also generates a considerable amount of glue code that is used for component configuration and deployment (note that a CCM-compliant CIDL implementation can generate Java or C++ code instead of IDL). In CORBA 2.x applications, application developers had to manually write nontrivial amounts of this glue code themselves, which is tedious and error prone. In CCM, much of this code is generated automatically and manipulated by various compilers and tools, which helps separate concerns and reduce development effort.

In our stock quoter example, when the **stock.cdl** file is passed to the CIAO CIDL compiler (http://www.dre.vanderbilt.edu/CIAO), it generates the following files shown in Figure 3 and described below. The names of the files and contents are based on the CIAO CCM implementation and are not standardized. They have been shown to enhance your understanding of the generated code and subsequent C++ code examples.

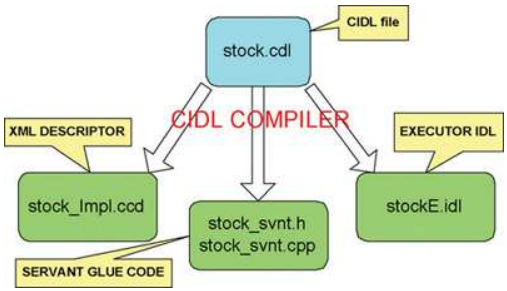

Figure 3: Files Generated by CIAO's CIDL Compiler for stock.cdl

- **IDL 2.x mappings for executors** defined within the CIDL are generated in **stockE.idl** (also called as executor IDLs). For example, the CIAO CIDL compiler generates the associated executor IDL 2.x interfaces, which in our example are called **StockDistributorHome_Exec**, **StockBrokerHome_Exec**, **StockDistributor_Exec**, and **StockBroker_Exec**, based on the names used in the **stock.cdl** file. All IDL interfaces generated by a CIDL compiler are *local interfaces*, which means that servants for these local interfaces are locality-constrained CORBA objects that cannot be activated within the POA. Moreover, object references of locality-constrained objects cannot be exported outside the address space of the object; i.e., applications cannot call **ORB::object_to_string()** or cannot (de)marshal the reference to/from a CDR stream. The CCM container infrastructure is responsible for mapping the servant generated for the component with the executor implementation provided by the component developer. An IDL compiler can be used to create language-specific mapping (e.g., C++ or Java) from the generated IDL 2.x executor IDL interfaces.

  A snippet of the executor IDL generated by the CIAO CIDL compiler for the **StockBroker_Impl** is:

```
local interface CCM_StockBroker : ::Components::EnterpriseComponent {
    void push_notifier_in (in ::StockName e);
};

module  StockBroker_Impl {
   local interface StockBroker_Exec : CCM_StockBroker,
                                      Components::SessionComponent {};
};
```

  The executor IDL 2.x mapping is standardized by the CCM specification, although it is entirely possible and permissible for a CIDL implementation to generate language-specific mapping directly from the CIDL specifications; i.e., bypassing the IDL 2.x executor IDL mapping. Component developers must extend the executor IDL interfaces to implement operations that perform their business logic, as shown in Step 3 below.

- **IDL2.x mappings for homes** defined within the IDL are generated in stockE.idl. For example, a CIDL compiler can generate the local interfaces **CCM_StockBrokerHomeExplicit**, **CCM_StockDistributorHomeExplicit**, **CCM_StockBrokerHomeImplicit**, and **CCM_StockDistributorHomeImplicit**, which have default **create()** factory operations that the CIF uses at runtime to create the corresponding component executors. Other operations declared within the home definitions are also generated. A snippet of the code generated from the CIDL compiler for homes is shown here:

```
local interface CCM_StockBrokerHomeImplicit {
   Components::EnterpriseComponent create ();
};
local interface CCM_StockBrokerHomeExplicit : Components::HomeExecutorBase {};
local interface CCM_StockBrokerHome : CCM_StockBrokerHomeExplicit,
                                      CCM_StockBrokerHomeImplicit {};
module StockBroker_Impl {
   local interface StockBrokerHome_Exec : ::CCM_StockBrokerHome {};
};
```

- **Servant code generation** for the executors used by the container and glue code for components that perform routine tasks are generated in **stock_svnt.h** and **stock_svnt.cpp**. For example, when a CIDL compiler processes the stock.cdl file, it will generate CORBA servants in C++ called **StockBroker_Servant**, **StockBrokerHome_Servant**, **StockDistributor_Servant**, **StockDistributorHome_Servant** and servants for *every* facet defined in the component, such as **StockQuoter_Servant**. These servants contain operations for all the ports (e.g., facets, receptacles, event sources/sinks, and attributes), the container callback operations (e.g., **ccm_activate()**), and the navigation and introspection operations (e.g., **get_all_facets()** and **get_named_facets()**). For example, the generated **StockBroker_Servant** class contains the **connect_quoter_info_in()**, **disconnect_quoter_info_in()**, **get_quoter_info_in()**, **get_consumer_notifier_in()** operations corresponding to the ports defined in Figure 1, as shown here:

```
class StockBroker_Servant : public virtual POA_StockBroker,
                            public virtual PortableServer::RefCountServantBase {
public:
    // Operation for the "consumes" port.
    StockNameConsumer_ptr get_consumer_notifier_in ();
    // Operation for the "uses" port.
    void connect_quoter_info_in (StockQuoter_ptr c);
    StockQuoter_ptr disconnect_quoter_info_in ();
    StockQuoter_ptr get_connection_quoter_info_in ();
    // Operations for Navigation interface.
    CORBA::Object_ptr   provide_facet (const char *name);
    Components::FacetDescriptions get_all_facets ();
    Components::FacetDescriptions  get_named_facets (const ::Components::NameList &);
    // Operations for Receptacles interface.
    Components::Cookie *connect (const char *name,
                                 CORBA::Object_ptr connection);
    CORBA::Object_ptr  disconnect (const char *name,
                                   ::Components::Cookie *ck);
    Components::ReceptacleDescriptions *get_all_receptacles ();
    // Operations for Events interface.
    Components::EventConsumerBase_ptr  get_consumer (const char *sink_name);
    Components::Cookie *subscribe (const char *publisher_name,
                                   ::Components::EventConsumerBase_ptr subscriber);
    // more operations...
};
```

- **Component category-specific context interfaces** are generated, which allow the component to bootstrap with its container and use container-provided services, such as persistence and event notification mechanisms. For example, the CIAO CIDL compiler generates the following local interfaces as part of its standard executor IDL mapping in the file **stockE.idl**:

```
local interface CCM_StockBroker_Context : ::Components::SessionContext {
   StockQuoter get_connection_quoter_info_in ();
```

```
};
module StockBroker_Impl {
   typedef ::CCM_StockBroker_Context  StockBroker_Exec_Context;
};
```

It also generates an implementation of component-specific context in the C++ servant glue code in the files **stock_svnt.h** and **stock_svnt.cpp**. A component-specific context interface is generated for every component in the executor IDL file; i.e., **stockE.idl** and an implementation for the local interface is generated in the C++ servant glue code. For instance, here's the component-specific context for our **StockBroker** component:

```
namespace StockBroker_Impl {
   class StockBroker_Context  : public virtual CCM_StockBroker_Context    {
       virtual ::Components::Principal_ptr get_caller_principal ();
       virtual ::Components::CCMHome_ptr get_CCM_home ();
       //
};
```

The CCM container framework creates an instance of **StockBroker_Impl::StockBroker_Context** during component deployment, passing it to the component executor via the **set_session_context()** callback operation if a session container is used or via **set_entity_context()** if an entity container is used. The context interface generated within the executor IDL; i.e., **StockBroker_Impl:: StockBroker_Exec_Context**, is the data type that the component application uses to interact with the container. We recommend that component applications hold a reference to the pointer it receives during the **set_session_context()** callback operation by calling **_duplicate()**, as shown in Step 3 below.

- **XML descriptors containing metadata** describing the stock broker components are generated by CIAO's CIDL compiler in the **stock_Impl.ccd** file. In general, a CIDL compiler will generate CORBA component descriptors (ccd) for every component that it sees as part of the composition. These descriptors list the ports declared in the component with the repository IDs associated with those ports. The schema for the descriptors is specified as part of the CCM specification. The XML descriptors generated by the CIDL compiler can be used by CCM deployment and configuration tools. The following snippet of ccd in XML was generated by CIAO's CIDL compiler for the **StockBroker** component contains information about ports and the corresponding repository IDs:

```
<corbacomponent>
<corbaversion>3.0</corbaversion>
    <componentrepid repid="IDL:StockBroker:1.0"/>
      <homerepid repid="IDL:StockBrokerHome:1.0"/>
        <componentkind>
          <session>
            <servant lifetime="container"/>
          </session>
        </componentkind>
    <homefeatures name="StockBrokerHome"
      repid="IDL:StockBrokerHome:1.0">
  </homefeatures>
<componentfeatures name="StockBroker" repid="IDL:StockBroker:1.0">
    <ports>
      <consumes  consumesname="notifier_in" eventtype="IDL:StockName:1.0">
        <eventpolicy policy="normal"/>
      </consumes>
      <uses usesname="quoter_info_in" repid="IDL:StockQuoter:1.0"></uses>
    </ports>
  </componentfeatures>
<interface name="StockQuoter" repid="IDL:StockQuoter:1.0"></interface>
</corbacomponent>
```

Server application developers can treat the code generated by the CIDL compiler (including the language-specific servant code and IDL 2.x IDLs) as a black-box; i.e., component application developers just need to compile and link the generated code with their code to create applications

and libraries. Although operations in the component/home servants generated by the CIDL compiler directly handle certain bookkeeping operations, such as navigation and introspection operations, they forward client business logic requests on the component to its corresponding executor. Server application developers therefore extend the C++ classes (***Home_Exec** and ***_Exec**) generated by the IDL 3.x compiler when it processes the executor IDL synthesized by the CIDL compiler. In our stock quoter example shown in Step 3 below, the executor classes we write are called **StockBrokerHome_Exec_i**, **StockBroker_Exec_i**, **StockDistributorHome_Exec_i**, and **StockDistributor_Exec_i**, where the ***Exec_i** suffix indicates that these are executor classes that we inherit from the generated ***_Exec** classes and implement to provide application-specific business logic.

Step 3: Implement Component and Home Executors

Up to this point, we have focused on the various CIDL-generated definitions that are required to bootstrap the CCM component server infrastructure. In this third step, we describe how to implement executors for the **StockBroker** and **StockDistributor** components and the homes that create them. These executor implementations form the bulk of the code that is actually written by server application developers — nearly everything else is generated automatically by either the CIDL compiler and/or IDL 3.x compiler! Implementing the StockBroker component executor. We first present the implementation of the CCM StockBroker component, whose ports are shown here:

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

StockBroker contains two ports that correspond to the following two roles it plays:

- It's a subscriber that consumes a **StockName eventtype** called **notifier_in** that's published by the **StockDistributor** when the value of a stock changes. As shown in Figure 1, the **notifier_in** event sink will be connected to the **StockDistributor**'s **notifier_out** event source by the standard CCM deployment and configuration tools when the application is launched.
- It uses the **StockQuoter** interface provided by the StockDistributor component, which reports additional information about a stock, such as the high, low, and most recent trading values of the stock during the day. The dependency of **StockBroker** on **StockQuoter** is indicated explicitly in IDL 3.x via the **quoter_info_in** receptacle, which will be connected to **StockDistributor**'s **quoter_info_out** facade by the deployment and configuration tools when the application is launched.

The following C++ code fragments illustrate one way to implement the **StockBroker** component. We start by defining a namespace for our executor.

```
namespace StockBroker_Impl {
```

We then define our **StockBroker_Exec_i** executor implementation, which inherits from the **StockBroker_Exec** local executor class generated by the IDL compiler from the executor IDL produced by CIAO's CIDL compiler when it processed the **StockBroker_Impl** composition shown in Step 1:

```
class StockBroker_Exec_i  : public virtual StockBroker_Exec {
```

The primary method in **StockBroker_Exec_i** we need to implement is **push_notifier_in**(), which is dispatched by the **StockBroker**'s ORB when a **StockDistributor** publishes a **StockName** in response to a change in the price of a stock it's monitoring.

```
                void push_notifier_in (Stock::StockName *stock) {
```

If the name of the stock is "ACME ORB Inc." this method uses the session context managed by the component's container to obtain the object reference to the **StockQuoter** instance provided by the **StockDistributor** component, as follows:

```
if (stock->name == "ACME ORB Inc.") {
    Stock::StockQuoter_var quoter =
context_->get_connection_quoter_info_in ();
```

After obtaining the object reference, the **push_notifier_in()** method calls back to the **StockDistributor** component to obtain the latest information about the stock it's interested in.

```
    Stock::StockInfo_var info =
  quoter->get_stock_info (stockname);
    // ...Use this info for various purposes...
      }
}
```

The executor implementation also needs to implement the following container callback methods:

```
void  ccm_activate (void);
void ccm_passivate (void);
void  ccm_remove (void);
```

The CCM container framework uses **ccm_activate()** to callback on the StockBroker component when the component is ready to receive requests. This method can initialize application-specific data members. The container framework also calls **ccm_passivate()** and **ccm_remove()** to enable the component to passivate and remove itself from service, respectively. We don't show the implementations of these methods since they aren't germane to the column. The container callback method that is of interest in this column is the following:

```
void set_session_context (Components::SessionContext_ptr ctx) {
  context_ = CCM_StockBroker_Context::_narrow (ctx);
```

}

This method is called back when the component has been created, but before activation during the deployment phase, to store the context information within the executor implementation which is subsequently used during the **push_notifier_in** () operation shown above.

```
    };
};
```

Some things to note about the implementation of the **StockBroker** component executor include:

- The container that hosts the **StockBroker** component keeps track of the context in which it runs, i.e., it stores the **StockQuoter** object reference that was connected to the **StockBroker** component by the standard CCM deployment and configuration tools when the application was launched, thereby reducing the amount of code that must be written by application developers.
- Application developers also don't need to worry about implementing any of the introspection and navigation interfaces since the CIDL compiler generates them automatically.

Implementing the StockDistributor component executor. We now present the implementation of the **StockDistributor** component, whose ports are shown here:

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

This CCM component extends the **Trigger** interface defined in [Schmidt:04b], which enables system administrator applications to **start()** and **stop()** instances of **StockDistributor**. It also publishes a **StockName eventtype** called **notifier_out** that is pushed to the **StockBroker** subscriber components when a stock value changes. In addition, it defines a **StockQuoter** facet called **quoter_info_out**, which defines a factory operation that returns object references that **StockBroker** components can use to obtain more information about a particular stock. Finally, this component defines the **notification_rate** attribute, which system administrator applications can use to control the rate at which the **StockDistributor** component checks the stock quote database and pushes changes to **StockBroker** subscribers.

The C++ implementation of the executor for the **StockDistributor** component is shown here:

```
namespace StockDistributor_Impl  {
            class StockDistributor_Exec_i : public virtual StockDistributor_Exec,
                                            public virtual ACE_Task_Base {
```

The **StockDistributor_Exec_i** class inherits from the **StockDistributor_Exec** local executor class generated by the IDL compiler from the executor IDL produced by CIAO CIDL compiler. It also inherits from **ACE_Task_Base** [C++NPv2], which is a base class provided by ACE that can be used to convert **StockDistributor_Exec_i** into an active object [POSA2]. When a system administrator application calls the **start()** operation, we use the **ACE_Task_Base::activate()** method to convert the executor into an active object, as follows:

```
void start () {
      activate ();
}
```

The **activate()** method is defined in **ACE_Task_Base** to spawn a new thread of control, which automatically invokes the following **svc()** hook method, of which we show just a portion:

```
virtual int svc () {
     while (thr_mgr ()->testcancel () == 0) {
 StockName_var stock;
 // ... monitor the stock database for changes ...
 stock.name = "ACME ORB Inc.";
 context_->push_notifier_out (stock.in ());
       }
}
```

This method runs continuously, monitoring the database and pushing **StockName** events to **StockBroker** consumers when stocks change their values. It stops when it's canceled, which occurs when a system administrator application calls the **stop()** operation:

```
void stop () {
     thr_mgr ()->cancel ();
}
```

The implementation of the **ACE_Thread_Manager::cancel()** method [C++NPv1] shown above sets an internal flag in the thread manager indicating the thread has been placed into the cancelled state, but doesn't perform asynchronous cancellation (which is problematic in C++ since asynchronous cancellation doesn't guarantee destructors are executed as the runtime stack is unwound). Since the **StockDistributor** component provides **StockQuoter** as a facet, the **get_quoter_info_out()** method can be implemented in

our executor to return a new instance of **StockQuoter_i**:

```
CCM_StockQuoter_ptr get_quoter_info_out () {
       return new StockQuoter_i;
}
```

To do that, however, we'll need to extend the executor skeleton of the **StockQuoter** facet and provide the following implementation:

```
namespace StockDistributor_Impl  {
          StockQuoter_i : public virtual CCM_StockQuoter {
             // operations and implementations
          };
};
```

The container for the **StockDistributor** component caches the local object of **StockQuoter_i** and maps the actual servant's reference it generates for the facet to the local object implementation that the component application provides. All calls on the generated servant are then delegated to the C++ class instance that implements the operation. For example, the **StockQuoter_Servant** generated by CIDL for the facet in the **StockDistributor** component is mapped to the implementation of the **CCM_StockQuoter** interface, which in our case happens to be **StockQuoter_i**. Any calls on operations defined within the **StockQuoter** interface are delegated to the implementation in the **CCM_StockQuoter** interface.

Note that the executor actually implements **get_*()** instead of **provide_*()**, which was shown as the IDL 2.x mapping of the **provides** keyword in our previous column [Schmidt:04b]. The IDL 2.x mapping presented in our previous column is an external *client*'s view of the component. A client that is *not* component-enabled can get the facet by calling the **provide_*()** operation on the components reference. In contrast, **get_*()** is the server-side executor mapping of the facet. The component's **provide_*()** operation to retrieve the facet is mapped automatically to the executor's **get_*()** method, which is then implemented by component developers.

The final methods we need to implement in our executor are simply the accessor and mutator methods for the **notification_rate** attribute:

```
CORBA::Long notification_rate () { return rate_; }
void notification_rate (CORBA::Long rate) { rate_ = rate; }
```

There's a corresponding data member in the private part of the executor class to store the rate value:

```
        private:
            CORBA::Long rate_;
    };
};
```

We omitted the implementations of the container callback operations on the **StockDistributor_Exec_i** class since the implementations are similar to the implementations in the **StockBroker_Exec_i** class.

Implementing the component homes. Instances of the **StockBroker** and **StockDistributor** components must be created by the CCM runtime system using the following component homes:

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

The C++ implementation of **StockBrokerHome** is shown here:

```
class StockBrokerHome_i : public virtual StockBrokerHome_Exec  {
      Components::EnterpriseComponent_ptr create () {
         StockBroker_i broker = new StockBroker_i;
```

```
        return broker->_this ();
      }

};
```

The implementation of **StockDistributorHome** is nearly identical, so we omit it.

Finally, we define the two entry points into the dynamic link library (DLL) in which our executor code is contained, as follows:

```
extern "C" {
components::HomeExecutorBase_ptr
createStockBrokerHome_Impl (void)
{
      return new StockBroker_Impl::StockBrokerHome_Exec_i;
}


components::HomeExecutorBase_ptr
createStockDistributorHome_Impl (void)
{
       return new StockBroker_Impl::StockDistributorHome_Exec_i;
}
}
```

The motivation for this entry point is to allow the standard CCM deployment and configuration tools to locate the home within the executor DLL. One of the descriptors associated with a component is called the *implementation artifact descriptor*, which is an XML instance document based on an OMG-specified schema that maps the component name to the DLL in which the implementation resides. The document also contains the entry point that can be invoked by the tools to instantiate a home upon opening the DLL. The name of the entry point can be specified in the XML document, but the signature of the C-style function is defined in the CCM specification.

Concluding Remarks

This column illustrated how to implement the components in our stock quoter example application using the CCM container framework and Component Implementation Framework (CIF). We explained details of the three canonical steps involved in implementing these components, specifically:

1. Creating component implementation definitions and compositions using IDL 3.x and CIDL.
2. Compiling the component compositions to generate most of the glue code required by your application.
3. Implementing the component and home executors that define the business logic of your application.

We showed a lot of C++ and IDL code in this column, but as we explained in Step 2, the CCM CIF generates most of it for you. The automatic generation of a fairly large amount of support code is one of the strengths of this approach over the older CORBA 2.x approach, where developers normally wrote this code for themselves.

Our next column will illustrate how to apply the new OMG Deployment and Configuration specification [D&C] to assemble components (e.g., group related components together and characterize their metadata that describes the components present in the assembly) and deploy your components and run your application (e.g., move the component assemblies to the appropriate nodes in the distributed system and invoke operations on components to perform the application logic). 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.

[C++NPv1] Schmidt, D. and S. Huston. *C++ Network Programming: Mastering Complexity with ACE and Patterns,* Addison-Wesley, 2002. http://www.cs.wustl.edu/~schmidt/ACE/book1/

[C++NPv2] Schmidt, D. and S. Huston. *C++ Network Programming: Systematic Reuse with ACE and Frameworks,* Addison-Wesley, 2003. http://www.cs.wustl.edu/~schmidt/ACE/book2/

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

[GoF], Gamma, E., R. Helms, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, MA, 1995.

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

[POSA2] Schmidt, D., H. Rohnert, M. Stal, and F. Buschmann. *Pattern Oriented Software Architecture: Concurrent and Networked Objects*, Wiley & Sons, 2000. http://www.cs.wustl.edu/~schmidt/POSA/

[Schmidt04a] Schmidt, D. 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/

[Schmidt04b] Schmidt, D. and S. Vinoski. "The CORBA Component Model, Part 2: Defining Components with the IDL 3.x Types," *C/C++ Users Journal*, April 2004. http://www.cuj.com/documents/s=9152/cujexp0404vinoski/

[Schmidt04c] Schmidt, D. and S. Vinoski. "The CORBA Component Model, Part 3: The CCM Container Architecture and Component Implementation Framework," *C/C++ Users Journal*, April 2004. http://www.cuj.com/documents/s=9301/cujexp0409vinoski/

Copyright © 2006 CMP Media LLC