Dr. Dobb's Portal
The World of Software Development

The CORBA Component Model

In this column, CCM expert Bala Natarajan joins us to expand our coverage of CCM by describing the container architecture, which provides a powerful runtime environment for components, and the Component Implementation Framework (CIF), which generates a significant amount of code so it needn't be written manually by CCM server application developers.

By Bala Natarajan, Douglas C. Schmidt, and Steve Vinoski, C/C++ Users Journal
szept. 01, 2004
URL:http://www.ddj.com/cpp/184403888

Introduction

Our previous column [Schmidt04b] illustrated portions of a hybrid publisher/subscriber and request/response distribution architecture that used CORBA Component Model (CCM) [CORBA3] features to enhance our familiar stock quoter example and alleviate key drawbacks with the use of polling-based request/response interactions. That column also focused on the use of IDL 3.x (which extends IDL 2.x with component-based features) to define the various CCM types used by our enhanced stock quoter example. In particular, we described the use of *components*, which extend CORBA objects by encapsulating various interaction models via *ports*, including (1) *facets*, which define an interface that accepts point-to-point operation invocations from other components, (2) *receptacles*, which indicate a dependency on a point-to-point operation interface provided by another component, and (3) *event sources/sinks*, which indicate a willingness to exchange typed messages with one or more components. In this column, CCM expert Bala Natarajan joins us to expand our coverage of CCM by describing the *container architecture*, which provides a powerful runtime environment for components, and the *Component Implementation Framework* (CIF), which generates a significant amount of code so it needn't be written manually 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.

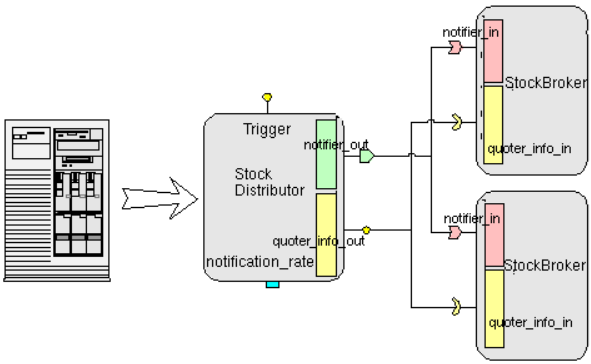

Figure 1: CCM Architecture of the Stock Quoter System.

The remainder of this column describes the CCM container architecture and Component Implementation Framework. Our next column will show how to implement the **StockBroker** and **StockDistributor** components using these CCM features.

The CCM Container Architecture

As discussed in [Schmidt04a], CCM components provide more powerful building blocks than CORBA objects. With CCM, more of the interactions between components are managed by middleware platform mechanisms and tools, thereby simplifying and automating key aspects of component-based applications. Specifically, CCM provides elements that assist with the construction, composition, configuration, and deployment of components into applications, and these elements are based on common patterns learned during the building and deploying of CORBA object systems over the previous decade. A key CCM element is the *container*, which is a framework that provides the runtime environment for one or more component implementations called *executors*, which are where components are actually implemented by server application developers. Containers provide the following capabilities for component application developers:

- They ease the task of building and deploying CCM application components by shielding component developers from low-level details of the underlying middleware. Though the CCM container framework doesn't prevent direct access to the underlying middleware framework, it's designed to provide a richer set of interfaces suitable for a broad range of applications.
- They are responsible for working in conjunction with other CCM features and tools to locate and create component instances, interconnect components, and enforce component policies associated with common services, such as lifecycle, security, and persistence.

Developers select and configure the container runtime environment and its associated policies using programming artifacts provided by the *Component Implementation Framework (CIF)*, which automates much of the component implementation "glue" code. The glue code generated from CIF and the executors developed by component developers interact with the container through a set of standard interfaces to provide a powerful component runtime environment.

Each container and the components it manages can be grouped together in an executable program known as a *component server* within which containers and the components they manage are instantiated. To maximize flexibility, glue code and component implementations are normally housed in separate dynamic linked libraries (DLLs), and are loaded into the component server's process at deployment time. With the addition of extra tools beyond those in the CCM specification, it's also possible to configure executors within component servers statically, which is useful for real-time and embedded systems. [CIAOSTAT] is an example of one such tool that can be used to statically configure CCM components.

Figure 2 shows the contents of a container and its relationship to the component executor(s) it manages.

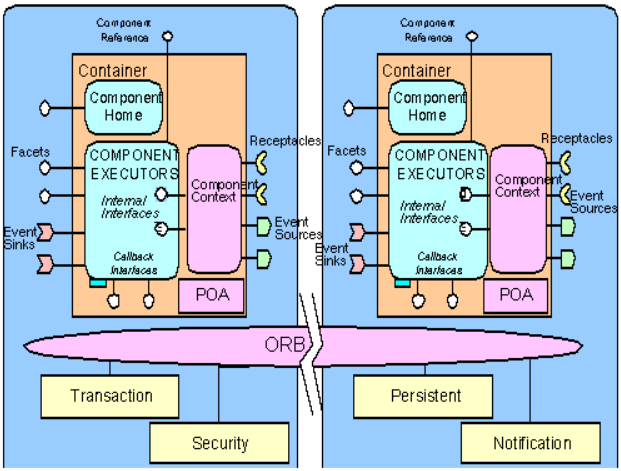

Figure 2: Structure of Components and Containers in CCM.

As shown in Figure 2, a container defines various operations that enable the component executors to access common middleware services (such as persistence, event notification, transaction, replication, load balancing, and security) and runtime strategies and policies (such as POA policies, event delivery strategies, and component usage categories). Each container is involved in initializing and providing the runtime context for the component executors it manages, including their connections to ports of other components and to common middleware services. Below, we describe the different types of interfaces and functionality provided by a container to realize a powerful runtime environment.

External APIs are interfaces the CCM container framework provides to clients of the component. There are two types of external APIs – **home** interfaces and **application** interfaces– that specify the contract between component developers and component clients. Home interfaces define operations that allow clients to obtain references to one of the application interfaces that the component implements. From the client's perspective, two design patterns [SCP] are supported by home interfaces: (1) factories for creating new component instances (i.e., executors) and (2) finder interfaces for locating existing components.

In our stock quoter application, the external APIs include the following home definitions:

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

and the following component definitions:

```
component StockDistributor supports Trigger {};
component StockBroker {};
```

all of which were explained in [Schmidt04b]. Our next column will show how the CIF uses the IDL 3.x definitions shown above along with other definitions (such as CIDL composition definitions) to generate C++ or Java code that implements the contract between developers of component clients and the components themselves. **Container APIs** within the CCM container framework consist of *internal interfaces* and *callback interfaces* that are used by component developers to build applications. Component developers use *internal interfaces* to access container facilities and implement component business logic. The following **SessionContext** is an example of an internal interface:

```
local interface CCMContext {
        Principal get_caller_principal ();
        CCMHome get_CCM_home ();
```

```
        boolean get_rollback_only () raises (IllegalState);
        Transaction::UserTransaction get_user_transaction () raises
(IllegalState);
        boolean is_caller_in_role (in string role);
        void set_rollback_only () raises (IllegalState);
};

local interface SessionContext : CCMContext {
    Object get_CCM_object () raises (IllegalState);
};
```

An internal interface like **SessionContext** serves as bootstrapper, offering executors access to services provided by the container. In contrast, *callback interfaces* are implemented by component developers within executors and are used by a container to call into the executor to achieve the required functionality. The following **SessionComponent** is an example of a callback interface:

```
local interface EnterpriseComponent { };

local interface SessionComponent : EnterpriseComponent {
    void set_session_context ( in SessionContext ctx) raises (CCMException);
    void ccm_activate () raises (CCMException);
    void ccm_passivate () raises (CCMException);
    void ccm_remove () raises (CCMException);
};
```

The **SessionComponent** is a type of **EnterpriseComponent** that is housed in a container and provides operations for associating context with a component (i.e., via the **set_session_context ()** operation) and managing the component's lifetime (i.e., via **ccm_activate()**, **ccm_passivate()**, and **ccm_remove()** operations). Our next column will illustrate how component developers can implement their executors by inheriting from the internal and callback interfaces shown above.

The container API specifies the contract between a component and the container that manages it. As shown above, these APIs are *locality constrained*; i.e., they use the **local interface** keyword in their IDL definitions. The **local interface** keyword informs the IDL compiler to suppress the generation of stubs and skeletons and instead generate language-specific mappings for locality constrained interfaces. Operations defined on local interfaces are therefore dispatched to process collocated local language-specific objects, rather than the standard mechanisms used to dispatch calls to servants registered within a POA.

The CCM specification defines two types of containers: (1) **session** containers, which define a framework for components using transient object references and (2) **entity** containers, which define a framework for components using persistent object references. These container types are analogous to the *session* and *entity* bean types in *Enterprise Java Beans* [EJB]. In CCM, developers can use different component types that are obtained by associating the above container types with different memory management policies and CORBA usage models. Below, we describe the various characteristics of containers supported by CCM.

Component category, which is the combination of the container API type (i.e., the component view) and the external API types (i.e., the client view). Table 1 illustrates the four different types of component categories supported by CCM.

COMPONENT CATEGORY	CONTAINER IMPL TYPE	CONTAINER TYPE	EXTERNAL TYPE
Service	Stateless	Session	Keyless
Session	Conversational	Session	Keyless
Process	Durable	Entity	Keyless
Entity	Durable	Entity	Keyfull

Table 1. Component Categories Supported by CCM.

Key features of the four component categories shown in Table 1 are explained below:

- The **service** component category is characterized by no state, no identity, and a behavior implemented as operations defined on the facet of the component. The lifespan of this category is equivalent to the lifetime of a single operation request and is therefore useful for operations, such as command objects, that have no duration beyond the lifetime of a single client interaction with them. The service component category provides a simple way to wrap existing procedural applications.

- The **session** component category is characterized by transient state, a nonpersistent identity (i.e., the identity could change if the same component is activated multiple times), and a behavior implemented as operations defined on the facet of the component. The lifespan of this category is specified using the lifetime policies, which are explained further below. A session component is useful for implementing our stock broker and stock distributor components, which require transient state for the lifetime of the interaction, but have no persistent state.

- The **entity** component category is characterized by persistent state, which is visible to the client and is managed by the implementation of entity container, a persistent identity that is architecturally visible to its clients through a *primary key* declaration in the home declaration, and behavior that may be transactional. As a fundamental part of the CCM architecture, entity components expose their persistent state to the client as a result of declaring a primary key value on their home declaration. An entity component can be used to implement an EJB entity bean.

- The **process** component category is characterized by persistent state that is not visible to the client (i.e., it's managed by the implementation of process container), a persistent identity that is visible to its clients only through user-defined operations on the home definitions, and a behavior that may be

transactional. The process component is intended to model objects that represent business processes (e.g., applying for a credit card or creating an order) rather than entities (e.g., stock distributor or accounts). The main difference between process components and entity components is that process components do not expose their persistent identity to clients (except through user-defined operations).

The desired component category can be selected at component development time by using the appropriate keyword in the Component Implementation Definition Language (CIDL) file, as discussed further below. Details of the contents and format of CIDL will be explained in our next column.

CORBA usage model, which defines the interactions between the container and other CORBA capabilities, such as the ORB Core, POA, and common object services. The following are the three CORBA usage models specified in CCM, based on the type of interaction between servants and POAs:

- **Stateless,** where the servants are activated in a POA that has a lifespan policy set to TRANSIENT and an identity uniqueness policy set to MULTIPLE_ID, allowing a servant to be shared among all requests from different clients for the same interface. This usage model requires that objects do not have any identity of their own and are stateless.

- **Conversational,** where the servants are activated in a POA that has a lifespan policy set to TRANSIENT and an identity uniqueness policy set to UNIQUE_ID. This usage model is useful for a wide range of systems, including our stock broker and stock distributor components.

- **Durable,** where the servants are activated in a POA that has a lifespan policy set to PERSISTENT and an identity uniqueness policy set to UNIQUE_ID. Persistent objects support either the factory design pattern or the finder design pattern [SCP], depending on the component category. Persistent objects support *self-managed* or *container-managed* persistence. Persistent objects can be used with the CORBA Persistent State Service [PSS] or a user-defined persistence mechanism. When the CORBA PSS is used, component lifetime management is aligned with the *PersistentId* defined by the CORBA persistent state service and the container supports the transformation of an *ObjectId* to and from a *PersistentId*. A *PersistentId* provides a persistent handle for a class of objects whose permanent state resides in a persistent store, such as a database or file.

The desired CORBA usage model can be selected by choosing the appropriate component category from Table 1 in a CIDL file. The relationship of the CORBA usage model with a component category implies a particular CORBA usage model.
Component lifetime management, which includes different policies for managing

the memory associated with components. Each component is associated with a programming language entity called a "servant," which represents the component within the POA and delegates certain operations to an executor, as described earlier. The lifetime of a component is therefore controlled via the lifetime of its servant within the POA. The following are the four lifetime policies associated with servants, and therefore executors:

- The **method** lifetime policy causes the container to activate the component upon every operation request and to passivate the component when an operation has completed. This policy limits memory consumption to the duration of an operation request, but incurs a greater cost of activation and passivation since the component is activated and passivated for every operation request. Moreover, there can be no state stored within the component, so this policy is mainly useful with the *stateless* CORBA usage model.

- The **container** lifetime policy causes the container to activate the component upon the first operation request and to leave it active until the container passivates it (possibly after thousands of requests are processed). Memory remains allocated until the container decides to reclaim it. This policy is useful with the *conversational* and *durable* CORBA usage models. We use this policy for the components in our stock broker application. Subsequent columns in this series will show how to select this CORBA usage model to develop our stock broker and stock distributor components.

- The **component** lifetime policy causes the container to activate the component on the first operation request and to leave it active until the component implementation requests it to be passivated. After the operation that requests the passivation completes, the container will passivate the component. Memory remains allocated until explicit application request, and is useful when component applications want to control the lifetime of the components themselves. This policy is commonly used with the *conversational* and *durable* CORBA usage models.

- The **transaction** lifetime policy causes the container to activate the component on the first operation request within a transaction and to leave it active until the transaction completes, at which point the component will be passivated. Memory remains allocated for the duration of the transaction. This policy is useful with the *conversational* and *durable* CORBA usage models, and is particularly useful for transaction systems, as its name implies.

The desired component lifetime management policy can be selected declaratively during the application deployment and configuration phase. We show an example of how this can be done via XML later below.

Context interface. In CCM, a *context* is an *internal interface* that provides a component executor with access to the common services provided by a container, such as access to the references of receptacles on the component, operations to publish events, access to the security credentials of the caller, and access to the **UserTransaction** interface that manages user-specified transactions as specified in the transaction service [CTS]. A session component can access its context via the **SessionContext** interface and an entity component can access its context via the **EntityContext** interface to gain access to the container and the underlying services it provides. The **SessionContext** and the **EntityContext** interfaces inherit from the **CCMContext** interface, as shown earlier. The context also serves as a "bootstrap" to the various services the container provides for the component. For example, in our **StockBroker** and **StockDistributor** components, the context will be used to push events to all the consumers and invoke operations on the receptacles, as we'll show in our next column.

The CCM Component Implementation Framework

CORBA 2.x helped simplify the life of application developers by shielding them from many complexities associated with distributed computing. For example, the left side of Figure 3 illustrates how an IDL 2.x compiler generates stub and skeleton code that automates marshaling and demarshaling tasks. The constructs supported by an IDL compiler are relatively limited, however. Developers of CORBA 2.x server applications therefore must still manually write a lot of code, e.g., defining the IDL interfaces themselves, implementing their servants, and writing all the code required to bootstrap and run the server. Some of this work is clearly unavoidable, i.e., *someone* has to implement the IDL interfaces and business logic of IDL operations. Conversely, other tasks performed by server application developers are unnecessarily tedious and error prone in CORBA 2.x.

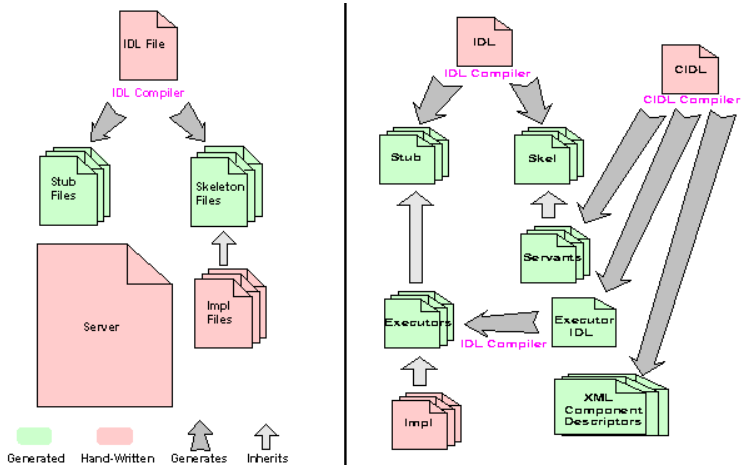

Figure 3: Automated Code Generation with IDL 2.x vs. IDL 3.x.

For example, server initialization and event loop code must be handwritten in CORBA 2.x, along with any support for introspection and navigation of object interfaces, which allows client applications to discover and traverse the operations and types defined in object interfaces. Likewise, CORBA 2.x server application developers must explicitly (1) keep track of dependencies their objects have on other objects and (2) manage the policies used to configure their POAs and manage object lifecycles. As a result, many CORBA 2.x server programs suffer from *ad hoc* design, code bloat, and inadequate reuse.

To address these problems, CCM defines the *Component Implementation Framework* (CIF), which consists of patterns, languages, and tools that simplify and automate the development of component implementations, which are called *executor implementations* in CCM lingo. The CIF framework can be viewed as an extension of the CORBA 2.x POA framework that shields server application developers from many complexities associated with programming POAs, particularly servant registration, servant activation and deactivation, and setting a consistent set of policies on the POAs. The capabilities provided by the CCM CIF can be implemented as a wrapper facade [POSA2] around the CORBA 2.x POA framework, without exposing the inherent complexities of the POA framework to component server application developers.

A key element of the CIF is the *Component Implementation Definition Language* (CIDL), which is a text-based declarative language that constitutes a key part of the CCM strategy for managing complex component-based applications by providing the capabilities described below.

1. Helps separate concerns by allowing developers to declaratively specify the

component category and persistence details (as in the case of **entity** and **process** components) from the actual component executor implementations. The CIDL description of a component implementation is an aggregate entity, of which the component itself may be a relatively small part. The term *composition* is used in a CIDL file to denote the set of artifacts that constitute the unit of component definition and implementation. Based on the composition definitions within a CIDL file, a CIDL compiler can work in conjunction with an IDL 3.x compiler to generate the skeleton glue code that associates the implementations of components and component homes with the containers that host them. A CIDL compiler also generates infrastructure skeleton glue code that can be extended by component developers and used by the CIF to (1) activate objects within the component server's POA, (2) manage the interconnection of a component's ports to the ports of other components, (3) provide implementations for operations that allow navigation of component facets, and (4) intercept invocations on executors to transparently enact various policies, such as component activation, security, transactions, load balancing, and persistence.

2. Increases the ratio of generated code to handwritten code. The right-hand side of Figure 3 illustrates how the CIF can help increase the ratio of generated to handwritten code by automatically synthesizing the following CCM-related entities:

- **Servants**, which unlike CORBA 2.x are not implemented by component developers in CCM. In CORBA 2.x, servants are implemented by server application developers, whereas in CCM servants are synthesized automatically by a CIDL compiler. These generated servants implement CCM introspection and navigation capabilities and hide low-level details of the ORB core and POA from the executors. Servants delegate certain requests (such as operations defined on component's supported interfaces and **get_*()** operations for facets) to the executors that are implemented by component developers.

- **Executor IDL**, which provides declarations of executors and container contexts as locality constrained interfaces with feature-specific operations, such as **push_*()** operations for events, **get_*()** operations for facets and receptacles, and accessors and mutators for attributes. The stubs generated from the executor IDL is inherited by executor classes that implement the component's business logic.

- **Context interfaces,** which extend container-specific interfaces as locality-constrained interfaces to aid component executor access and bootstrap the various container-specific features, such as access to references of the connected receptacles, access to operations that can publish events, and access to the security credentials of clients.

- **XML component descriptors**, which are used by the standard OMG Deployment and Configuration [D&C] specification to configure components together when an application is launched.

The remainder of this column shows examples of how each of these generated entities can be used in our stock quoter applications.

An example of a servant generated by a CIDL compiler for the **StockBroker** component is shown below (exception specifications have been omitted):

```
class StockBroker_Servant : public virtual POA_StockBroker
                            public virtual PortableServ
{
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
    // Operations for Receptacles interface.
    Components::Cookie *connect (const char *name,
                                 CORBA::Object_ptr connect
    CORBA::Object_ptr  disconnect (const char *name,
                                   ::Components::Cookie *c
    Components::ReceptacleDescriptions *get_all_receptacle
    // Operations for Events interface.
    Components::EventConsumerBase_ptr  get_consumer (const
    Components::Cookie *subscribe (const char *publisher_n
                                   ::Components::EventCons
    // more operations..
};
```

The generated **StockBroker_Servant** servant class shown above has operations for connecting/disconnecting receptacles with facets, subscribing/unsubscribing to published events, and introspecting/navigating components. A CIDL compiler will generate both the interface of the servant and implementations for all the methods shown above.

A sample of the executor IDL generated by a CIDL compiler for the stock broker is as follows:

```
local interface CCM_StockBroker : ::Components::Enterprise
       void push_notifier_in (in ::StockName e);
};
local interface CCM_StockBrokerHomeImplicit  {
    Components::EnterpriseComponent create ();
};
local interface CCM_StockBrokerHomeExplicit : Components::
local interface CCM_StockBrokerHome : CCM_StockBrokerHomeI
                                                  CCM_Stoc
local interface StockBroker_Exec : CCM_StockBroker,
                                   Components::SessionComp
local interface StockBrokerHome_Exec : ::CCM_StockBrokerHo
```

The executor IDL shown above provides component-specific (i.e., stock broker component-specific) internal and callback interfaces that must be passed through an IDL compiler to generate language-specific executor classes, such as **StockBroker_Exec** and **StockBrokerHome_Exec**. Component developers then inherit from these executor classes to implement their business logic. Our next column will show C++ examples of how to extend and implement the operations on the executor interfaces.

An example of the context interfaces generated by a CIDL compiler for our stock broker context is:

```
local interface CCM_StockBroker_Context : ::Components::Se
     StockQuoter get_connection_quoter_info_in ();
};
```

The locality constrained context interface above shows the new context generated for our stock broker component, which contains an operation to access the reference to a connected receptacle. Our next column will show how executors can use the operations on the context interfaces to access container-specific features, such as obtaining references to connected receptacles and using them to invoke operations (such as **StockQuoter::get_stock_info()**) and using the **push_*()** operations to publish stock names whose values have changed recently.

```
<corbacomponent>
<corbaversion>3.0
    <componentrepid repid="IDL:StockBroker:1.0"/>
      <homerepid repid="IDL:StockBrokerHome:1.0"/>
       <componentkind>
          <session>
            <servant lifetime="container"/>
          </session>
        </componentkind>
  <threading policy="multithread"/>
```

```
   <configurationcomplete set="true"/>
  <homefeatures name="StockBrokerHome"
      repid="IDL:StockBrokerHome:1.0">
  </homefeatures>
<componentfeatures name="StockBroker" repid="IDL:StockBrol
    <ports>
      <consumes  consumesname="notifier_in" eventtype="IDl
        <eventpolicy policy="normal"/>
      </consumes>
      <uses usesname="quoter_info_in" repid="IDL:StockQuot
    </ports>
  </componentfeatures>
<interface name="StockQuoter" repid="IDL:StockQuoter:1.0":
</corbacomponent>
```

This XML component descriptor specifies **StockBroker** component characteristics during design and deployment time. In addition to describing the component features, such as the port names and the corresponding repository IDs for the ports, the XML descriptor specifies default values for component category, lifetime management policy, event policy, and threading policy (see [CORBA3] for more details on these policies).

In CORBA 2.x applications, all the C++, IDL, and XML code shown above had to be written manually within the application to achieve the same level of functionality, 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.

Concluding Remarks

This column describes the CCM container architecture and Component Implementation Framework (CIF), focusing on how these capabilities can be applied to our stock quoter example. We showed how the CCM container and the CIF support common CORBA patterns for building, configuring, and deploying distributed applications. For example, with CORBA 2.x, the management of object and servant lifecycles is a common source of errors. We explained how the CCM container architecture and CIF together support common patterns of object and servant lifecycle management. As a result, application developers simply select the desired management patterns *declaratively* via configuration options and tools, and the component middleware platform handles all the gory details. The result of the separation of concerns and automation provided by CCM is an overall simplification of the development, customization, and deployment of component-based applications.

As we mentioned throughout this column, our next column will illustrate in detail how to apply all the CCM capabilities we discussed thus far to implement our stock quoter example using C++. As always, if you have any comments on this column or any previous one, please contact us at mailto:object_connect@cs.wustl.edu.

Acknowledgments

Thanks very much to Gan Deng, Tao Lou, and Jeff Parsons for their comments and improvements to this column.

References

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

[C++NPv2] D. Schmidt and S. Huston, "C++ Network Programming: Systematic Reuse with ACE and Frameworks," Addison-Wesley, 2003.

[CIAOSTAT] "CIAO Static Configuration Support for Real-Time Platforms", http://www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/CIAO/docs/static_ciao_index

[CTS] "CORBA Object Transaction Service," OMG Document formal/2003-09-02

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

[EJB] "Enterprise Java Beans Technology," http://java.sun.com/products/ejb/.

[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.

[PSDL] "Persistent State Definition Language," OMG Document formal/2002-06-65, June 2002.

[PSS] Persistent State Service 2.0 Specification, OMG Document orbos/99-07-07, July 1999.

[SCP] Markus Volter, Alexander Schmid, Eberhard Wolff: *Server Component Patterns: Component Infrastructures Illustrated with EJB*, John Wiley & Sons, Nov. 2002.

[Schmidt04a] D. Schmidt and S. Vinoski. “The CORBA Component Model, Part 1: Evolving Towards Component Middleware,” C/C++ Users Journal, Feb. 2004, http://www.cuj.com/documents/s=9039/cujexp0402vinoski/.

[Schmidt04b] D. Schmidt and S. Vinoski. “Object Interconnections: Defining Components with the IDL 3.x Types,” *C/C++ Users Journal*, April 2004, http://www.cuj.com/documents/s=9152/cujexp0404vinoski/.

Copyright © 2006 CMP Media LLC