CHAPTER 32

CIAO and CCM

32.1 Introduction

The OMG CORBA Component Model (CCM) (OMG Document formal/06-04-01) defines a specification for implementing *component middleware*. The Component-Integrated ACE ORB (CIAO) is TAO's implementation of the CCM specification.

The CORBA Component Model is a step in the longtime evolution of software engineering best practices towards higher levels of abstraction. CCM is a realization of the concept of composing software from reusable, pluggable *components*, assembled into an application at run time. When properly applied, component-based software development promotes improved software reuse, deployment flexibility, and programmer productivity.

32.1.1 Prerequisites

To better understand this chapter, the reader should be familiar with the content of the following chapters from this guide:

- Chapter 2, "Building ACE and TAO"

- Chapter 4, “The Makefile, Project, and Workspace Creator (MPC)”
- Chapter 5, “TAO IDL Compiler”
- Chapter 11, “Value Types”
- Chapter 13, “Local Interfaces”

32.1.2 What is a Component?

A component is a pluggable, self-contained software entity consisting of its own encapsulated business logic and data with clearly defined interfaces for collaboration. A component defines both the capabilities it *provides* and the services it *requires* as well as events it *publishes* and *consumes*, as illustrated by the diagram.

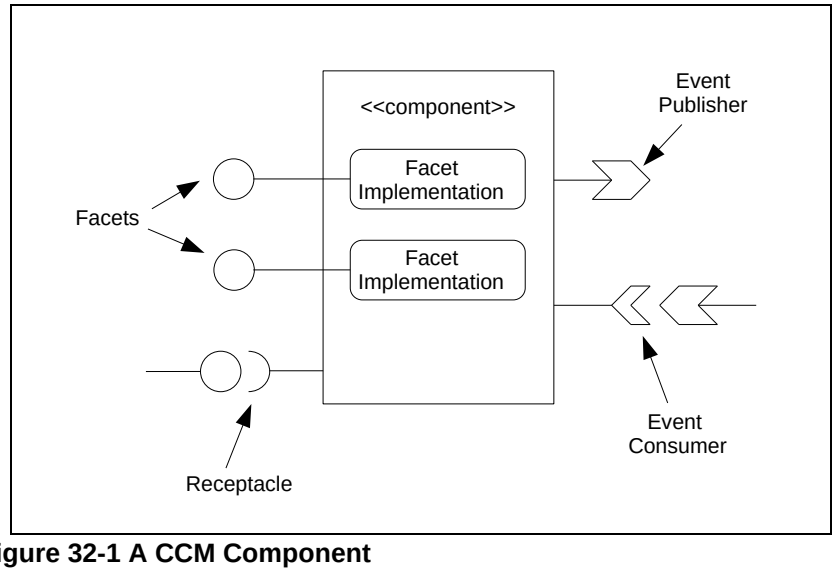

Figure 32-1 A CCM Component

The CORBA 2.x object model lacks the expressiveness required to create pluggable components. A CORBA 2.x IDL interface specifies a contract between a client and a server. That contract specifies what the server *provides* and what the client can *expect*. However, a great deal of information is missing from that IDL contract. A client or server has no formal mechanism to specify what it *requires*—namely, which IDL interfaces it depends upon to accomplish its tasks. These dependencies are hidden in the implementation code. Without knowledge of what each client or server requires, it is impossible to connect the clients and servers at run time in a generic way.

The CORBA Component Model includes new IDL constructs for expressing both the client and the server sides of component collaboration. This new edition of IDL is called *IDL3*. IDL3 is a superset of traditional CORBA IDL, or *IDL2*. The TAO IDL compiler accepts both IDL3 and IDL2 interface specifications.

A component defines its collaborations in terms of provided and required interfaces. An IDL3 component specification consists of *ports* that indicate how the component interacts with other components as both a client and a server. There are several types of ports providing various capabilities:

- A *facet* defines an IDL interface provided by a component. This is the server-side of the traditional IDL contract.
- A *receptacle* defines an IDL interface that is used by a component. The component may interact with that interface either through synchronous calls or through AMI. Facets and receptacles are connected via *assembly descriptors* that are processed at run-time.
- An *event source* or *publisher* defines an event type that is published by a component. CCM events are strongly typed, as our example will illustrate.
- An *event sink* or *consumer* defines an event type that is consumed by a component. Event sources and sinks are connected via assembly descriptors that are processed at run-time.
- An *attribute* provides a mechanism for configuring component properties at application start-up.

An application consists of several components packaged together and deployed at run time. A CCM-based application may consist of numerous binary component implementations implemented in several different programming languages communicating through CORBA.

The CCM specification defines a Component Implementation Framework (CIF) consisting of tools to simplify the implementation of components. The CIF uses the Component Implementation Definition Language (CIDL), through which a component developer defines a *composition* to describe a component's implementation details. A CIDL compiler generates a skeletal version of the component's C++ implementation, or *executor*. The developer is left to concentrate on application logic.

32.1.3 Component Deployment

A developer configures an application's component connections—facet to receptacle, event source to event sink—via descriptor files that a *component server* process loads at run time. The component server creates a *component container* to instantiate a component and connect it to any collaborating components through the appropriate ports. The component itself is deployed in a library that is dynamically loaded into the component server at run time.

CORBA is the underlying middleware infrastructure for the component containers. The container programming model is built on the Portable Object Adapter (POA). Components communicate through CORBA, assuring interoperability. The diagram illustrates the component container's relationship to the CORBA infrastructure.

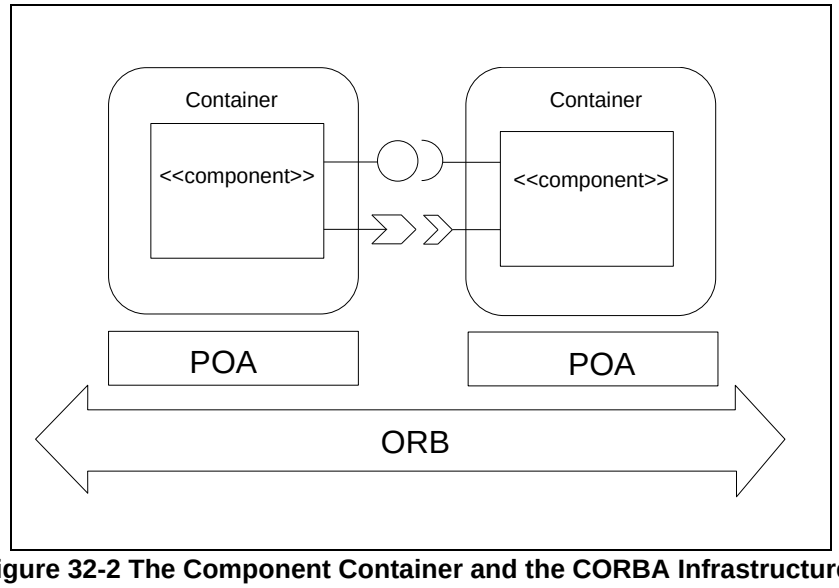

Figure 32-2 The Component Container and the CORBA Infrastructure

32.1.4 Summary of the CCM Programming Model

The CCM model of component programming extends the CORBA 2.x programming model in the following ways:

- A component specifies not only what it provides, but also what it requires.
- A component can provide multiple interfaces that are not related through inheritance.

- A component specifies events it publishes and consumes directly in its interface. Events are strongly typed value objects.
- An application developer assembles and deploys a component-based application by writing standard XML-based assembly and deployment descriptors. The component server reads the descriptors at run-time to load libraries and connect components, promoting loose coupling of component implementations.
- A component developer can add capabilities to an existing component without affecting existing clients by providing a new facet.
- A component developer does not need to have any direct interaction with the Portable Object Adapter. The component container interacts with the POA.
- A component developer does not write a `main()` function.
- The component container instantiates and destroys the component.
- The component server provides standard services such as event publication, transactions, persistent state, and security and enforces usage policies consistently.

A CCM client does not have to be component-aware. A CORBA 2.x client can bind to a component facet and interact with it without any knowledge that it is part of a CCM component.

32.1.5 Road Map

The following sections illustrate the CCM programming model with an example. The example illustrates the steps involved in developing a CIAO application by tracing the road map outlined in the diagram.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component's composition
  - Implement a C++ executor for each component and facet
- Describe the application's deployment
  - Describe each component's libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application
- Run the application

Figure 32-3 Road Map

As you can see, component development and deployment primarily consists of five phases: defining interfaces, implementing interfaces, describing the deployment, building, and running. Defining and implementing interfaces should be familiar to any CORBA developer. We will find that some of the steps in implementing an IDL3 interface are a bit different as we take advantage of the CCM programming model. Describing the deployment, involves defining XML descriptors to define how each component is composed from its libraries and how the components are connected together to form an application. In the fourth step, building, we create a set of dynamic libraries for each component. Finally, we run the application by executing component servers to load the dynamic libraries and connect the components together.

32.2 Example - The Messenger Application

Our CIAO example builds on the Messenger example used throughout this guide. The example's source code, build files, and XML descriptor files ares

in the `$CIAO_ROOT/examples/DevGuideExamples/Messenger` directory.

The CIAO Messenger example consists of three components: a Messenger, a Receiver, and an Administrator. The Messenger publishes message events and provides a history of all published messages. The Receiver subscribes to message events and retrieves the Messenger's message history. The Administrator controls the Messenger, starting and stopping publication and changing the attributes of what the Messenger publishes. The relationship between the three component types is demonstrated by the following component diagram:

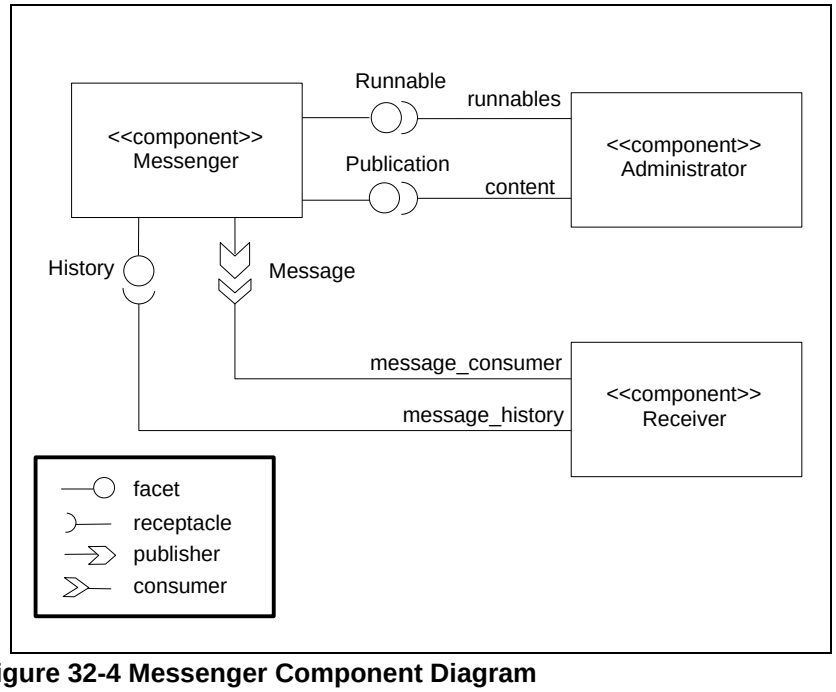

Figure 32-4 Messenger Component Diagram

The diagram illustrates that the Messenger component provides three *facets*, `Runnable`, `Publication`, and `History`. Each facet is an IDL interface. The Messenger component also publishes `Message` events. Each `Message` event is a value-based *event type*. The Receiver component has a *receptacle* that connects to the Messenger's `History` facet. It also consumes `Message` events published by the Messenger. Finally, the Administrator component has two

receptacles connected to the Messenger's `Runnable` and `Publication` facets, respectively.

The Messenger does not start publishing messages immediately at start-up. The Administrator connects to the Messenger's `Runnable` facet and invokes the `start()` operation on it to trigger message publication. Upon receiving a `start()` request, the `Messenger` publishes messages to all connected Receivers until the Administrator tells it to `stop()`. The "start publication" collaboration is illustrated in the following interaction diagram:

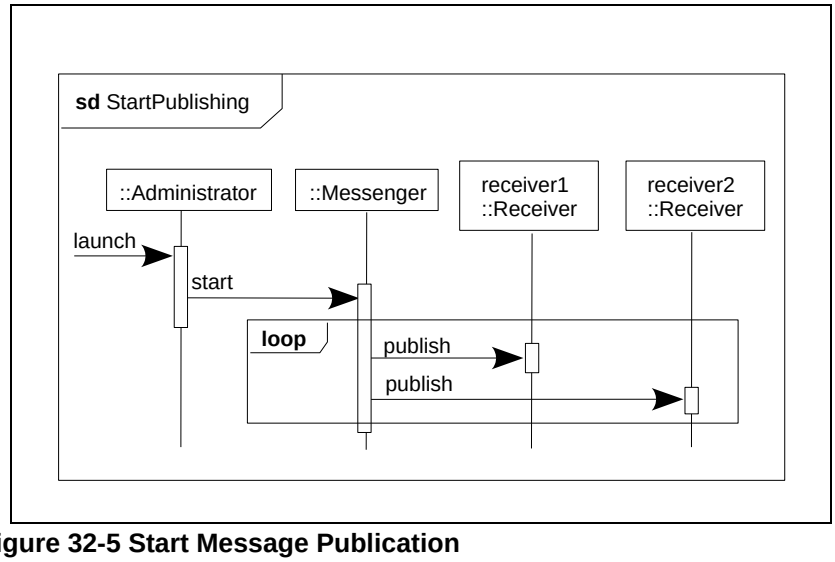

Figure 32-5 Start Message Publication

32.2.1 The Messenger Application's IDL Interfaces

The first task is to specify the Messenger application's interfaces using IDL. Or, more accurately, using IDL3. Next, we create a component type specification for the Messenger, Receiver, and Administrator. After that, we specify standard IDL interfaces for each of the facets provided by the

Messenger, namely the `History`, `Runnable`, and `Publication`. Finally, we create a `Message` event type whose instances the Messenger publishes.

- **Define an IDL interface for each component and its facets**
- Implement each component and its facets
  - Define each component's composition
  - Implement a C++ executor for each component and facet
- Describe the application's deployment
  - Describe each component's libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application
- Run the application

Figure 32-6 Road Map

32.2.1.1 The Messenger Component and Facets

The Messenger component provides facets that implement the `Runnable`, `Publication`, and `History` interfaces. It also publishes a `Message`. Each Receiver component consumes the `Messages` published by the Messenger and uses the `History` facet provided by the Messenger. The Administrator component uses the `Runnable` and `Publication` facets provided by the Messenger.

First, we specify IDL interfaces for the `Runnable` and `Publication` facets. Both of these are IDL2 interfaces that would be recognized by any CORBA client:

```
// file Runnable.idl
interface Runnable {
   void start();
   void stop();
};

// file Publication.idl
interface Publication {
  attribute string text;
  attribute unsigned short period;
};
```

We put each IDL interface in its own file as a programming convention. The `Runnable` interface provides control over starting and stopping of message publication. The `Publication` interface provides control over the published message text and the period, in seconds, between messages.

The Messenger component publishes `Message` events. We define the `Message` type using the new IDL3 keyword `eventtype`. An `eventtype` is an IDL value type that inherits from the abstract value type `Components::EventBase`. Our `Message` event type has three public string members: `subject`, `user`, and `text`.

```
// file Message.idl
#include <Components.idl>

eventtype Message {
  public string subject;
  public string user;
  public string text;
};
typedef sequence<Message> Messages;
```

We must include the IDL file `Components.idl` to use IDL3 keywords such as `eventtype`. Like any IDL value type, The `Message` event type may contain operations and a factory. For more information on value types, see Chapter 11.

However, we can simplify our event type implementation by restricting the contents of the event type to public data members. For such an event type, the IDL compiler generates a full event type implementation and automatically registers the event type factory for us. Therefore, we do not add operations or a factory to the event type.

The `History` facet contains operations to retrieve published `Message` events.

```
// file History.idl
#include <Components.idl>
#include <Message.idl>

interface History {
  Messages get_all();
  Message get_latest();
};
```

The implementation of the `History` facet must keep track of each message that it publishes for later retrieval by clients.

Finally, we declare the Messenger component. The Messenger declaration illustrates several of the new IDL3 keywords introduced for component-based programming.

```
// file Messenger.idl

#include <Components.idl>
#include <Runnable.idl>
#include <Publication.idl>
#include <Message.idl>
#include <History.idl>

component Messenger {
  attribute string subject;

  provides Runnable control;
  provides Publication content;

  publishes Message message_publisher;
  provides History message_history;
};

home MessengerHome manages Messenger {};
```

The Messenger's component specification must include `Components.idl` to make the IDL3 keywords available. It also includes IDL files for each of its three facets and for the `Message` events it publishes.

The keyword `component` is a new IDL3 keyword that is used to define a component.

```
component Messenger {
```

The component's definition can contain IDL attributes just like an IDL2 interface. However, the component's definition may not contain IDL operations.

The Messenger component contains one attribute, the subject.

```
  attribute string subject;
```

Despite of the fact that the `subject` attribute is writable it is not exposed to the Messenger's clients.

The Messenger component `provides` three facets.

```
  provides Runnable control;
  provides Publication content;
  provides History message_history;
```

Each facet is an IDL interface. A component uses the `provides` keyword to indicate the services that it *offers*. In the example, the Messenger's three facets are a `Runnable` facet called `control` for starting and stopping message publication, a `Publication` facet called `content` for control over the message content and publication period, and a `History` facet called `message_history` for access to all messages published by the component. There is no limit to the number of clients that may access the Messenger's facets.

The Messenger component `publishes` events:

```
  publishes Message message_publisher;
};
```

Recall that a `Message` is an event type. Published events are strongly typed. There is no limit to the number of subscribers for a published event. The Messenger component has neither direct knowledge of the event's subscribers nor knowledge of the underlying messaging mechanism.

A `publishes` port may publish to an unlimited number of subscribers. A second kind of publisher, called an *emitter*, is limited to one subscriber. An emitter uses the `emits` keyword instead of the `publishes` keyword. The CCM deployment framework enforces the emitter's limitation to one subscriber at deployment time. Aside from the keyword, the emitter's IDL syntax is the same as the publisher's. For example:

```
  emits Message message_publisher;
```

The `home`, called `MessengerHome`, manages the life cycle of the component.

```
home MessengerHome manages Messenger {};
```

Each component has a corresponding home. The component server uses the `home` to create and destroy component instances. Our Messenger's `home` is the simplest possible home, implicitly defining a `create()` operation. The `home` construct will be discussed in more detail later.

32.2.1.2 The Receiver Component

The Receiver component receives `Message` events from the Messenger and retrieves the message `History` from the `Messenger`.

```
// file Receiver.idl
#include <Components.idl>
#include <Messsage.idl>
#include <History.idl>

component Receiver {
  consumes Message message_consumer;
  uses History message_history;
};

home ReceiverHome manages Receiver {};
```

The Receiver does not expose any facets, but instead indicates what it *requires* via a `uses` specification. The Receiver *uses* a `History` facet, and *consumes* `Message` events. The specification of not only what a component *offers* but also what it *requires* is a significant step forward, as it enables connection of components at deployment time. Both of these Receiver receptacles are connected to corresponding facets on the Messenger component at deployment.

The Receiver also has a `home`, `ReceiverHome`, which is responsible for creating and destroying Receiver component instances. Again, this is the simplest possible home declaration.

```
home ReceiverHome manages Receiver {};
```

Note *The Receiver's IDL file does not have a dependency on the Messenger. The Receiver knows about `Message` and `History`, but it does not need to know anything about the component that provides those services. A component may depend on IDL interfaces and event types, but it need not depend on other components.*

32.2.1.3 The Administrator Component

Finally, the third component type, an Administrator, triggers the Messenger's event publication and controls the period of its publication and the text that it publishes.

```
// file Administrator.idl
#include <Components.idl>
#include <Runnable.idl>
#include <Publication.idl>

component Administrator {
  uses multiple Runnable runnables;
  uses multiple Publication content;
};

home AdministratorHome manages Administrator {};
```

The Administrator uses both the `Runnable` and `Publication` facets provided by the Messenger. These two receptacles are later connected to corresponding facets provided by the Messenger. The Administrator's home is responsible for creating and destroying the Administrator component instance at run time.

The `uses multiple` keyword on the Administrator's `runnables` and `content` receptacles indicates that the Administrator can connect to more than one `Runnable` facet and more than one `Publication` facet. These facets may be provided by the same component or by different components; the Administrator does not need to know. In our sample deployment the Administrator connects to one `Runnable` facet and one `Publication` facet, both from the same Messenger component.

Note *The Administrator's IDL file does not have a dependency on the Messenger. The Administrator knows about* `Runnable` *and* `Publication`*, but it does not need to know anything about the component that provides those services.*

The Administrator, like all components, has a `home` to manage its life cycle:

```
home AdministratorHome manages Administrator {};
```

The homes in our example are the simplest possible. The default home contains a factory that acts like a default constructor. It is possible to override that factory and provide parameters to be passed into it.

To summarize, we've been exposed to several new IDL3 keywords:

Table 32-1 IDL3 Keywords

IDL3 Keyword	Description
`component`	Declares a component that can provide and use facets, publish and consume events
`provides`	Declares an IDL interface that the component offers; the interface defines a service offered
`uses`	Declares an IDL interface that the component requires
`uses multiple`	Declares that the component can connect to one or more instances of the required interface
`eventtype`	Declares an event type that the component publishes; the `eventtype` is an IDL `valuetype`
`publishes`	Declares that the component publishes instances of an event type to a potentially unlimited number of consumers
`emits`	Declares that the component publishes instances of an event type to exactly one consumer
`consumes`	Declares that the component expects the event type to be published to it by one or more publishers
`home`	Declares an interface used by the component container to manage the component's life cycle
`manages`	Declares which component is managed by the `home`

32.2.2 Implementing the Components

The CORBA Component Model specification defines a Component Implementation Framework (CIF) consisting of tools to simplify and automate the implementation of components. A significant part of the CIF is the Component Implementation Definition Language (CIDL), through which a component developer provides implementation details for each component type. The CIDL compiler compiles the CIDL files and generates a significant portion of the C++ implementation code. The developer is left to concentrate on application logic.

We write CIDL files for the Messenger, Receiver, and Administrator component types. Each CIDL file contains a component *composition*.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - **Define each component's composition**
  - Implement a C++ executor for each component and facet
- Describe the application's deployment
  - Describe each component's libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application
- Run the application

Figure 32-7 Road Map

32.2.2.1 The Messenger Composition

The primary entity of a CIDL file is a `composition`. A `composition` describes how a `component` is connected to its `home`. A `component` can be instantiated by more than one `home`; the `composition` designates the `home` responsible for the `component`.

The declaration of the Messenger's composition follows:

```
// file Messenger.cidl
#include <Messenger.idl>

composition session Messenger_Impl
{
    home executor MessengerHome_Exec
    {
        implements MessengerHome;
        manages    Messenger_Exec;
    };
};
```

The `session` is the component's *life cycle category*. A `session` composition provides transient object references and maintains its transient state for the lifetime of the session. Once the component is destroyed, its object references

are invalidated and its state is lost. The other valid composition life cycle categories are `entity`, `service`, and `process`. They are discussed later.

The name of the composition is `Messenger_Impl`. The CIDL compiler generates its implementation code into a C++ namespace called `Messenger_Impl`. The composition can have any name; it is customary to end the name with `_Impl`.

An implementation of a component or a home is called an *executor*. A CCM developer implements an executor rather than a *servant*. The CIDL compiler generates two abstract C++ executor classes, one for the `component` and one for its `home`, using the names `Messenger_Exec` and `MessengerHome_Exec` specified in the CIDL composition. The Messenger executors may have any name; it is customary to end the each with the suffix `_Exec`.

```
home executor MessengerHome_Exec
{
    implements MessengerHome;
    manages    Messenger_Exec;
};
```

The `home executor` defines which `home` is used to manage the life cycle of the Messenger component.

The `implements` declaration declares which of the component's homes manages the component's life cycle. The Messenger component only has one home, the `MessengerHome`, so that is the `home` we'll use. Note that we don't need to indicate that the `MessengerHome` manages the Messenger component; that relationship is defined in the `MessengerHome`'s declaration.

The component developer overrides pure virtual methods in the generated executor classes to provide the component implementation. The CIDL compiler can optionally generate a default implementation of each C++ executor class. By default, it appends `_i` to the executor class name. The default implementation of the Messenger executor is `Messenger_exec_i`, and the default implementation of the `MessengerHome` executor is `MessengerHome_exec_i`. The component developer fills the application logic into the generated Messenger executor implementation. The CIDL compiler generates a full implementation for the `MessengerHome`'s executor, so no developer intervention is required.

32.2.2.2 The Receiver and Administrator Compositions

The Receiver and Administrator compositions are similar to the Messenger composition.

```
// file Receiver.cidl
#include <Receiver.idl>

composition session Receiver_Impl
{
    home executor ReceiverHome_Exec
    {
        implements ReceiverHome;
        manages    Receiver_Exec;
    };
};
```

The Receiver's composition is called `Receiver_Impl`, and it's home executor implements the `ReceiverHome`. CIDL compiler generates an abstract executor class for the `ReceiverHome` called `ReceiverHome_Exec` and an abstract executor class for the Receiver component called `Receiver_Exec`. Optionally, the CIDL compiler can generate default implementations of the two executors.

```
// file Administrator.cidl
#include <Administrator.idl>

composition session Administrator_Impl
{
    home executor AdministratorHome_Exec
    {
        implements AdministratorHome;
        manages    Administrator_Exec;
    };
};
```

The Administrator's composition is called `Administrator_Impl`, and its home executor implements the `AdministratorHome`. The CIDL compiler generates an abstract executor class for the `AdministratorHome` called `AdministratorHome_Exec` and an abstract executor class for the Administrator component called `Administrator_Exec`.

To summarize, we've been exposed to several new CIDL keywords:

Table 32-2 CIDL Keywords

CIDL Keyword	Description
`composition`	Declares a set of entities that work together to manage the component's life cycle and implement the component's behavior
`session`	A component category characterized by transient state and transient object identity
`service`	A component category characterized by objects having no duration beyond the lifetime of a single client interaction
`entity`	A component category characterized by persistent state that is visible to the user and persistent object identity
`process`	A component category characterized by persistent state that is not visible to the user and persistent object identity
`executor`	Declares the name of the abstract component home executor class
`implements`	Declares the home that manages the component
`manages`	Declares the name of the abstract component executor class

32.2.3 Compiling the IDL and CIDL

We compile the IDL files with TAO's IDL compiler. The TAO IDL compiler recognizes IDL3 constructs such as `component` and `eventtype`. Additional information on compiling the Messenger's IDL files is contained in 32.2.6.

We compile the CIDL files with CIAO's CIDL compiler. Additional information on compiling the Messenger's CIDL files is also contained in 32.2.6. The CIDL Compiler Reference in 32.5 contains more extensive information on using the CIDL compiler.

This section concentrates on the output of the IDL and CIDL compilers rather than the mechanics of executing the IDL and CIDL compilers.

The CIDL compiler can generate most of the code for home, component, and facet executor implementations through its `--gen-exec-impl` command-line option. For each component, home, or facet it generates a C++ class that inherits from a generated abstract executor class, leaving the component developer to fill in the application logic.

The diagram shows the files that the CIDL compiler generates when it compiles `Messenger.cidl`.

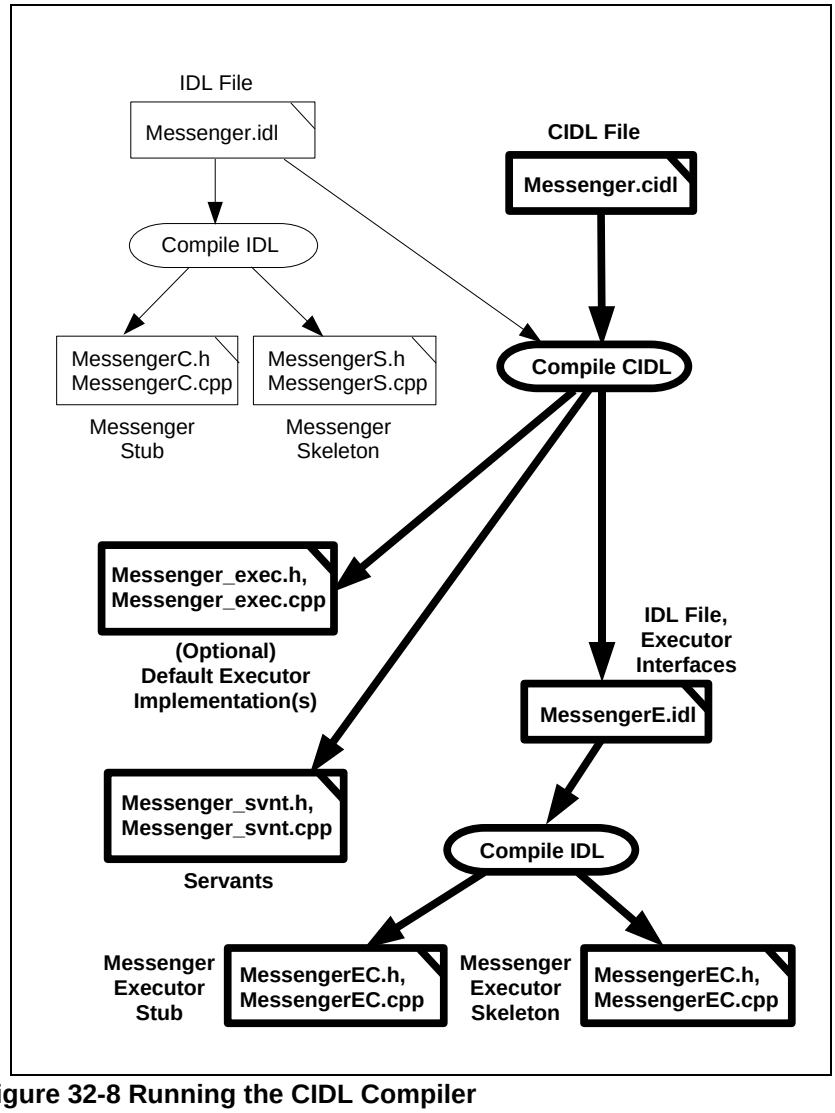

Figure 32-8 Running the CIDL Compiler

We show both `Messenger.cidl` and `Messenger.idl` as inputs to the CIDL compiler because the `Messenger.cidl` file includes `Messenger.idl`.

The CIDL compiler generates an IDL file, `MessengerE.idl`, containing local interfaces for the Messenger's component, home, and facet executors. We compile this IDL file with the IDL compiler to generate an abstract C++ executor class for each component and facet. Each component, home, and facet executor implementation implements one of the local interfaces declared in `MessengerE.idl`.

The CIDL compiler also generates complete C++ header and implementation files for the servant classes. There is a servant class for each component, home, and facet executor class. The CCM developer does not directly instantiate servants; instead, the component container instantiates servants and registers them with the Portable Object Adapter automatically.

The CIDL compiler optionally generates default component, home, and facet executor implementation classes in files called `Messenger_exec.h` and `Messenger_exec.cpp`. Those files contain definitions for five classes: `Messenger_exec_i`, `MessengerHome_exec_i, Runnable_exec_i`, `Publication_exec_i`, and `History_exec_i`. The latter three classes are executors for the Messenger's `Runnable`, `Publication`, and `History` facets. For safety, copy the `Messenger_exec.h` and `Messenger_exec.cpp` files to something like `Messenger_exec_i.h` and `Messenger_exec_i.cpp`. You may also want to break the implementations for `History_exec_i`, `Runnable_exec_i`, etc., into different header and implementation files as we've done in our sample code.

The diagram illustrates the Messenger executor's classes.

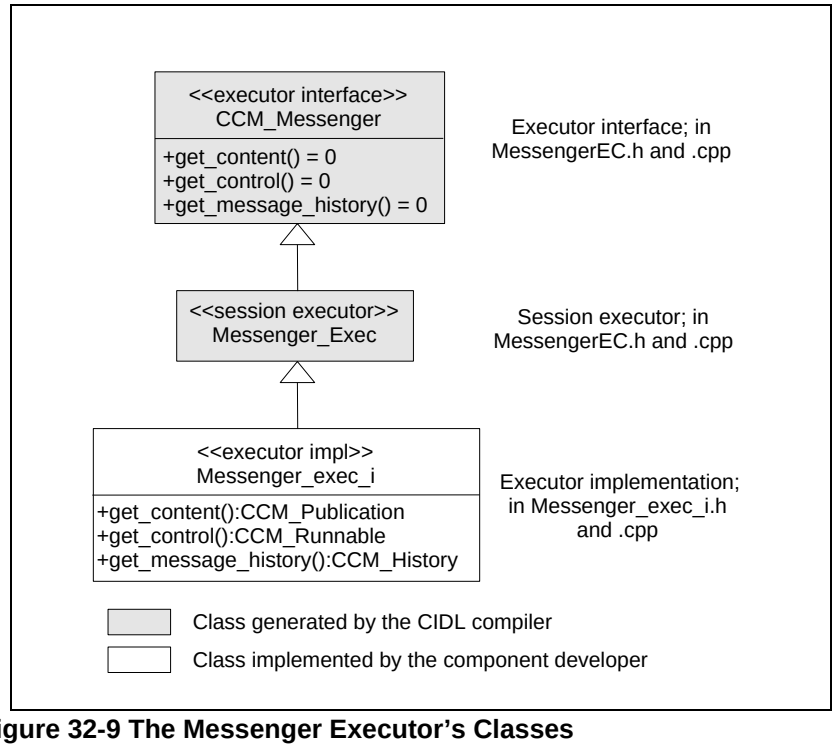

Figure 32-9 The Messenger Executor's Classes

The table summarizes the Messenger's executor implementation classes.

Table 32-3 Executor Implementation Classes

Executor Implementation Class	Description
`Messenger_exec_i`	Implements the `Messenger` component
`MessengerHome_exec_i`	Implements the `MessengerHome`
`Runnable_exec_i`	Implements the `Runnable` facet
`Publication_exec_i`	Implements the `Publication` facet
`History_exec_i`	Implements the `History` facet

32.2.4 Implementing the Executors

The CIAO CIDL compiler generates an empty implementation of each component and facet executor. In the following sections, we implement the facet executors for the `Runnable`, `Publication`, and `History` facets and the component executors for the Messenger, Receiver, and Administrator components.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component's composition
  - **Implement a C++ executor for each component and facet**
- Describe the application's deployment
  - Describe each component's libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application
- Run the application

Figure 32-10 Road Map

32.2.4.1 The Runnable Facet Executor

The `Runnable` facet is provided by the Messenger component and permits a client to start and stop message publication. The component diagram highlights the role of the Messenger's `Runnable` facet.

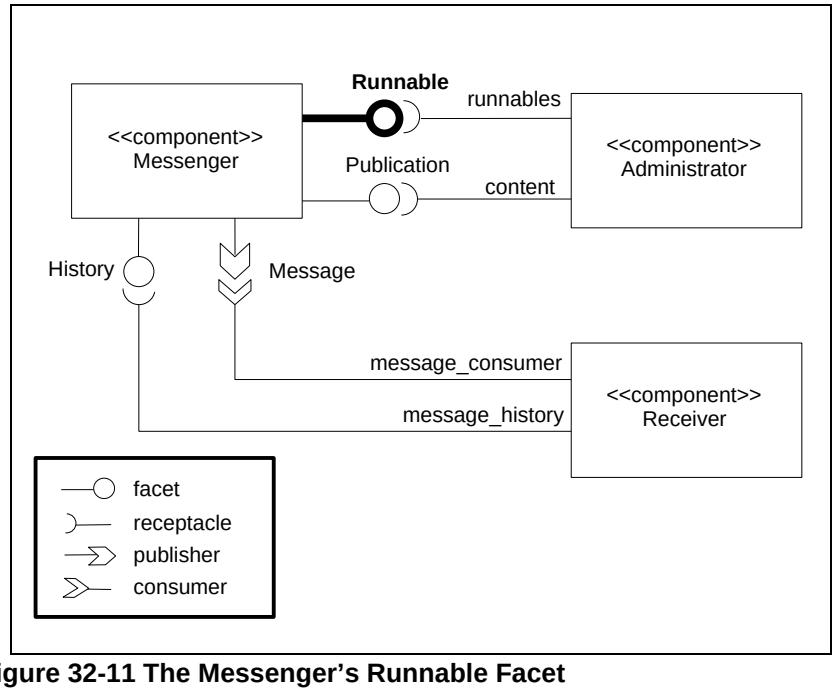

Figure 32-11 The Messenger's Runnable Facet

Recall that the `Runnable` IDL interface is as follows:

```
// file Runnable.idl
interface Runnable {
   void start();
   void stop();
};
```

The CIDL compiler generates a default `Runnable` executor with empty implementations of `start()` and `stop()`. The class diagram illustrates the `Runnable` executor's class hierarchy.

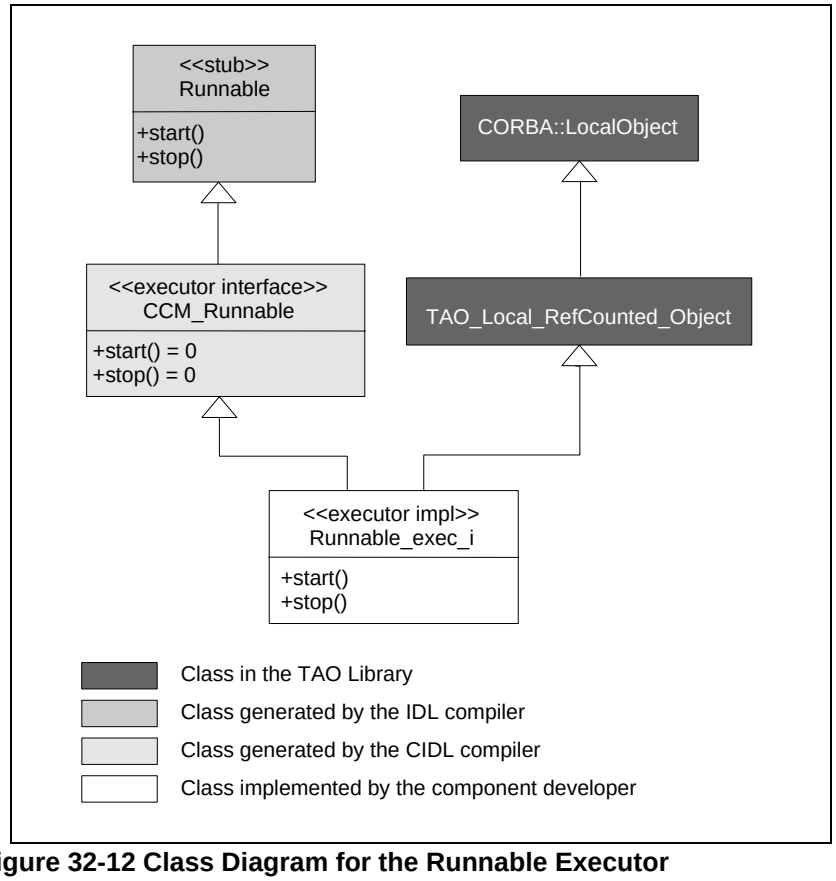

Figure 32-12 Class Diagram for the Runnable Executor

The IDL compiler generates a `Runnable` stub. The CIDL compiler generates an abstract executor base class, `CCM_Runnable`, and optionally generates an empty executor implementation, `Runnable_exec_i`. The CIDL compiler generates a default constructor, a destructor, and a virtual method for each of `Runnable`'s IDL operations and attributes.

Note *For each IDL interface "MyInterface" that is a facet of a component, the CIDL compiler generates an abstract facet executor class called "CCM_MyInterface."*

An *executor* is a local CORBA object. Its generated implementation class also inherits from ::`CORBA::LocalObject`. Additional information on local objects can be found in Chapter 13.

The Messenger component only publishes messages when it can acquire the `Runnable` executor's `run_lock`. If the Messenger cannot acquire the run lock, it blocks waiting for it to be released. A client of the `Runnable` facet controls the run lock via the `start()` and `stop()` operations.

The `Runnable` executor implementation follows. Changes to the CIDL-generated empty executor implementation are in **bold**:

```
// file Runnable_exec_i.h

#include "Messenger_svnt.h"
#include "tao/LocalObject.h"
#include <ace/Thread_Mutex.h>

namespace Messenger_Impl
{
  class MESSENGER_EXEC_Export Runnable_exec_i
    : public virtual ::CCM_Runnable,
      public virtual ::CORBA::LocalObject
  {
  public:
    Runnable_exec_i (void);
    virtual ~Runnable_exec_i (void);

    // Operations from ::Runnable

    virtual void start ();

    virtual void stop ();

    ACE_Thread_Mutex& get_run_lock();

  private:
    ACE_Thread_Mutex run_lock_;
  };
}
```

The included Messenger_svnt.h header file contains the servant class definitions for the Runnable, Publication, History, Messenger, and MessengerHome. A component developer does not implement servant classes; instead, the CIDL compiler generates servant classes and the component container automatically instantiates them at run time. A component developer implements executor classes that have no relationship to the server's POA. The automatically-generated servant class delegates its execution to the developer-written local executor object.

The executor implementation inherits from the generated abstract executor base class, CCM_Runnable, and from ::CORBA::LocalObject. CCM_Runnable, in turn, inherits from the generated Runnable stub class that the client uses. Thus, the executor implements the Runnable interface generated by the IDL compiler. The start() and stop() operations are declared as pure virtual methods in the CCM_Runnable class, forcing the executor to implement them.

Note the lack of inheritance from a POA_Runnable class; instead, the CIDL compiler generates a Runnable_Servant class for us.

The inheritance from ::CORBA::LocalObject enforces two behaviors: first, the executor is a ::CORBA::LocalObject, meaning that it can only be used from within the server process; second the executor has reference counting, meaning that the inherited _add_ref() and _remove_ref() operations must be used to manage the executor's memory.

Our Runnable implementation contains a private ACE_Thread_Mutex lock and a public accessor method to retrieve it. The Messenger acquires this run_lock before publishing each message and releases it after publishing each message. If the Messenger cannot acquire the run_lock, it blocks until the lock is released. A Runnable client can acquire and release the run_lock through the start() and stop() operations. In this way, a client can control whether or not the Messenger publishes any messages.

The CIDL compiler also generates an empty, default implementation of the Runnable_exec_i class. We implement a constructor, the start() and stop() operations, and an accessor for the mutex lock. Changes to the CIDL-generated default executor implementation code are in **bold**.

```
// file Runnable_exec_i.cpp

#include "Messenger_exec_i.h"
#include "ciao/CIAO_common.h"
```

```
namespace Messenger_Impl
{
  //=================================================================
  // Facet Executor Implementation Class:   Runnable_exec_i
  //=================================================================

  Runnable_exec_i::Runnable_exec_i (void)
  {
    // initially, the Messenger does not publish
    this->stop();
  }

  Runnable_exec_i::~Runnable_exec_i (void)
  {
  }

  // Operations from ::Runnable

  void
  Runnable_exec_i::start ()
  {
    // Your code here.

    // allows the Messenger to acquire the lock and publish
    this->run_lock_.release()
   }

  void
  Runnable_exec_i::stop ()
  {
    // Your code here.

    // prevents the Messenger from acquiring the lock; can't publish
    this->run_lock_.acquire()
  }

  ACE_Thread_Mutex&
  Runnable_exec_i::get_run_lock()
  {
    return this->run_lock_;
  }
}
```

The `Runnable` executor creates an `ACE_Thread_Mutex` lock for the Messenger to acquire in its event loop before publishing messages. If the Messenger can't acquire the lock, then it does not publish messages. This agreement between the `Runnable` executor and the Messenger executor

controls the suspension and resumption of message publication. Initially, the `Runnable` executor holds the lock. The implementations of `start()` and `stop()` release and acquire the lock, respectively. The `get_run_lock()` accessor exposes the lock to the Messenger executor.

The CIDL compiler also generates executor implementations for the `Publication` and `History` interfaces and the Messenger, Receiver, and Administrator components.

32.2.4.2 The Publication Facet Executor

The `Publication` facet is provided by the Messenger component and permits a client to modify the text published and the period (in seconds) between published messages.

Recall that the `Publication` IDL interface is as follows:

```
interface Publication {
  attribute string text;
  attribute unsigned short period;
};
```

The CIDL compiler generates an empty implementation of the `Publication` executor. We add private class attributes to keep track of the message subject, text, and period. Changes to the CIDL-generated code are in **bold**.

```
#include <string>
#include <ace/Thread_Mutex.h>

namespace Messenger_Impl
{
  class MESSENGER_EXEC_Export Publication_exec_i
  : public virtual ::CCM_Publication,
    public virtual ::CORBA::LocalObject
  {
  public:
    Publication_exec_i (const char* text,
                        CORBA::UShort period);
    virtual ~Publication_exec_i (void);

    // Operations from ::Publication

    vrtual char* text ();

    virtual void text (const char* text);
```

```
    virtual CORBA::UShort period ();

    virtual void period (CORBA::UShort period);

  private:
    std::string text_;
    CORBA::UShort period_;

    ACE_Thread_Mutex lock_;
  };
}
```

The pattern is similar to the `Runnable` executor's. The `Publication_exec_i` executor inherits from both the generated `CCM_Publication` class and TAO's `::CORBA::LocalObject` class. The accessor and modifier for the `text` and `period` attributes are declared as pure virtual methods in the `CCM_Publication` class, forcing us to implement them in our executor.

The `text_` and `period_` class members hold information about the publication. Because clients can modify the text and period, the executor uses an `ACE_Thread_Mutex` lock to protect them from simultaneous access. We have to assume that a provided facet might be accessed by multiple threads at the same time.

The implementation of the `Publication` executor follows. Again, changes to the CIDL-generated default executor implementation code are in **bold**.

```
#include "Publication_exec_i.h"
#include "ciao/CIAO_common.h"

namespace Messenger_Impl
{
  //==================================================================
  // Facet Executor Implementation Class:   Publication_exec_i
  //==================================================================

  Publication_exec_i::Publication_exec_i (const char* text,
                                          CORBA::UShort period)
    : text_( text ),
      period_( period)
  {
  }

  Publication_exec_i::~Publication_exec_i (void)
  {
  }
```

```
  // Operations from ::Publication

  char*
  Publication_exec_i::text ()
  {
    ACE_Guard<ACE_Thread_Mutex> guard(this->lock_);
    return CORBA::string_dup( this->text_.c_str() );
  }

  void
  Publication_exec_i::text (const char* text)
  {
    ACE_Guard<ACE_Thread_Mutex> guard(this->lock_);

    this->text_ = text;
    ACE_DEBUG((LM_INFO, ACE_TEXT("publication text changed to %C\n"), text ));
  }

  CORBA::UShort
  Publication_exec_i::period ()
  {
    ACE_Guard<ACE_Thread_Mutex> guard(this->lock_);
    return this->period_;
  }

  void
  Publication_exec_i::period (CORBA::UShort period)
  {
    ACE_Guard<ACE_Thread_Mutex> guard( this->lock_ );

    if ( period > 0 ) {
      this->period_ = period;
      ACE_DEBUG((LM_INFO,
               ACE_TEXT("publication period changed to %d seconds\n"), period ));
    } else {
      ACE_DEBUG((LM_INFO,
                 ACE_TEXT("ignoring a negative period of %d\n"), period ));
    }
  }
}
```

The `Publication` executor contains text and a publication period. Because the client may change either the text or publication period, we protect both with a mutex lock. The constructor sets the text and period values. The attribute accessors and modifiers are straightforward, protecting those values with the mutex lock. The period modifier ensures that the new period is a positive number.

32.2.4.3 The History Facet Executor

The Messenger component stores the messages that it publishes in a `History` executor. The `History` executor contains an STL list of published `Message` events. We protect access to the list with an `ACE_Thread_Mutex` lock because multiple threads might add to and query the `History` list simultaneously. We must assume that simultaneous access will happen.

Recall that the `History` IDL interface is as follows, where `Message` is an event type:

```
#include <Message.idl>

interface History {
  Messages get_all();
  Message get_latest();
};
```

The CIDL-generated `History` executor implementation follows, with our changes in **bold**.

```
#include "Messenger_svnt.h"
#include "Messenger_exec_export.h"
#include "tao/LocalObject.h"

#include <list>
#include <ace/Thread_Mutex.h>

namespace Messenger_Impl
{
  class MESSENGER_EXEC_Export History_exec_i
  : public virtual ::CCM_History,
    public virtual ::CORBA::LocalObject
  {
  public:
    History_exec_i (void);
    virtual ~History_exec_i (void);

    // Operations from ::History

    virtual ::Messages* get_all ();

    virtual ::Message* get_latest ();

    void add(::Message* message);

  private:
    ACE_Thread_Mutex lock_;
```

```
    typedef std::list<::Message_var> MessageList;
    MessageList messages_;
  };
}
```

We add a mutex lock and an STL `list` of messages as private class attributes. The lock protects the message list from simultaneous access by multiple threads. The STL list stores `Message_var`s to properly handle reference counting and memory ownership. The Messenger component uses the public `add()` method to add messages to the history as it publishes them.

The `History` executor implementation follows. As always, changes to the CIDL-generated default executor implementation code are in **bold**. Comments are interspersed with the code.

```
namespace Messenger_Impl
{
  //=================================================================
  // Facet Executor Implementation Class:   History_exec_i
  //=================================================================

  History_exec_i::History_exec_i (void)
  {
  }

  History_exec_i::~History_exec_i (void)
  {
  }

  // Operations from ::History
```

The implementation of the history's `get_all()` operation is the most challenging. It converts the STL list of `Message_var`s into an IDL sequence of `Messages`.

```
  ::Messages*
  History_exec_i::get_all ()
  {
    // Your code here.

    ACE_Guard<ACE_Thread_Mutex> guard(this->lock_);

    ACE_DEBUG((LM_INFO, ACE_TEXT("History_i::get_all\n") ));

    // create a Messages sequence, set its length
```

```
  ::Messages* retval = new ::Messages();
  retval->length( this->messages_.size() );

  // iterate through the MessageList, copying messages into the return sequence
  CORBA::ULong i = 0;
  for ( MessageList::iterator messageItr = this->messages_.begin();
        messageItr != this->messages_.end();
        ++messageItr )
    {
      // because the MessageList contains Message_vars, reference counting
      // upon assignment into the sequence is handled properly for us.
      (*retval)[i++] = *messageItr;
    }
  return retval;
}
```

The `get_all()` operation creates a new `Messages` sequence, setting its length. It then iterates through the internal STL list of `Message_var`, adding each `Message` to the sequence. Because the STL list stores `Message_vars` the assignment of each Message from the STL list to the sequence handles memory management properly for us by incrementing the reference count on each returned `Message`.

The `get_latest()` operation simply retrieves the last `Message` added to the list and returns it.

```
::Message*
History_exec_i::get_latest ()
{
  // Your code here.

  ACE_Guard<ACE_Thread_Mutex> guard(this->lock_);

  ACE_DEBUG((LM_INFO, ACE_TEXT("History_i::get_latest\n") ));

  // just get the last message from the history.  because the MessageList
  // contains Message_vars, _var to _var assigmnent handles the reference
  // counting properly for us.
  ::Message_var retval = this->messages_.back();
  return retval._retn();
}
```

We extract the last `Message` into a `Message_var` and return it with the `_retn()` operation to handle the reference counting of the Message properly. We give up ownership of the `Message` when we return it, but we also want to

keep the `Message` in the internal list. The reference counting handles that for us.

The `Messenger` calls the local `add()` method to store published `Messages`.

```
  void History_exec_i::add (::Message* message)
  {
    ACE_Guard<ACE_Thread_Mutex> guard(lock_);

    // bump up the reference count; we don't own the parameter.
    // the _var in the STL list takes ownership of the "copy"
    message->_add_ref();
    this->messages_.push_back( message );
  }
}
```

It increments the reference count of the `Message` and stores it in the class's STL list. If we do not increment the reference count, then the STL list would attempt to take ownership of a `Message` that it does not own.

The `get_all()` and `get_latest()` operations are exposed to clients through the `History` interface. The `add()` method is not part of the IDL interface and is visible only through the Messenger implementation.

32.2.4.4 The Messenger Component Executor

The Messenger component provides the `Runnable`, `Publication`, and `History` facets and publishes `Message` events. It delegates much of its work to the `Runnable`, `Publication`, and `History` executors.

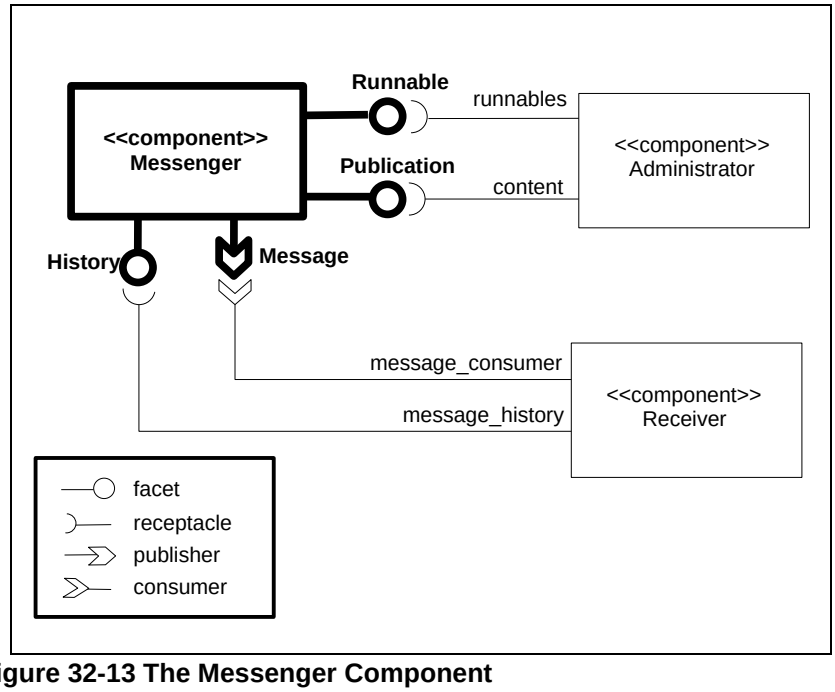

Figure 32-13 The Messenger Component

Recall that the Messenger's IDL specification is as follows:

```
component Messenger {
  attribute string subject;

  provides Runnable control;
  provides Publication content;

  publishes Message message_publisher;
  provides History message_history;
};

home MessengerHome manages Messenger {};
```

The Messenger's component executor contains a `get_<facet_name>()` operation for each of its three provided facets to expose the facet to the component container. As a general rule, for each IDL3 statement of the form

```
provides <facet_interface> <facet_name>;
```

the CIDL compiler generates a operation of the form

```
::CCM_<facet_interface>_ptr get_<facet_name>();
```

Thus, the IDL statement

```
provides Publication content;
```

causes the CIDL compiler to generate an operation in the Messenger executor with the signature

```
::CCM_Publication_ptr get_content();
```

The Messenger's `MessengerHome` manages the Messenger's life cycle. The component container creates an instance of the Messenger executor through its `MessengerHome`.

Recall that the Messenger's CIDL composition is as follows:

```
composition session Messenger_Impl
{
    home executor MessengerHome_Exec
    {
        implements MessengerHome;
        manages    Messenger_Exec;
    };
};
```

The CIDL compiler uses both the Messenger's IDL interface and its CIDL composition to generate an implementation of its executor. It generates a default Messenger executor with empty implementations of the `get_control()`, `get_content()`, and `get_message_history()` facet

accessors. The class diagram illustrates the `Messenger` executor class hierarchy.

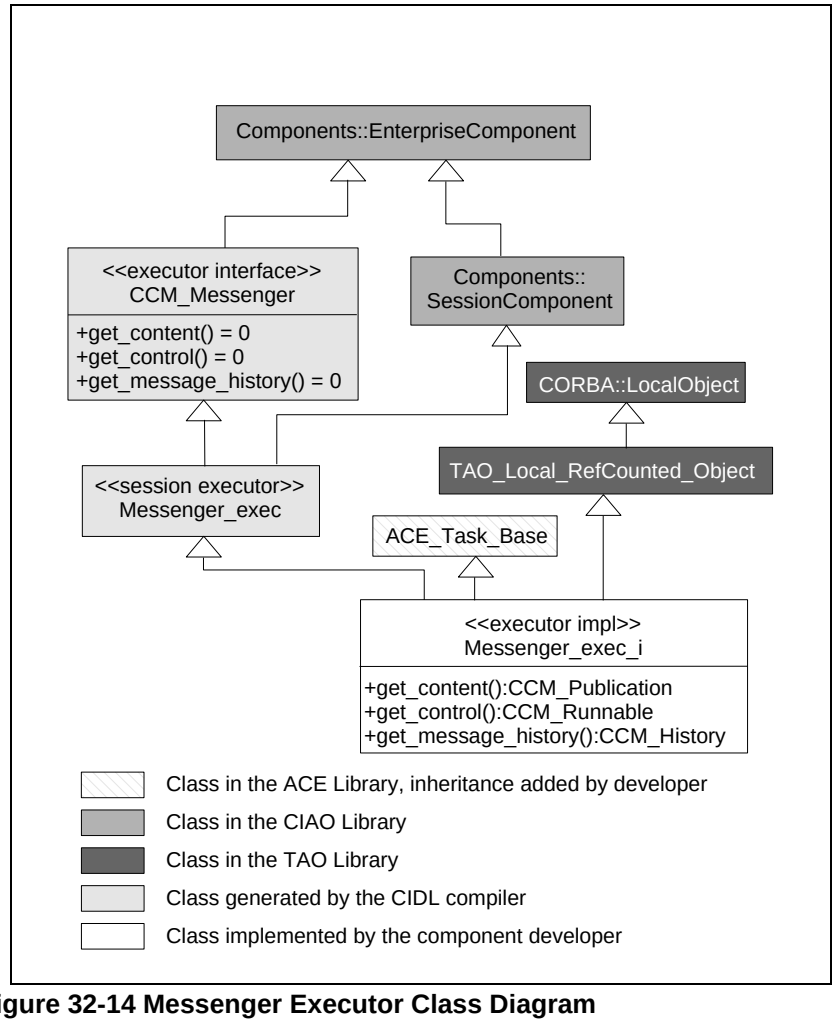

Figure 32-14 Messenger Executor Class Diagram

The IDL compiler does *not* generate a `Messenger` stub. The `Messenger` component is not an IDL2 interface. The CIDL compiler generates an abstract executor base class, `CCM_Messenger`, just as it did for the `Runnable` facet. The CIDL compiler also generates a `Messenger_exec` class that identifies the `Messenger` as a *session* component. A session component exports

transient object references and is responsible for managing its own persistent state if it has any.

The CIDL compiler optionally generates an empty executor implementation, `Messenger_exec_i`. The CIDL compiler generates a default constructor, a destructor, and a virtual method for each of the `Messenger`'s facets.

The executor implementation class also inherits from `::CORBA::LocalObject`, marking the `Messenger_exec_i` as a `CORBA::LocalObject` and allowing the component container to manage the executor's memory through reference counting.

The CIDL-generated executor implementation is as follows; as always, our changes are in **bold**. Comments are interspersed through the class definition.

```
#include "Messenger_svnt.h"
#include "Messenger_exec_export.h"
#include "tao/LocalObject.h"

#include <string>
#include <ace/Task.h>

namespace Messenger_Impl
{
  // forward declarations for executor implementations referenced
  // in the Messenger_exec_i class definition
  class Runnable_exec_i;
  class Publication_exec_i;
  class History_exec_i;
```

The Messenger executor is an active object, publishing messages in its own thread. It inherits from `ACE_Task_Base` to realize the active object behavior. There will be more on the implications of this later.

```
  class MESSENGER_EXEC_Export Messenger_exec_i
    : public virtual Messenger_Exec,
      public virtual ::CORBA::LocalObject,
      public virtual ACE_Task_Base
  {
  public:
```

The CIDL compiler generates a default constructor and destructor. There is no reason to change the signatures of these methods.

```
    Messenger_exec_i (void);
    virtual ~Messenger_exec_i (void);
```

The CIDL compiler generates an empty accessor and modifier for the `subject` attribute.

```
virtual char* subject ();

virtual void subject (const char* subject);
```

The CIDL compiler generates a `get_` operation for each of the Messenger's three provided facets.

```
virtual ::CCM_Runnable_ptr get_control ();

virtual ::CCM_Publication_ptr get_content ();

virtual ::CCM_History_ptr get_message_history ();
```

The Messenger has three facets: a `Runnable` facet called `control`, a `Publication` facet called `content`, and a `History` facet called `message_history`.

```
// Operations from Components::SessionComponent
```

The CIDL compiler generates a callback operation to set the component's *session context*. It generates a session context class that is specific to the component type. The component container instantiates and sets the component instance's session context at application startup.

The session context contains methods that enable the component to interact with the other components to which it is connected. Contexts are the glue that plug collaborating components together. As we'll see later, the Messenger component publishes `Message` events to interested consumers through its context.

```
virtual void set_session_context (::Components::SessionContext_ptr ctx);
```

The component container calls `set_session_context()` after it instantiates the component executor instance. The CIDL compiler also generates a private class member called `context_` to store the context and generates the implementation of the `set_session_context()` operation. No work is required on the part of the component developer.

The CIDL compiler generates three callback operations through which the component container indicates when the component is being activated, passivated, or removed.

```
virtual void ccm_activate ();

virtual void ccm_passivate ();

virtual void ccm_remove ();
```

The component container calls `ccm_activate()` to notify component that it has been activated. The `ccm_activate()` call completes before any other component operations are invoked. The component executor may perform its initialization in `ccm_activate()`. The component developer can assume that the session context has been initialized when the component container calls `ccm_activate()`. The Messenger's implementation of `ccm_activate()` calls `ACE_Task_Base::activate()` to launch a message-publishing thread.

The component container calls `ccm_passivate()` to notify the component that it has been deactivated. Here, the component executor should release any resources acquired in `ccm_activate()`. The component container then calls `ccm_remove()` when the component executor is about to be destroyed. The component developer can assume that the session context is still available when the component container calls `ccm_passivate()` or `ccm_remove()`.

The `ccm_activate()`, `ccm_passivate()`, and `ccm_remove()` operations are required by the OMG CORBA Component Model specification.

The `svc()` method is an implementation detail that is specific to our implementation of the `Messenger` executor.

```
virtual int svc();
```

It is overridden from the inherited `ACE_Task_Base` class. Our implementation of `ccm_activate()` calls `ACE_Task_Base::activate()` to launch a thread that executes the `svc()` method. The implementation of the `svc()` method publishes `Message` events to interested consumers.

The CIDL compiler automatically generates a `context_` class member.

```
private:
  ::CCM_Messenger_Context_var context_;
```

The component container calls `set_session_context()` to set the context when it initializes the component executor. The Messenger publishes its `Message` events through the context.

The component developer may add additional class members required to implement the component executor. We add several.

```
  private:
    Runnable_exec_i*    control_;
    Publication_exec_i* content_;
    History_exec_i*     history_;

    std::string       subject_;
    const std::string user_;
 };
}
```

The private `control_`, `content_`, and `history_` class members will be initialized by the user's code to contain pointers to the facet executors for the `Runnable`, `Publication`, and `History` facets of the Messenger component. The `user_` class member is a string that contains a user name that the Messenger embeds into each `Message` it publishes.

The Messenger executor implementation follows. As always, changes to the CIDL-generated default executor implementation code are in **bold**. Comments are interspersed with the code.

```
#include "Messenger_exec_i.h"
#include "ciao/CIAO_common.h"
```

The Messenger executor includes the executor class definitions for its `History`, `Runnable`, and `Publication` facets.

```
#include <ace/OS.h>
#include "History_exec_i.h"
#include "Runnable_exec_i.h"
#include "Publication_exec_i.h"

namespace Messenger_Impl
{
  //==================================================================
  // Component Executor Implementation Class:   Messenger_exec_i
  //==================================================================
```

The constructor creates executors for each of the `Messenger`'s three facets and initializes a "username."

```
Messenger_exec_i::Messenger_exec_i ()
  : subject_("Test Subject"),
    user_( "ciao_user" )
{
  // initialize user-defined data members
  this->control_ = new Runnable_exec_i();
  this->history_ = new History_exec_i();
  this->content_ = new Publication_exec_i(
    "Test Subject",
    "The quick brown fox jumped over the lazy dog",
    2 );
}
```

The destructor releases the memory for the Messenger's three facet executors. Because an executor is reference counted, we call `_remove_ref()` to release the memory of the executor rather than use the C++ `delete` operation.

```
Messenger_exec_i::~Messenger_exec_i (void)
{
  this->control_->_remove_ref();
  this->history_->_remove_ref();
  this->content_->_remove_ref();
}
```

The bulk of the Messenger's logic is in the `svc()` method.

```
int Messenger_exec_i::svc() {

  ACE_DEBUG((LM_INFO, ACE_TEXT("svc()\n")));

  while (1)
  {
    ACE_OS::sleep( this->content_->period() );

    // get the run_lock from the Runnable executor; we have an
    // agreement with the Runnable executor that we must wait for
    // the run_lock to be released before we publish.
    ACE_Guard<ACE_Thread_Mutex> guard( this->control_->get_run_lock() );

    // create a message to publish
    ::Message_var msg = new ::OBV_Message();
    msg->subject( this->subject() );
    msg->text( this->content_->text() );
    msg->user( CORBA::string_dup( this->user_.c_str() ) );
```

```
    // add the message to the message history
    this->history_->add( msg.in() );

    ACE_DEBUG((LM_INFO,
               ACE_TEXT("Messenger_exec_i::svc: publishing message\n") ));

    // publish to all interested consumers
    this->context_->push_message_publisher( msg.in() );

    ACE_DEBUG((LM_INFO,
      ACE_TEXT("Published Message on subject %C\n   User %C\n   Text %C\n"),
      msg->subject(),
      msg->user(),
      msg->text() ));
  }

  // not reached
  return 0;
}
```

We override the `svc()` method from the inherited `ACE_Task_Base` class. Our implementation of `ccm_activate()` calls the `ACE_Task_Base::activate()` method which launches the `svc()` method in a new thread. This method performs the bulk of the Messenger's work, looping continuously and publishing messages.

First, the `svc()` method sleeps for the period of time defined by the `period` attribute of the Messenger's `Publication` executor. Next, the `svc()` method attempts to acquire a lock from its `Runnable` executor. The Messenger executor and the `Runnable` executor have an agreement that the Messenger will not publish messages unless it can acquire the `Runnable` executor's `ACE_Thread_Mutex` lock. This permits the `Runnable` executor to start and stop message publication.

Once the Messenger acquires the `Runnable`'s lock, it publishes a message through its `context_`. The context acts like a proxy representing all interested consumers.

The CIDL compiler generates an empty accessor and modifier for the `subject` attribute. We add an implementation to each.

```
char*
Messenger_exec_i::subject ()
{
  return CORBA::string_dup( this->subject_.c_str() );
}
```

```
void
Messenger_exec_i::subject (const char* subject)
{
  this->subject_ = CORBA::string_dup( subject );
}
```

The CIDL compiler generates an empty implementation of each of the `get_content()`, `get_control()`, and `get_message_history()` facet accessor operations. We modify each to return the appropriate facet executor, incrementing its reference count before returning it.

```
::CCM_Publication_ptr
Messenger_exec_i::get_content ()
{
  // Your code here.

  // bump up ref count because we give up ownership when we return this
  this->content_->_add_ref();
  return this->content_;
}
```

The `Publication` facet controls the Messenger's message text and publication period. It is important to increment the reference count before returning the facet executor because we give up ownership of the facet executor when we return it. This behavior is consistent with the CORBA's memory management rules. Notice that we do not need to convert the executor to an object reference; we merely return it an allow the component container to do the heavy lifting.

The implementation of the `get_control()` facet accessor is nearly identical...

```
::CCM_Runnable_ptr
Messenger_exec_i::get_control ()
{
  // Your code here.

  // bump up ref count because we give up ownership when we return this
  this->control_->_add_ref();
  return this->control_;
}
```

...as is the implementation of `get_message_history()`.

```
::CCM_History_ptr
Messenger_exec_i::get_message_history ()
{
  // Your code here.

  // bump up ref count because we give up ownership when we return this
  this->history_->_add_ref();
  return this->history_;
}
```

The CIDL compiler generates a complete implementation of the `set_session_context()` operation.

```
// Operations from Components::SessionComponent

void
Messenger_exec_i::set_session_context (::Components::SessionContext_ptr ctx)
{
  this->context_ = ::CCM_Messenger_Context::_narrow (ctx);

  if (CORBA::is_nil (this->context_.in ()))
  {
    throw CORBA::INTERNAL ());
  }
}
```

The component container calls this operation immediately after it instantiates the Messenger executor. The component container calls `ccm_activate()` to notify the component that it has been activated.

```
void
Messenger_exec_i::ccm_activate ()
{
  // Your code here.
  ACE_DEBUG((LM_INFO, ACE_TEXT("Messenger_exec_i::ccm_activate\n")));
  this->activate();
}
```

The container does not send any requests to the component until `ccm_activate()` completes. This is typically where the component executor initializes itself. The Messenger executor calls the `ACE_Task_Base::activate()` method to spawn a thread running the `svc()` method.

The component container calls `ccm_passivate()` to notify the component that it has been deactivated and the component container will call

`ccm_remove()` when the component executor is about to be destroyed. Once `ccm_passivate()` is called, the component instance cannot be reactivated.

```
void
Messenger_exec_i::ccm_passivate ()
{
  // Your code here.
}

void
Messenger_exec_i::ccm_remove ()
{
  // Your code here.
}
```

Typically, a component executor cleans up after itself in `ccm_passivate()`, where it is guaranteed that the container hasn't started destroying its other component executors yet. Our executor has nothing to clean up.

32.2.4.5 The MessengerHome Executor

The CIDL compiler generates an implementation of the Messenger's home. Recall that the `MessengerHome`'s IDL3 interface is as follows:

```
component Messenger { ... };

home MessengerHome manages Messenger {};
```

and the Messenger's CIDL composition is as follows:

```
composition session Messenger_Impl
{
    home executor MessengerHome_Exec
    {
        implements MessengerHome;
        manages    Messenger_Exec;
    };
};
```

The CIDL compiler generates a complete implementation of the `MessengerHome` executor and a library entry point function. The component container instantiates the `MessengerHome` through the entry point function when it dynamically loads the Messenger's library. It generates the

`MessengerHome` executor in the same file as the Messenger component executor.

The `MessengerHome` is responsible for creating and destroying instances of the Messenger executor. The component container instantiates the Messenger executor through its home when it activates the Messenger. In our example the component developer does not need to modify any of the generated `MessengerHome` code nor the library entry point function.

The class diagram illustrates the `MessengerHome` executor class hierarchy.

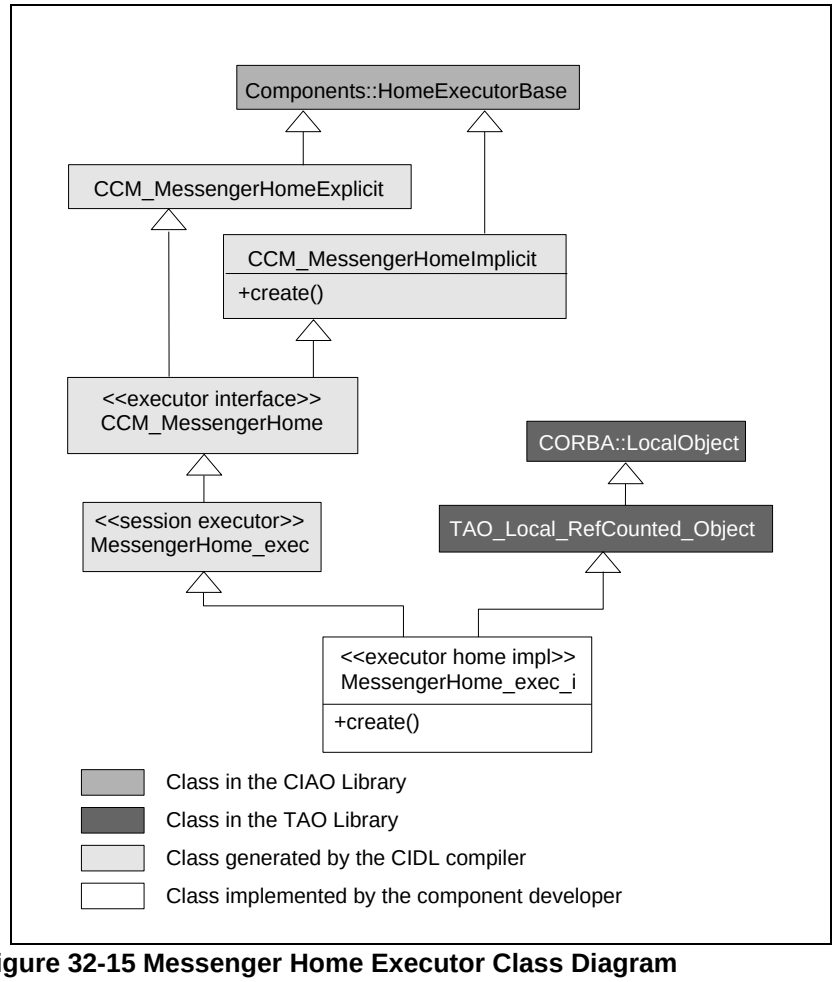

Figure 32-15 Messenger Home Executor Class Diagram

Both the class definition and the implementation of the `MessengerHome` and its entry point function are as follows. We make no changes to the generated code; thus, nothing is shown in bold. Comments are interspersed.

```
namespace CIDL_Messenger_Impl
{
```

Like the Messenger executor, the `MessengerHome` executor inherits from a generated executor base class.

```
  class MESSENGER_EXEC_Export MessengerHome_exec_i
  : public virtual MessengerHome_Exec,
    public virtual ::CORBA::LocalObject

  {
  public:
```

The CIDL compiler generates a default constructor and destructor.

```
    MessengerHome_exec_i (void);
    virtual ~MessengerHome_exec_i (void);
```

The CIDL compiler generates a default `create()` operation. The component container calls this `create()` operation to instantiate the Messenger component executor.

```
    virtual ::Components::EnterpriseComponent_ptr create ();

  };
```

Finally, the CIDL compiler generates a library entry point function. The component container's underlying ACE Service Configurator calls this entry point function to instantiate the `MessengerHome` executor when it dynamically loads the component's library. The entry point function must have "C" linkage to prevent C++ name mangling. The name of the function is CIAO-specific, but every CCM implementation generates an entry point function with this signature.

```
  extern "C" MESSENGER_EXEC_Export ::Components::HomeExecutorBase_ptr
    create_MessengerHome_Impl (void);
}
```

As you can see, we also make no changes to the generated `MessengerHome` implementation, which follows:

```
//=================================================================
// Home Executor Implementation Class:   MessengerHome_exec_i
//=================================================================
```

The CIDL compiler generates a default constructor and an empty destructor. The component developer may modify these. However, if the developer modifies the signature of the constructor, the developer must modify the implementation of the component home's library entry point function to pass the appropriate constructor arguments. The library entry point function will be shown in a few paragraphs.

```
MessengerHome_exec_i::MessengerHome_exec_i (void)
{
}

MessengerHome_exec_i::~MessengerHome_exec_i (void)
{
}
```

The Messenger's home has one operation, `create()`. The CIDL-generated implementation of `create()` simply invokes the Messenger executor's default constructor.

```
::Components::EnterpriseComponent_ptr
MessengerHome_exec_i::create ()
{
  ::Components::EnterpriseComponent_ptr retval =
    ::Components::EnterpriseComponent::_nil ();

  ACE_NEW_THROW_EX (retval,
                    Messenger_exec_i,
                    CORBA::NO_MEMORY ());

  return retval;
}
```

Finally, the CIDL compiler generates a library entry point function for the Messenger. This function simply creates an instance of the Messenger's home executor. The component container calls this function to create a `MessengerHome` when it loads the Messenger's dynamic library.

```
extern "C" MESSENGER_EXEC_Export ::Components::HomeExecutorBase_ptr
create_MessengerHome_Impl (void)
{
```

```
    ::Components::HomeExecutorBase_ptr retval =
      ::Components::HomeExecutorBase::_nil ();

    ACE_NEW_RETURN (retval,
                    MessengerHome_exec_i,
                    ::Components::HomeExecutorBase::_nil ());

    return retval;
  }
}
```

32.2.4.6 The Receiver Component Executor

The Receiver component connects to the Messenger component in two ways. First, its `message_consumer` port connects to the Messenger's `message_publisher` port. Second, its `message_history` receptacle connects to the Messenger's `message_history` facet.

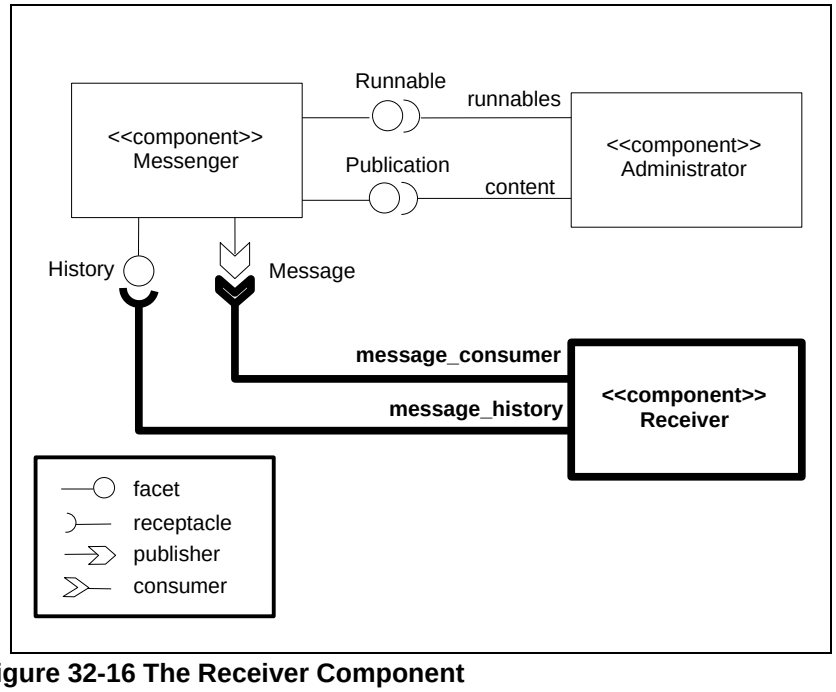

Figure 32-16 The Receiver Component

Recall that the Receiver's IDL specification is as follows:

```
component Receiver {
```

```
  consumes Message message_consumer;
  uses History message_history;
};

home ReceiverHome manages Receiver {};
```

The Receiver component subscribes to `Message` events and uses a `History` facet through which it retrieves a history of messages published. Both of these facets happen to be provided by the Messenger component, but the Receiver does not know that and does not need to know that. In fact, the `Message` events could be published by several different suppliers without the Receiver being aware of it.

The Receiver's `ReceiverHome` manages the Receiver's life cycle. The component container creates an instance of the Receiver executor through its `ReceiverHome`.

Recall that the Receiver's composition is as follows:

```
composition session Receiver_Impl
{
    home executor ReceiverHome_Exec
    {
        implements ReceiverHome;
        manages    Receiver_Exec;
    };
};
```

There are many similarities between the Receiver's executor and the Messenger's executor. As with the Messenger, the CIDL compiler uses both the Receiver's IDL interface and its CIDL composition to generate its executor. The CIDL compiler generates a class definition for the Receiver executor, another for the `ReceiverHome`, and a library entry point function. The CIDL compiler generates the Receiver's executor classes into the `Receiver_Impl` C++ namespace as specified by the Receiver's CIDL composition.

The most noticeable difference between the Messenger and the Receiver executors is the Receiver's message consumption callback operation. As a general rule, an IDL3 statement of the form

```
consumes <event_type> <facet_name>;
```

causes the CIDL compiler to generate a callback operation in the component's executor of the form:

```
virtual void push_<facet_name>( <event_type>* ev );
```

In our example, the IDL statement

```
consumes Message message_consumer;
```

causes the CIDL compiler to generate a callback operation with the signature

```
virtual void push_message_consumer( ::Message* ev);
```

The component container calls this operation when a connected `Message` supplier (in our case, the Messenger) publishes a `Message` event. The component container connects the suppliers and consumers dynamically at deployment time.

The CIDL-generated `Receiver` and `ReceiverHome` executor implementation class definitions are as follows. We make no changes to the generated Receiver executor class definition. Comments are interspersed.

```
#include "Receiver_svnt.h"
#include "Receiver_exec_export.h"
#include "tao/LocalObject.h"

namespace Receiver_Impl
{
```

The Receiver's executor implements the abstract executor base class `Receiver_exec`.

```
  class RECEIVER_EXEC_Export Receiver_exec_i
  : public virtual Receiver_Exec,
    public virtual ::CORBA::LocalObject
  {
```

The CIDL compiler generates a default constructor and a destructor.

```
  public:
    Receiver_exec_i (void);
    virtual ~Receiver_exec_i (void);
```

The component container calls `push_message_consumer()` on the Receiver when the Messenger publishes a `Message`.

```
virtual void push_message_consumer (::Message *ev);
```

Like the Messenger component, the Receiver component has a CIDL-generated `set_session_context()` callback operation. Again, the component container calls this operation after it instantiates the Receiver executor.

```
virtual void set_session_context (::Components::SessionContext_ptr ctx);
```

The CIDL compiler also generates standard `ccm_activate()`, `ccm_passivate()`, and `ccm_remove()` operations.

```
virtual void ccm_activate ();

virtual void ccm_passivate ();

virtual void ccm_remove ();
```

Finally, the CIDL compiler generates type-specific context member for the Receiver component.

```
private:
  ::CCM_Receiver_Context_var context_;
};
```

Our Receiver executor has no state nor private methods, so we make no changes to the CIDL-generated Receiver executor class definition.

The CIDL compiler also generates an implementation of the Receiver executor. The bulk of the work in implementing the Receiver executor is in the `push_message_consumer()` operation. The `push_message_consumer()` implementation uses the Receiver's `message_history` facet to get a list of all `Message` events published. The Receiver accesses the `message_history` facet through its session context.

As a general rule, an IDL3 statement of the form

```
uses <facet_interface> <facet_name>;
```

is mapped to a C++ function in the component's session context with the signature

```
::<facet_interface>_ptr get_connection_<facet_name>();
```

Thus, the Receiver's IDL3 statement

```
uses History message_history;
```

is mapped to a C++ function in the Receiver's context with the signature

```
::History_ptr get_connection_message_history();
```

Note *The Receiver specifies the services that it requires through the* `uses` *statement. This syntax enables the dynamic connection of service providers and service users at run time.*

The Receiver executor implementation follows. Changes to CIDL-generated executor implementation code are noted in **bold**:

```
#include "Receiver_exec_i.h"
#include "ciao/CIAO_common.h"

namespace Receiver_Impl
{
  //=================================================================
  // Component Executor Implementation Class:   Receiver_exec_i
  //=================================================================

  Receiver_exec_i::Receiver_exec_i (void)
  {
  }

  Receiver_exec_i::~Receiver_exec_i (void)
  {
  }
```

The component container invokes the `push_message_consumer()` operation a connected supplier publishes a `Message` event.

```
  void
  Receiver_exec_i::push_message_consumer (::Message * ev)
```

```
{
  // Your code here.

  CORBA::String_var subject = ev->subject();
  CORBA::String_var user = ev->user();
  CORBA::String_var text = ev->text();

  ACE_DEBUG(
    (LM_INFO,
    ACE_TEXT("Received Message:\n   Subject: %C\n   User: %C\n   Text: %C\n"),
     subject.in(),
     user.in(),
     text.in() ));

  // Use the history to (inefficiently) get the total number of messages
  // published on this item so far
  ::History_var history = this->context_->get_connection_message_history();
  ::Messages_var messages = history->get_all();
  ACE_DEBUG((LM_INFO,
             ACE_TEXT("   Subject \"%C\" has published %d messages so far\n"),
             subject.in(),
             messages->length() ));
}
```

The implementation of `push_message_consumer()` prints out the message's subject, user, and text. Then, it gets the component's `message_history` facet, gets a list of all `Message` events published, and prints out the number of messages published so far.

As with the Messenger component, the CIDL compiler generates a complete implementation of the `set_session_context()` method.

```
// Operations from Components::SessionComponent

void
Receiver_exec_i::set_session_context (::Components::SessionContext_ptr ctx)
{
  this->context_ = ::CCM_Receiver_Context::_narrow (ctx)

  if (CORBA::is_nil (this->context_.in ()))
  {
    throw CORBA::INTERNAL();
  }
}
```

The component container calls the `set_session_context()` operation to set the Receiver's context immediately after it instantiates the Receiver

executor. The Receiver uses this context to access its connected `message_history` facet. We do not make any changes to this method.

The CIDL compiler generates empty implementations of `ccm_activate()`, `ccm_passivate()`, and `ccm_remove()`. We do not modify any of these methods.

```
void
Receiver_exec_i::ccm_activate ()
{
  // Your code here.
}

void
Receiver_exec_i::ccm_passivate ()
{
  // Your code here.
}

void
Receiver_exec_i::ccm_remove ()
{
  // Your code here.
}
```

32.2.4.7 The ReceiverHome Executor

Recall that the `ReceiverHome`'s IDL3 interface is as follows:

```
component Receiver { ... };

home ReceiverHome manages Receiver {};
```

and the Receiver's CIDL composition is as follows:

```
composition session Receiver_Impl
{
    home executor ReceiverHome_Exec
    {
        implements ReceiverHome;
        manages    Receiver_Exec;
    };
};
```

The `ReceiverHome` class definition and implementation are nearly identical to the `MessengerHome` class definition and implementation. We make no changes to the CIDL-generated code.

```
  class RECEIVER_EXEC_Export ReceiverHome_exec_i
  : public virtual ReceiverHome_Exec,
    public virtual ::CORBA::LocalObject
  {
  public:
    ReceiverHome_exec_i (void);
    virtual ~ReceiverHome_exec_i (void);

    virtual ::Components::EnterpriseComponent_ptr create ();
  };

  extern "C" RECEIVER_EXEC_Export ::Components::HomeExecutorBase_ptr
    create_ReceiverHome_Impl (void);
}
```

Finally, the CIDL compiler generates implementations of the `ReceiverHome` executor and the Receiver's library entry point function. We do not modify these implementations.

```
  //=================================================================
  // Home Executor Implementation Class:   ReceiverHome_exec_i
  //=================================================================

  ReceiverHome_exec_i::ReceiverHome_exec_i (void)
  {
  }

  ReceiverHome_exec_i::~ReceiverHome_exec_i (void)
  {
  }

  ::Components::EnterpriseComponent_ptr
  ReceiverHome_exec_i::create ()
  {
    ::Components::EnterpriseComponent_ptr retval =
    ::Components::EnterpriseComponent::_nil ();

    ACE_NEW_THROW_EX (retval,
                      Receiver_exec_i,
                      CORBA::NO_MEMORY ());


    return retval;
  }
```

```
  extern "C" RECEIVER_EXEC_Export ::Components::HomeExecutorBase_ptr
  create_ReceiverHome_Impl (void)
  {
    ::Components::HomeExecutorBase_ptr retval =
    ::Components::HomeExecutorBase::_nil ();

    ACE_NEW_RETURN (retval,
                    ReceiverHome_exec_i,
                    ::Components::HomeExecutorBase::_nil ());

    return retval;
  }
}
```

32.2.4.8 The Administrator Component Executor

The Administrator component starts and stops the Messenger's message publication and controls the published text and the publication period. The Administrator component demonstrates use of the `uses multiple` mechanism of connecting a component receptacle to multiple facets simultaneously.

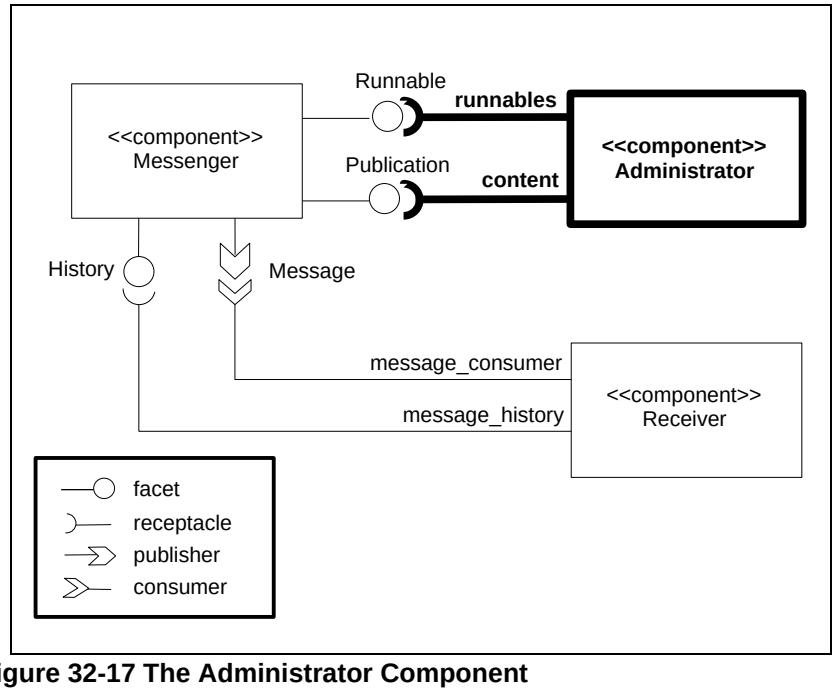

Figure 32-17 The Administrator Component

Recall that the Administrator's IDL3 interface is as follows:

```
component Administrator {
  uses multiple Runnable runnables;
  uses multiple Publication content;
};

home AdministratorHome manages Administrator {};
```

At deployment time, we connect the Administrator's `runnables` and `content` receptacles to the Messenger's `Runnable` and `Publication` facets. The Administrator controls the starting and stopping of message publication through the `Runnable` facet and controls the message text published and the publication period through the `Publication` facet.

The `uses multiple` modifier on the `runnables` and `content` receptacles permits the Administrator to connect an unlimited number of `Runnable` and `Publication` facets. This type of receptacle is a *multiplex receptacle*. Our example uses only one of each; however, an application deployer may add

additional Messengers into the application through XML-based configuration at deployment time. More information on deployment is available in 32.2.5.

Recall that the Administrator's composition, describing the implementation of the Administrator component, is as follows:

```
composition session Administrator_Impl
{
    home executor AdministratorHome_Exec
    {
        implements AdministratorHome;
        manages    Administrator_Exec;
    };
};
```

The CIDL compiler generates the Administrator's executor implementation code into an `Administrator_Impl` namespace. It generates an `Administrator_exec` executor base class for the Administrator and an `AdministratorHome_exec` executor base class for the Administrator's home. In addition, the CIDL compiler optionally generates default implementations of the executors. These are analogous to the CIDL-generated classes for the Messenger and Receiver components.

It is not surprising that there are many similarities between the Administrator executor and the Messenger and Receiver executors. The most noticeable change we make to the CIDL-generated executor implementation is to add inheritance from the `ACE_Task_Base` class and override the `ACE_Task_Base::svc()` method. The Administrator's implementation launches a thread that displays a simple text menu to start, stop, and otherwise control the Messenger's message publication. A more realistic application might use a GUI of some kind. The purpose of this example implementation is to demonstrate that a user can manually interact with a deployed component.

The Administrator's IDL3 interface introduces the `uses multiple` receptacle, or *multiplex* receptacle. A multiplex receptacle can connect to an unlimited number of type-compatible facets. The mapping from IDL to C++ is different for a multiplex receptacle than for a *simplex* receptacle, which connects to exactly one facet.

You might remember that for a receptacle of the form

```
uses <facet_interface> <facet_name>;
```

the CIDL compiler generates a C++ function in the component's context with the signature

```
::<facet_interface>_ptr get_connection_<facet_name>();
```

For example,

```
uses History message_history;
```

is mapped to a C++ function in the component's context with the signature

```
::History_ptr get_connection_message_history();
```

This type of receptacle is called a simplex receptacle because it connects to exactly one facet. However, a *multiplex* receptacle of the form

```
uses multiple <facet_interface> <facet_name>;
```

causes the IDL and CIDL compilers to generate the following:

- A C++ struct called `<facet_name>Connection`:

```
struct <facet_name>Connection
{
  typedef <facet_name>Connection_var _var_type;
  <facet_type>_var objref;
  Components::Cookie_var ck;
};
```

- A typedef for a sequence of `<facet_name>Connection` elements called `<facet_name>Connections`. We illustrate with the equivalent IDL2:

```
typedef sequence< <facet_name>Connection > <facet_name>Connections;
```

- A facet accessor C++ function through which the component retrieves a sequence of connected facets:

```
::<component name>::<facet_name>Connections*
  get_connections_<facet_name>();
```

In our example, the multiplex `Runnables` receptacle

```
uses multiple Runnable runnables;
```

causes the IDL and CIDL compilers to generate the following:

- A C++ struct called `runnablesConnection`:

```
struct runnablesConnection
{
  typedef runnablesConnection_var _var_type;
  Runnable_var objref;
  Components::Cookie_var ck;
};
```

- A C++ sequence type of `runnablesConnection` elements called `runnablesConnections`. Again, we illustrate with the equivalent IDL2:

```
typedef sequence< runnablesConnection > runnablesConnections;
```

- A facet accessor C++ function through which the component retrieves a sequence of connected `Runnable` facets:

```
::Administrator::runnablesConnections* get_connections_runnables();
```

The `startPublishing()`, `stopPublishing()`, `changePublicationText()`, and `changePublicationPeriod()` helper methods illustrate the usage of the multiplex receptacle.

The `Administrator` executor follows. Changes to CIDL-generated code are in **bold**.

```
#include "Administrator_svnt.h"
#include "Administrator_exec_export.h"
#include "tao/LocalObject.h"
#include <ace/Task.h>

namespace Administrator_Impl
{
```

The executor inherits from `ACE_Task_Base` to realize the active object pattern.

```
  class ADMINISTRATOR_EXEC_Export Administrator_exec_i
  : public virtual Administrator_Exec,
    public virtual ACE_Task_Base,
    public virtual ::CORBA::LocalObject
  {
  public:
```

The remainder of the executor class definition is the analogous to the Messenger's and Receiver's...

```
    Administrator_exec_i (void);
    virtual ~Administrator_exec_i (void);

    virtual void set_session_context (::Components::SessionContext_ptr ctx);

    virtual void ccm_activate ();

    virtual void ccm_passivate ();

    virtual void ccm_remove ();


  private:
    ::CCM_Administrator_Context_var context_;
```

...except that we override the `ACE_Task_Base::svc()` method and add several private helper methods to control the connected `Runnable` and `Publication` facets.

```
  public:
    // Overridden from ACE_Task_Base
    int svc();

  private:
    void startPublishing();
    void stopPublishing();
    void changePublicationPeriod();
    void changePublicationText();
  };
```

The Administrator's executor implementation follows, with changes to CIDL-generated code in **bold**:

```
#include "Administrator_exec_i.h"
#include "ciao/CIAO_common.h"

#include <iostream>
#include <string>

namespace Administrator_Impl
{
  //==================================================================
  // Component Executor Implementation Class:   Administrator_exec_i
```

```
//==================================================================

Administrator_exec_i::Administrator_exec_i (void)
{
}

Administrator_exec_i::~Administrator_exec_i (void)
{
}
```

As usual, we have no need to modify the generated `set_session_context()` callback operation.

```
void
Administrator_exec_i::set_session_context (
  ::Components::SessionContext_ptr ctx)
{
  this->context_ = ::CCM_Administrator_Context::_narrow (ctx);

  if (CORBA::is_nil (this->context_.in ()))
  {
    throw CORBA::INTERNAL ();
  }
}
```

In the `ccm_activate()` implementation, the executor calls `ACE_Task_Base::activate()` to launch its `svc()` method in a thread.

```
void
Administrator_exec_i::ccm_activate ()
{
  // Your code here.

  // Activate the Task, triggering its svc() method
  this->activate();
}
```

We have no need to modify any of the remaining generated operations.

```
void
Administrator_exec_i::ccm_passivate ()
{
  // Your code here.
}

void
Administrator_exec_i::ccm_remove ()
```

```
{
  // Your code here.
}
```

The Administrator's implementation of the `ccm_activate()` launches a thread and calls `svc()`. The `svc()` method creates a small text menu through which a user can start and stop message publication, change the message text, and change the publication period for every Messenger attached to the Administrator. Helper methods implement those behaviors. Nothing in the implementation of the `svc()` method involves the CORBA Component Model. Code relevant to CCM is in the helper methods, described later.

```
int
Administrator_exec_i::svc()
{
  enum SelectionType { START=1, STOP, CHANGE_PERIOD, CHANGE_TEXT };

  while ( 1 ) {
    std::cout << "\nWhat do you want to do to the Messenger(s)?" << std::endl;
    std::cout << START << ".  Start" << std::endl;
    std::cout << STOP << ".  Stop" << std::endl;
    std::cout << CHANGE_PERIOD << ".  Change Publication Period" << std::endl;
    std::cout << CHANGE_TEXT << ".  Change Publication Text" << std::endl;

    char selection_text[10];
    std::cout << "Please enter a selection: ";
    std::cin.getline( selection_text, sizeof(selection_text) );
    int selection = ACE_OS::atoi(selection_text);

    switch (selection) {
      case START:
        startPublishing();
        break;
      case STOP:
        stopPublishing();
        break;
      case CHANGE_PERIOD:
        changePublicationPeriod();
        break;
      case CHANGE_TEXT:
        changePublicationText();
        break;
      default:
        std::cout << "Please enter a valid option" << std::endl;
    }
  }
  return 0;
}
```

The following four developer-written methods are helper methods invoked by `svc()` in response to user interaction. The first method, `startPublishing()`, retrieves the `Runnable` facets connected to the Administrator's `runnables` receptacle and invokes `start()` on each one.

```
void Administrator_exec_i::startPublishing()
{
  // Get the attached Runnable facet(s)
  ::Administrator::runnablesConnections_var connections =
    this->context_->get_connections_runnables();

  std::cout << "Starting Publication" << std::endl;
  for ( CORBA::ULong i = 0; i < connections->length(); ++i ) {
    Runnable_var runnable = (*connections)[i].objref;
    runnable->start();
  }
}
```

In `startPublishing()`, we use the Administrator context's `get_connections_runnables()` method to get a list of the `Runnable` facets connected to the Administrator's `runnables` receptacle. That method returns a sequence of `runnablesConnection` structs. One of the members of the `runnablesConnection` struct is a `Runnable` object reference called `objref`. We pull the object reference out of the struct and call `start()` on it to start message publication.

The `stopPublishing()` method is almost identical to the `startPublishing()` method...

```
void Administrator_exec_i::stopPublishing()
{
  // Get the attached Runnable facet(s)
  ::Administrator::runnablesConnections_var connections =
    this->context_->get_connections_runnables();

  std::cout << "Stopping Publication" << std::endl;
  for ( CORBA::ULong i = 0; i < connections->length(); ++i ) {
    Runnable_var runnable = (*connections)[i].objref;
    runnable->stop();
  }
}
```

..., except it calls `stop()` on each connected `Runnable` facet instead of `start()`.

The `changePublicationPeriod()` and `changePublicationText()` methods are similar to `startPublishing()` and `stopPublishing()`. The main difference is that they operate on the `Publication` receptacle rather than the `Runnables` receptacle.

```
void Administrator_exec_i::changePublicationPeriod()
{
  // Get the attached Publication facet(s)
  ::Administrator::contentConnections_var contents =
    this->context_->get_connections_content();

  char period[10];
  std::cout << "Please enter a new period in seconds: ";
  std::cin.getline( period, sizeof( period ) );
  for ( CORBA::ULong i = 0; i < contents->length(); ++i ) {
    Publication_var publication = (*contents)[i].objref;
    publication->period( ACE_OS::atoi(period) );
  }
}
```

In `changePublicationPeriod()`, we use the Administrator context's `get_connections_content()` operation to get a list of the `Publication` facets connected to the Administrator's `content` receptacle. That method returns a sequence of `contentConnection` structs. One of the members of the `contentConnection` struct is a `Publication` object reference called `objref`. We pull the object reference out of the struct and call `period()` on it to change the message publication period.

The `changePublicationText()` method is nearly identical to `changePublicationPeriod()`...

```
void Administrator_exec_i::changePublicationText()
{
  // Get the attached Publication facet(s)
  ::Administrator::contentConnections_var contents =
    this->context_->get_connections_content();

  char buffer[1024];
  std::cout << "Please enter new text: ";
  std::cin.getline( buffer, sizeof(buffer) );
  for ( CORBA::ULong i = 0; i < contents->length(); ++i ) {
    Publication_var publication = (*contents)[i].objref;
    publication->text( buffer );
  }
}
```

... except it calls the `text()` method on each `Publication` object reference to change the published text.

32.2.4.9 The AdministratorHome Executor

Recall that the `AdministratorHome`'s IDL3 interface is as follows:

```
component Administrator { ... };

home AdministratorHome manages Administrator {};
```

and the `Administrator`'s CIDL composition is as follows:

```
composition session Administrator_Impl
{
    home executor AdministratorHome_Exec
    {
        implements AdministratorHome;
        manages    Administrator_Exec;
    };
};
```

The `AdministratorHome` class definition and implementation are nearly identical to the `MessengerHome` and `ReceiverHome` class definition and implementation. As with those, the CIDL compiler can optionally generate the full class definition, implementation, and library entry point function.

```
 class ADMINISTRATOR_EXEC_Export AdministratorHome_exec_i
  : public virtual AdministratorHome_Exec,
    public virtual ::CORBA::LocalObject
  {
  public:
    AdministratorHome_exec_i (void);
    virtual ~AdministratorHome_exec_i (void);

   virtual ::Components::EnterpriseComponent_ptr create ();
  };

  extern "C" ADMINISTRATOR_EXEC_Export ::Components::HomeExecutorBase_ptr
    create_AdministratorHome_Impl (void);
}
```

The implementation of the class and the library entry point function follows.

```
  //=================================================================
  // Home Executor Implementation Class:   AdministratorHome_exec_i
```

```
  //=================================================================

  AdministratorHome_exec_i::AdministratorHome_exec_i (void)
  {
  }

  AdministratorHome_exec_i::~AdministratorHome_exec_i (void)
  {
  }

  ::Components::EnterpriseComponent_ptr
  AdministratorHome_exec_i::create ()
  {
    ::Components::EnterpriseComponent_ptr retval =
    ::Components::EnterpriseComponent::_nil ();

    ACE_NEW_THROW_EX (retval,
                      Administrator_exec_i,
                      CORBA::NO_MEMORY ());


    return retval;
  }

  extern "C" ADMINISTRATOR_EXEC_Export ::Components::HomeExecutorBase_ptr
  create_AdministratorHome_Impl (void)
  {
    ::Components::HomeExecutorBase_ptr retval =
    ::Components::HomeExecutorBase::_nil ();

    ACE_NEW_RETURN (retval,
                    AdministratorHome_exec_i,
                    ::Components::HomeExecutorBase::_nil ());

    return retval;
  }
}
```

32.2.4.10 Summary of the Code

The code for our Messenger example is complete. We have implemented executors for the Messenger, Receiver, and Administrator components. We also implemented executors for the `Runnable`, `Publication`, and `History` facets of the Messenger component.

We have seen how the component container injects a *context* into each component executor to facilitate connections between facets and receptacles and between publishers and consumers.

Consider what we have not done. We have not written a `main()`. We have not interacted with the Portable Object Adapter, nor have we been exposed to any classes with the prefix `POA_`. We have not written any code that attempts to find another component or object; instead, those connections are provided through each component's context.

In the following sections we deploy the Messenger example through CIAO's implementation of the CCM Deployment and Configuration specification.

32.2.5 Deploying the Messenger Application

As the previous sections have demonstrated, a CCM-based application consists of small, self-contained, reusable software components defined using IDL3. A component defines its interactions with other components via ports indicating provided and required interfaces and messages published and consumed. The deployment activity separates business application logic from process and interaction details.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component's composition
  - Implement a C++ executor for each component and facet
- **Describe the application's deployment**
  - Describe each component's libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application
- Run the application

Figure 32-18 Road Map

An *assembly* is a set of interconnected component instances. Each assembly is itself a component; an assembly may be deployable as a full application, or may represent a higher-level component for use in a larger application. An assembly is a realizations of the *composite* pattern; an assembly may be composed of other assemblies and may be a part of other assemblies.

A component deployer deploys an assembly into one or more application servers that might be distributed across a network. XML descriptor files

describe how the components are plugged together into an assembly and how each component of the assembly is mapped to an application server process. Thus, component assembly and deployment is completely independent of component implementation, achieving a separation of concerns.

Each component may be platform-specific, but an assembly may be heterogeneous. A component implementation neither knows nor cares which assemblies it may be a part of nor knows to which other components it may connect.

Our deployment of the Messenger application is described by the following UML deployment diagram.

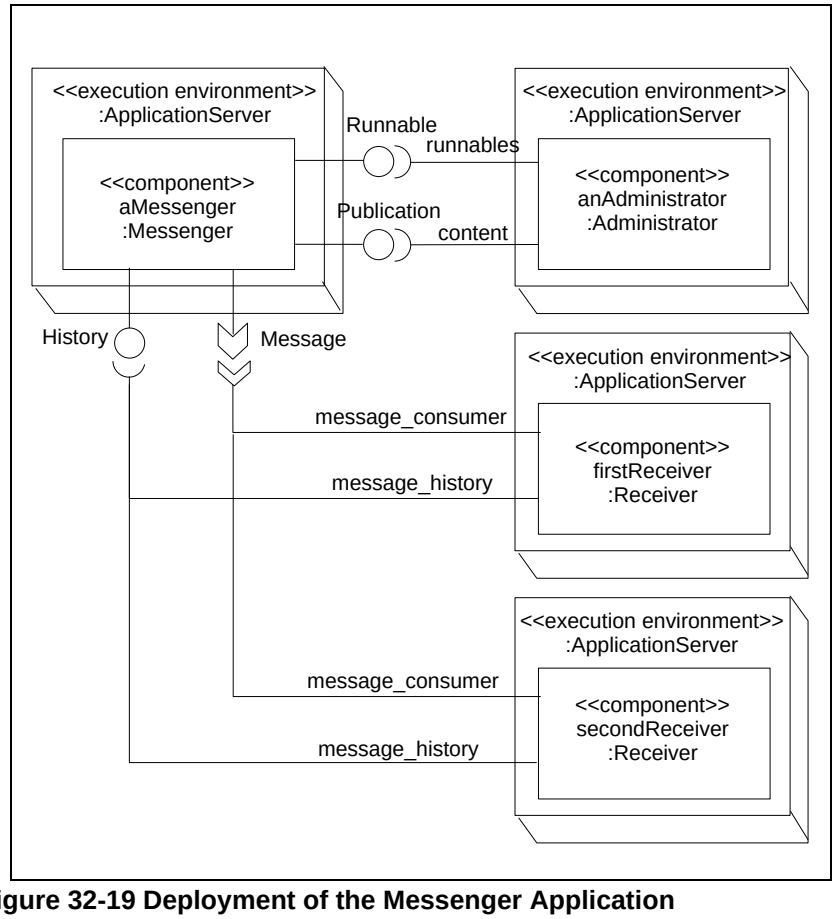

Figure 32-19 Deployment of the Messenger Application

The deployment consists of one Messenger instance, two Receiver instances, and one Administrator instance. We deploy each instance in its own component server, although it is not necessary to do that. In our deployment, the four component servers execute on the same host, although we could distribute them across the network with no code changes and minimal configuration changes.

The "Deployment and Configuration of Component-based Distributed Applications" specification (OMG Document ptc/03-07-08) prescribes how components are instantiated, connected, and assigned to processes in a distributed software system. The following sections illustrate the salient points of the D&C specification through the deployment of the Messenger application.

The example's XML descriptor files ares in the `$CIAO_ROOT/examples/DevGuideExamples/Messenger/descriptors` directory.

32.2.5.1 Deployment and Configuration Specification

The OMG's specification for the "Deployment and Configuration of Component-based Distributed Applications" is a deployment and configuration specification that is independent of the CORBA Component Model. It describes a general-purpose deployment and configuration framework for use by any component-based application. The specification defines IDL interfaces and XML descriptor file formats for configuring individual components and component assemblies for deployment.

This Deployment and Configuration specification supersedes the "Packaging and Deployment" chapter of the OMG CORBA Component Model (CCM) (OMG Document formal/02-06-65) specification.

CIAO's realization of the Deployment and Configuration specification is called DAnCE (Deployment And Configuration Engine). DAnCE supersedes CIAO's implementation of the CCM specification's "Packaging and Deployment" chapter. We deploy the Messenger application using DAnCE in this section.

32.2.5.2 Deployment Descriptors

The Deployment and Configuration specification presents a set of XML descriptors for describing deployment aspects of a software system. Each

component has a set of descriptors to define its libraries, exposed ports, and implementation. Each application consists of one or more assemblies that describe the application's packaging and deployment onto component servers.

A CCM application deployer writes many deployment descriptor files to describe the application's deployment. These files are written by hand. Usually, an application deployer copies and edits an existing set of XML deployment descriptors to describe a new application's deployment. In the future, we might expect tools such as Vanderbilt's CoSMIC or Kansas State's Cadena to generate the bulk of our CIAO application's deployment descriptors.

A component such as the Messenger component describes its deployment using the following descriptors:

- An *Implementation Artifact Descriptor* (.iad) file for each of the Messenger's libraries.
- A *CORBA Component Descriptor* (.ccd) file defining the Messenger's exposed ports and attributes.
- A *Component Implementation Descriptor* (.cid) file describing the component's implementation in terms of its Implementation Artifact Descriptors and its CORBA Component Descriptor.
- A *Component Package Descriptor* (.cpd) file describing one or more implementations of the component.

An assembly is a composite component, consisting of interconnected subcomponents. We assemble a Messenger component instance, two Receiver component instances, and an Administrator component instance into a deployable assembly. The assembly describes its deployment using the following descriptors:

- Another *CORBA Component Descriptor* (.ccd) file defining the assembly's exposed ports and attributes, if any.
- Another *Component Implementation Descriptor* (.cid) file describing the assembly's implementation in terms of its subcomponents and the connections between their ports.
- Another *Component Package Descriptor* (.cpd) file describing one or more implementations of the assembly.

An application consists of one or more assemblies. The application describes its deployment using the following descriptors:

- A *Package Configuration Descriptor* (.pcd) that describes a deployable component package.
- A *Top-level Package Descriptor* (package.tpd) that contains one or more Package Configuration Descriptors.
- A *Component Deployment Plan* (.cdp) that maps each component instance in the assembly's Component Implementation Description to a logical node.
- A *Component Domain Descriptor* (.cdd) that describes available nodes, interconnects, and bridges.
- A *Node Map* that maps each logical node to a physical component server process.

The deployment engineer may choose to combine several of these descriptor files into one. In fact, we could describe the Messenger application's deployment with one rather large XML descriptor. However, we will keep the descriptor files separate for maximum flexibility.

The following sections outline the deployment of the individual components, the Messenger assembly, and the full application.

32.2.5.3 Deployment.xsd and XMI.xsd Files

The deployment descriptors are described by two XML Schema Definition (XSD) files: `Deployment.xsd` and `XMI.xsd`. The directory containing the Messenger application's deployment descriptors must contain a copy of each of these files. Each file can be copied from DAnCE's root directory, `$CIAO_ROOT/DAnCE` or `%CIAO_ROOT%\DAnCE`. These directories are referred to as $DANCE_ROOT or %DANCE_ROOT% from this point forward.

32.2.5.4 Deploying the Messenger Component

The Messenger component's deployment descriptors package its libraries and its exposed ports into one deployable package.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component's composition
  - Implement a C++ executor for each component and facet
- Describe the application's deployment
  - **Describe each component's libraries and ports**
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application
- Run the application

Figure 32-20 Road Map

Six XML descriptors describe the Messenger component's deployment: three descriptors describe each of the Messenger's three libraries; one descriptor describes its exposed ports; one descriptor combines the library descriptors and the port descriptor into an implementation; and one descriptor packages the Messenger into a deployable package.

The drawing illustrates the relationships between the six Messenger deployment descriptors.

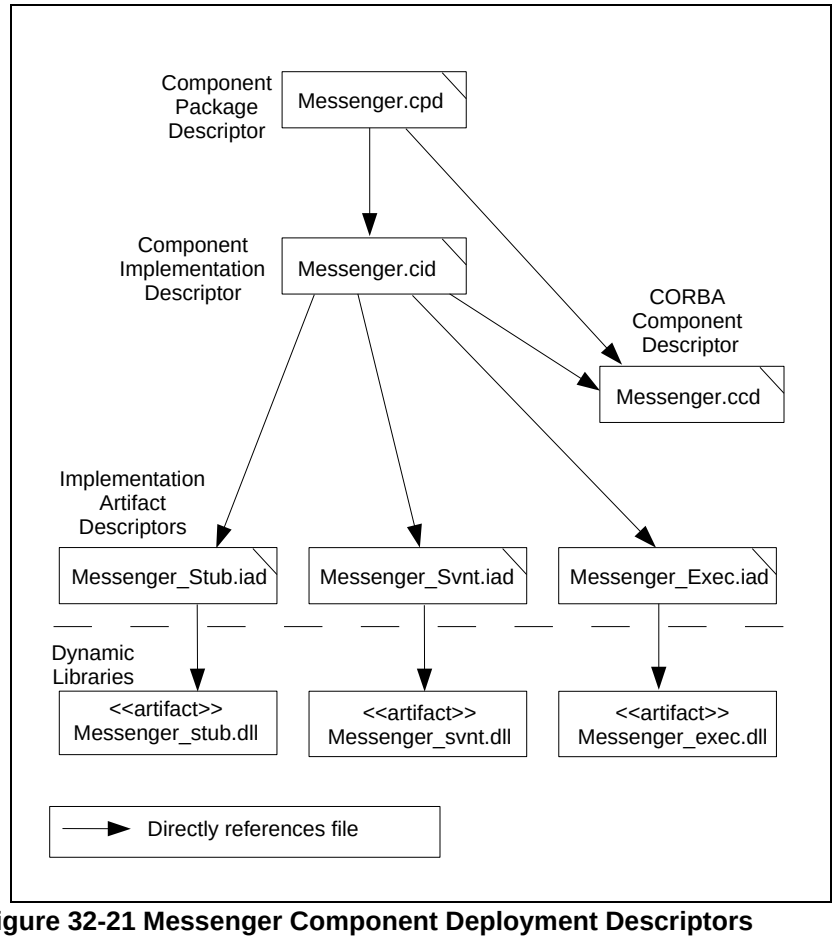

Figure 32-21 Messenger Component Deployment Descriptors

The Messenger's implementation is linked into three dynamic libraries: `Messenger_stub`, containing the Messenger's IDL-generated stub code; `Messenger_svnt`, containing the Messenger's IDL- and CIDL-generated skeleton and servant code; and `Messenger_exec`, containing the Messenger's home and component executors. We create an *Implementation Artifact Descriptor* for each of these libraries.

A *CORBA Component Descriptor* describes the Messenger's public interface. It contains information about each of the component's ports, including the

port name, the type of port (`Facet`, `EventPublisher`, `EventConsumer`, etc.) and the port's supported IDL interfaces.

The *Component Implementation Descriptor* describes the Messenger component's *monolithic* implementation. A monolithic component implementation consists of one or more implementation artifacts. In a C++ application, an implementation artifact is a dynamic library.

At the top level, a *Component Package Descriptor* may describe several alternate implementations of a component, permitting the component server to choose the correct implementation at run-time based on platform and QoS requirements. Our example provides one implementation of the Messenger component.

Messenger Component - Implementation Artifact Descriptors (.iad)

An *implementation artifact* is a library, a jar file, or some other artifact containing a component's executable code. We create an Implementation Artifact Descriptor for each of our three Messenger libraries. We also create a separate Implementation Artifact Descriptor for the ACE, TAO, and CIAO libraries.

The Implementation Artifact Descriptor for the ACE, TAO, and CIAO libraries is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ImplementationArtifactDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>ACE/TAO/CIAO Libraries</label>
  <location>$ACE_ROOT/lib/ACE</location>
  <location>$ACE_ROOT/lib/TAO</location>
  <location>$ACE_ROOT/lib/CIAO_Client</location>
</Deployment:ImplementationArtifactDescription>
```

The optional `<label>` element contains a human-readable description of the implementation artifact. It may be used by a tool for display purposes. The mandatory `<location>` elements reference the `ACE`, `TAO` and `CIAO_Client` libraries that the Messenger depends upon. File extensions for the libraries are not necessary, or even desired. Multiple alternate location for the same entity can be provided. The underlying implementation uses the operating system's

dynamic library capabilities, meaning that it can use the contents of the PATH and/or the LD_LIBRARY_PATH to find the dynamic libraries.

Note *Notice the simplicity of the specified library names. The ACE library is specified merely as* ACE*, not as* ACE.dll*,* ACEd.dll*,* libACE.so*, etc. The simplified name enables the component developer to describe an implementation artifact in a platform-independent manner. This behavior is specific to CIAO.*

An Implementation Artifact Descriptor describes each of the three Messenger libraries. The Messenger_stub library is described as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ImplementationArtifactDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Stub Artifact</label>
  <location>Messenger_stub</location>
  <dependsOn>
    <name>ACE/TAO/CIAO</name>
    <referencedArtifact href="Libraries.iad"/>
  </dependsOn>
</Deployment:ImplementationArtifactDescription>
```

The optional <label> element may be used by a tool for display purposes. The <location> element contains the simplified name of the library. Since we have not provided a path, the library must be in the application's PATH (Windows) or LD_LIBRARY_PATH (UNIX). Optionally, we could provide a path, as we did for the ACE, TAO, and CIAO libraries.

Each <dependsOn> element contains references to dependent implementation artifacts. We depend on the ACE/TAO/CIAO libraries, so our <dependsOn> entry references the Libraries.iad file containing references to the ACE/TAO/CIAO libraries. The mandatory <name> sub-element may be used by a tool for display purposes.

The descriptor also recognizes one or more optional <infoProperty> elements that provide non-functional information that might be displayed by a tool. For example:

```
  <infoProperty>
    <name>comment</name>
    <value>
      <type>
        <kind>tk_string</kind>
      </type>
      <value>
        <string>This IAD describes the Messenger's stub library</string>
      </value>
    </value>
  </infoProperty>
```

The Messenger has two more Implementation Artifact Descriptors, one for its `Messenger_svnt` library and one for its `Messenger_exec` library. The `Messenger_svnt.iad` files is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ImplementationArtifactDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Servant Artifact</label>
  <location>Messenger_svnt</location>
  <dependsOn>
    <name>ACE/TAO/CIAO</name>
    <referencedArtifact href="Libraries.iad"/>
  </dependsOn>
  <dependsOn>
    <name>Messenger_Stub</name>
    <referencedArtifact href="Messenger_Stub.iad"/>
  </dependsOn>
  <execParameter>
    <name>entryPoint</name>
    <value>
      <type>
        <kind>tk_string</kind>
      </type>
      <value>
        <string>createMessengerHome_Servant</string>
      </value>
    </value>
  </execParameter>
</Deployment:ImplementationArtifactDescription>
```

The `<label>`, `<location>`, and first `<dependsOn>` elements contain similar information to that in the `Messenger_stub.iad` file. However, the `Messenger_svnt` library also depends on the `Messenger_stub` library, as

reflected in the additional `<dependsOn>` element. In addition, the `Messenger_svnt` library has an *entry point function*, configured through the `<execParameter>` element. An entry point function always has the `<name>` of "entryPoint". The execution parameter's `<value>` element is actually an XML representation of a CORBA Any. The `<value>` element's string value matches the name of the Messenger's library entry point function as generated by the CIDL compiler. Additional information on the CIDL compiler is available in 32.5.

Note *CIAO looks for an "entryPoint" execution parameter for any implementation artifact that ends in _svnt or _exec. Thus, your servant and executor implementation artifacts should end in _svnt and _exec, respectively.*

The `Messenger_exec` Implementation Artifact Descriptor is nearly identical to the `Messenger_svnt` descriptor, as is shown below:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ImplementationArtifactDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Executor Artifact</label>
  <location>Messenger_exec</location>
  <dependsOn>
    <name>ACE/TAO/CIAO</name>
    <referencedArtifact href="Libraries.iad"/>
  </dependsOn>
  <dependsOn>
    <name>Messenger_Stub</name>
    <referencedArtifact href="Messenger_Stub.iad"/>
  </dependsOn>
  <execParameter>
    <name>entryPoint</name>
    <value>
      <type>
        <kind>tk_string</kind>
      </type>
      <value>
        <string>createMessengerHome_Impl</string>
      </value>
    </value>
  </execParameter>
</Deployment:ImplementationArtifactDescription>
```

The `Messenger_exec` implementation artifact also has a library entry point function, matching the name of the entry point function generated by the CIDL compiler.

In summary, each of the component's Implementation Artifact Descriptor files contains information about one of the component's libraries. There is an Implementation Artifact Descriptor for each of the `Messenger_stub`, `Messenger_svnt`, and `Messenger_exec` libraries.

Messenger Component - CORBA Component Descriptor (.ccd)

The *CORBA Component Descriptor* describes the component's IDL3 interface in an XML format. Primarily, it describes the component's exposed ports and attributes. The mapping from a component's IDL file to a CORBA Component Descriptor is purely mechanical.

There are six kinds of component ports: `Facet`, `SimplexReceptacle`, `MultiplexReceptacle`, `EventPublisher`, `EventEmitter`, and `EventConsumer`, as shown in Table 32-4:

Table 32-4 Component Port Types

Port Kind	Sample IDL3 Declaration
`Facet`	`provides Runnable control`
`SimplexReceptacle`	`uses Runnable control`
`MultiplexReceptacle`	`uses multiple Runnable controls`
`EventPublisher`	`publishes Message message_publisher`
`EventEmitter`	`emits Message message_emitter`
`EventConsumer`	`consumes Message message_consumer`

Recall that the Messenger component's IDL3 is as follows:

```
component Messenger {
  attribute string subject;

  provides Runnable control;
  provides Publication content;

  publishes Message message_publisher;
  provides History message_history;
};
```

We create the `Messenger.ccd` file describing the Messenger's IDL3 interface as follows. Comments are interspersed.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentInterfaceDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Component</label>
  <specificType>IDL:Messenger:1.0</specificType>
  <supportedType>IDL:Messenger:1.0</supportedType>
  <idlFile>Messenger.idl</idlFile>
```

The optional `<label>` element contains a description that may be used by a tool for display purposes. The `<specificType>` element contains the Interface Repository Id of the component's IDL interface.

The descriptor has a `<supportedType>` element for the component's Interface Repository Id. It also has a `<supportedType>` element for each IDL2 interface supported by the component either directly or through inheritance. A component may indicate that it *supports* an IDL2 interface through the `supports` keyword. For example.

```
component Messenger supports MyInterface {
```

would map to an additional `<supportedType>` element such as this:

```
<supportedType>IDL:MyInterface:1.0</supportedType>
```

A component supporting an interface inherits the operations, attributes, etc. from that interface. However, we don't use the `supports` keyword in our examples.

The optional `<idlFile>` element points to the IDL file that is the source of this information. It is for documentation purposes.

```
<property>
  <name>subject</name>
  <type>
    <kind>tk_string</kind>
  </type>
</property>
```

A `<property>` element describes each of the component's IDL attributes. This `<property>` element describes the component's `subject` attribute. The

<name> element's value matches the attribute name in the IDL file. The <type> element's <kind> is a type code. In our example, the attribute is a string. The "Deployment and Configuration" specification (OMG Document ptc/03-07-08) and the Deployment.xsd schema file contain more information on representing data types in XML descriptors.

```
<port>
  <name>control</name>
  <exclusiveProvider>false</exclusiveProvider>
  <exclusiveUser>false</exclusiveUser>
  <optional>true</optional>
  <provider>true</provider>
  <specificType>IDL:Runnable:1.0</specificType>
  <supportedType>IDL:Runnable:1.0</supportedType>
  <kind>Facet</kind>
</port>
```

Each provides, uses, uses multiple, publishes, emits, and consumes declaration has a matching <port> element. This port corresponds to the

```
provides Runnable control;
```

facet. The <name> element's value matches the facet name in the IDL file. The <specificType> element contains the Interface Repository Id of the facet's IDL interface. There may be several <supportedType> elements; there is one for the facet's most specific IDL interface and one for each inherited interface regardless of whether the inheritance is direct or indirect.

The <kind> element's value is Facet. Valid <kind> values are Facet, SimplexReceptacle, MultiplexReceptacle, EventPublisher, EventEmitter, and EventConsumer. The <optional> element indicates if connecting to the port is optional or mandatory. The <provider> element's value is true for provides and consumes, false for uses and publishes.

```
<port>
  <name>content</name>
  <exclusiveProvider>false</exclusiveProvider>
  <exclusiveUser>false</exclusiveUser>
  <optional>true</optional>
  <provider>true</provider>
  <supportedType>IDL:Publication:1.0</supportedType>
  <specificType>IDL:Publication:1.0</specificType>
  <kind>Facet</kind>
</port>
```

This is the `Publication` facet called `content`. Its declaration is nearly identical to that of the `Runnable` facet

```
<port>
  <name>message_publisher</name>
  <exclusiveProvider>false</exclusiveProvider>
  <exclusiveUser>false</exclusiveUser>
  <optional>true</optional>
  <provider>false</provider>
  <supportedType>IDL:Message:1.0</supportedType>
  <specificType>IDL:Message:1.0</specificType>
  <kind>EventPublisher</kind>
</port>
```

This is the `Message` publishing port called `message_publisher`. The `<supportedType>` and `<specificType>` are the Interface Repository Id of the event type being published. The port's `<kind>` is `EventPublisher`.

```
<port>
  <name>message_history</name>
  <exclusiveProvider>false</exclusiveProvider>
  <exclusiveUser>false</exclusiveUser>
  <optional>true</optional>
  <provider>true</provider>
  <supportedType>IDL:History:1.0</supportedType>
  <specificType>IDL:History:1.0</specificType>
  <kind>Facet</kind>
</port>
```

This is the `History` facet called `message_history`. Its declaration is nearly identical to that of the to the `Runnable` and `Publication` facets.

```
<configProperty>
  <name>subject</name>
  <value>
    <type>
      <kind>tk_string</kind>
    </type>
    <value>
      <string>Default Subject</string>
    </value>
  </value>
</configProperty>
```

This `<configProperty>` element sets a default value for the Messenger's `subject` attribute. Both the "Deployment and Configuration" specification and the `Deployment.xsd` schema file contain more information on representing data types and values in XML descriptors.

Note *CIAO currently ignores `<configProperty>` elements that set values for IDL attributes.*

```
</Deployment:ComponentInterfaceDescription>
```

The descriptor also recognizes one or more optional `<infoProperty>` elements that provide non-functional information that might be displayed by a tool as explained above.

In summary, the Messenger's CORBA Component Descriptor file, `Messenger.ccd`, is an XML rendition of the component's IDL3 interface. It contains a `<property>` element for each component attribute and a `<port>` element for each port.

Messenger Component - Component Implementation Descriptor (.cid)

The Messenger's Component Implementation Descriptor describes the *monolithic implementation* of the Messenger component. A monolithic implementation consists of a set of implementation artifacts, or libraries. (By contrast, an *assembly implementation* is a component implementation that consists of subcomponents). Our monolithic Messenger implementation pulls together the Messenger's three dynamic libraries—`Messenger_stub`, `Messenger_svnt`, and `Messenger_exec`.

The Messenger component's Component Implementation Descriptor follows. Comments are interspersed.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentImplementationDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Implementation</label>
```

The optional `<label>` element describes the implementation. A tool may use it for display purposes.

```
<implements href="Messenger.ccd"/>
```

The `<implements>` element describes the interface that the component implements by referencing the component's CORBA Component Descriptor file.

```
<monolithicImpl>
  <primaryArtifact>
    <name>Messenger_Stub</name>
    <referencedArtifact href="Messenger_Stub.iad"/>
  </primaryArtifact>
  <primaryArtifact>
    <name>Messenger_Svnt</name>
    <referencedArtifact href="Messenger_Svnt.iad"/>
  </primaryArtifact>
  <primaryArtifact>
    <name>Messenger_Exec</name>
    <referencedArtifact href="Messenger_Exec.iad"/>
  </primaryArtifact>
</monolithicImpl>
```

The `<monolithicImpl>` element pulls together the Messenger's three libraries. Each library is a `<primaryArtifact>` represented by a reference to an Implementation Artifact Descriptor file.

```
<configProperty>
  <name>ComponentIOR</name>
  <value>
    <type>
      <kind>tk_string</kind>
    </type>
    <value>
      <string>Messenger.ior</string>
    </value>
  </value>
</configProperty>
```

CIAO supports one optional `<configProperty>`, the `ComponentIOR` property, for a component implementation. At run time, the component server writes the component's object reference to the file indicated by the `ComponentIOR` property value. By default, the component server writes the

file to the directory from which it was launched. A non-CCM CORBA client may use that IOR file to discover the component.

```
</Deployment:ComponentImplementationDescription>
```

The Component Implementation Descriptor also accepts `<capability>` elements which can be used by the component server to choose between component implementations. It also accepts non-functional `<infoProperty>` elements as explained above.

In summary, the Messenger's Component Implementation Descriptor constructs a monolithic Messenger implementation from the Messenger's libraries.

Messenger Component - Component Package Descriptor (.cpd)

The Component Package Descriptor is the component's top-level packaging file. It can describe multiple alternative implementations of the same component interface and can contain configuration properties for the component.

In our example, we merely reference the component implementation defined in the previous subsection.

The Component Package Descriptor for the Messenger component is as follows, with comments interspersed:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentPackageDescription
xmlns:Deployment="http://www.omg.org/Deployment"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">

  <label>Messenger Component</label>
```

The optional `<label>` element is a human readable package label that may be used by a tool for display purposes.

```
  <realizes href="Messenger.ccd"/>
```

The `<realizes>` element indicates the component's IDL3 interface by referencing the component's CORBA Component Descriptor file.

```
<implementation>
  <name>MessengerImpl</name>
  <referencedImplementation href="Messenger.cid"/>
</implementation>
```

The `<implementation>` element references one or more Component Implementations Descriptors. Our example has just one Messenger implementation, so we refer to that implementation here. A more complex example may have multiple implementations and may use `<deployRequirement>` elements to enable a component server to choose between them.

```
</Deployment:ComponentPackageDescription>
```

The Component Package Descriptor also recognizes <infoProperty> documentation elements and `<configProperty>` default attribute value configuration elements.

Note *CIAO does not yet support the setting of a default attribute value through a* `<configProperty>` *element, so this value is currently ignored.*

To summarize, the Component Package Descriptor is the component's top-level descriptor, representing the component to the rest of the application.

Messenger Component - Summary

The table summarizes the six Messenger component descriptor files

Table 32-5 : Messenger Descriptor Files

File	Description
`Messenger_stub.iad`	Implementation Artifact Descriptor for the Messenger's stub library
`Messenger_svnt.iad`	Implementation Artifact Descriptor for the Messenger's servant library
`Messenger_exec.iad`	Implementation Artifact Descriptor for the Messenger's executor library
`Messenger.ccd`	CORBA Component Descriptor for the Messenger's IDL3 interface
`Messenger.cid`	Component Implementation Descriptor describing the Messenger's implementation in terms of its libraries

Table 32-5 : Messenger Descriptor Files

File	Description
`Messenger.cpd`	Component Package Descriptor packaging the Messenger component into one deployable package.

32.2.5.5 Receiver Component Descriptors

The Receiver component type has a similar set of descriptor files, as illustrated by the diagram.

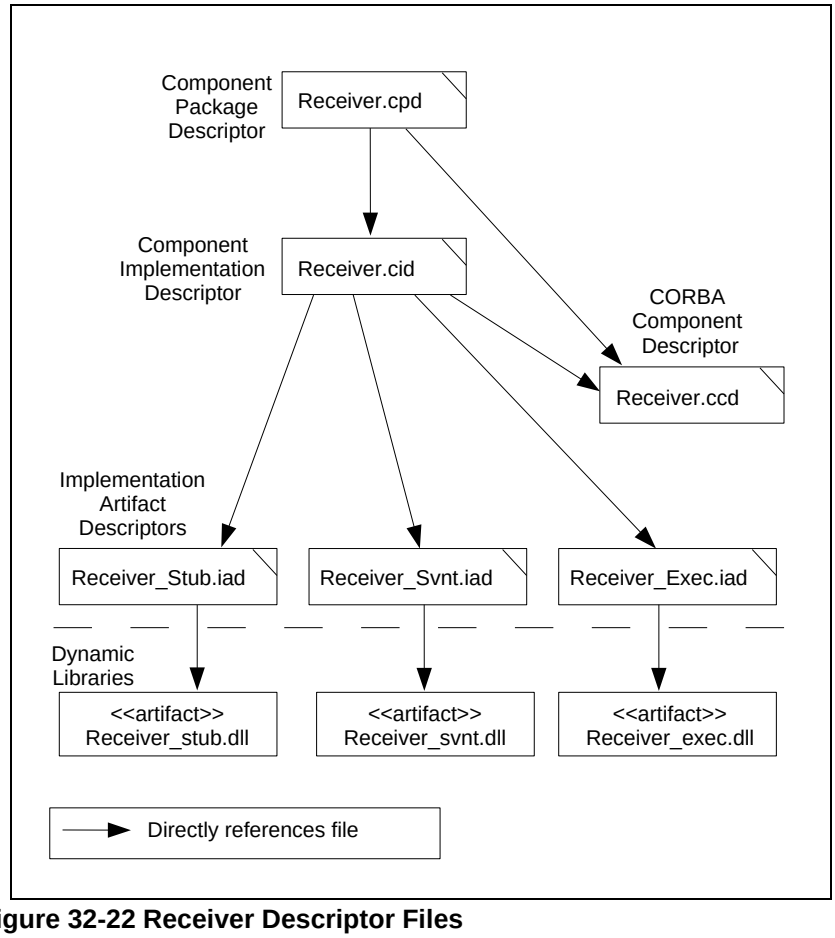

Figure 32-22 Receiver Descriptor Files

The primary differences between the Receiver's and the Messenger's descriptor files are in the Implementation Artifact Descriptor and the CORBA Component Descriptor.

The Receiver's three Implementation Artifact Descriptors reflect the fact that each of the Receiver's three libraries has a dependency on the Messenger's stub library. The dependency is illustrated by the `Receiver_stub.iad` file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ImplementationArtifactDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Receiver Stub Artifact</label>
  <location>Receiver_stub</location>
  <dependsOn>
    <name>ACE/TAO/CIAO</name>
    <referencedArtifact href="Libraries.iad"/>
  </dependsOn>
  <dependsOn>
    <name>Messenger_Stub</name>
    <referencedArtifact href="Messenger_Stub.iad"/>
  </dependsOn>
</Deployment:ImplementationArtifactDescription>
```

The Receiver's servant and executor Implementation Artifact Descriptor files are analogous to the Messenger's with the additional dependency on the `Messenger_stub` library. We do not show them here.

The Receiver's CORBA Component Descriptor file describes the Receiver's IDL3 interface, which is as follows:

```
component Receiver {
  consumes Message message_consumer;
  uses History message_history;
};
```

The CORBA Component Descriptor describes the Receiver's two ports, an `EventConsumer` and a `SimplexReceptacle`. It is as follows, with comments interspersed:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentInterfaceDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
```

```
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Receiver Component</label>
  <specificType>IDL:Receiver:1.0</specificType>
  <supportedType>IDL:Receiver:1.0</supportedType>
  <idlFile>Receiver.idl</idlFile>

  <port>
    <name>message_consumer</name>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>false</exclusiveUser>
    <optional>false</optional>
    <provider>true</provider>
    <supportedType>IDL:Message:1.0</supportedType>
    <specificType>IDL:Message:1.0</specificType>
    <kind>EventConsumer</kind>
  </port>
```

The `message_consumer` port is an `EventConsumer` that consumes `Message` events. The `<supportedType>` and `<specificType>` elements contain the Interface Repository Id of the published event type.

```
  <port>
    <name>message_history</name>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>true</exclusiveUser>
    <optional>true</optional>
    <provider>false</provider>
    <supportedType>IDL:History:1.0</supportedType>
    <specificType>IDL:History:1.0</specificType>
    <kind>SimplexReceptacle</kind>
  </port>
```

The `message_history` port uses the Messenger's `History` interface. The `uses` keyword in the component's interface indicates that it is a `SimplexReceptacle`, meaning that the receptacle connects to exactly one `History` facet. The `message_consumer` port may receive `Message` events from multiple publishers, but the `message_history` port may only retrieve the `History` from one provider.

```
</Deployment:ComponentInterfaceDescription>
```

The table summarizes the six Receiver descriptor files.

Table 32-6 Receiver Descriptor Files

File	Description
`Receiver_stub.iad`	Implementation Artifact Descriptor for the Receiver's stub library
`Receiver_svnt.iad`	Implementation Artifact Descriptor for the Receiver's servant library
`Receiver_exec.iad`	Implementation Artifact Descriptor for the Receiver's executor library
`Receiver.ccd`	CORBA Component Descriptor for the Receiver's IDL3 interface
`Receiver.cid`	Component Implementation Descriptor describing the Receiver's implementation in terms of its libraries
`Receiver.cpd`	Component Package Descriptor packaging the Receiver component into one deployable package.

32.2.5.6 Administrator Component Descriptors

The Administrator component type also has a similar set of descriptor files, as illustrated in the diagram.

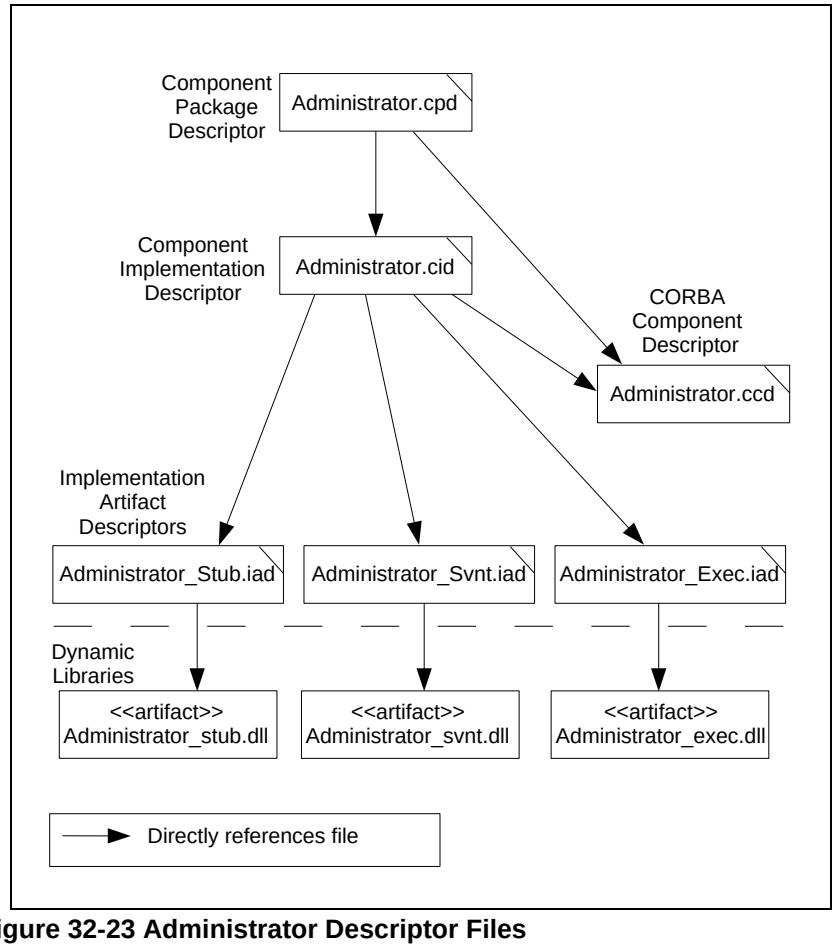

Figure 32-23 Administrator Descriptor Files

Like the Receiver, each of the Administrator's libraries depends on the Messenger's `Messenger_stub` library. We won't replicate the Implementation Artifact Descriptor files here.

The Administrator's CORBA Component Descriptor file describes the Administrator's IDL3 interface, which is as follows:

```
component Administrator {
```

```
  uses multiple Runnable runnables;
  uses multiple Publication content;
};
```

The Administrator's CORBA Component Descriptor describes the two Administrator `MultiplexReceptacle` ports. It is as follows, with comments interspersed:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentInterfaceDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Administrator Component</label>
  <specificType>IDL:Administrator:1.0</specificType>
  <supportedType>IDL:Administrator:1.0</supportedType>
  <idlFile>Administrator.idl</idlFile>

  <port>
    <name>runnables</name>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>true</exclusiveUser>
    <optional>true</optional>
    <provider>false</provider>
    <supportedType>IDL:Runnable:1.0</supportedType>
    <specificType>IDL:Runnable:1.0</specificType>
    <kind>MultiplexReceptacle</kind>
  </port>
```

The `runnables` port uses the Messenger's `Runnable` interface. The `uses multiple` keyword in the IDL3 interface indicates that it is a `MultiplexReceptacle`, meaning that it may connect to many `Runnable` facets.

```
  <port>
    <name>content</name>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>true</exclusiveUser>
    <optional>true</optional>
    <provider>false</provider>
    <supportedType>IDL:Publication:1.0</supportedType>
    <specificType>IDL:Publication:1.0</specificType>
    <kind>MultiplexReceptacle</kind>
  </port>
```

The `content` port uses the Messenger's `Publication` interface. It is also a `MultiplexReceptacle`, meaning that it may connect to many `Publication` facets.

```
</Deployment:ComponentInterfaceDescription>
```

The table summarizes the six Administrator descriptor files.

Table 32-7 Administrator Descriptor Files

File	Description
`Administrator_stub.iad`	Implementation Artifact Descriptor for the Administrator's stub library
`Administrator_svnt.iad`	Implementation Artifact Descriptor for the Administrator's servant library
`Administrator_exec.iad`	Implementation Artifact Descriptor for the Administrator's executor library
`Administrator.ccd`	CORBA Component Descriptor for the Administrator's IDL3 interface
`Administrator.cid`	Component Implementation Descriptor describing the Administrator's implementation in terms of its libraries
`Administrator.cpd`	Component Package Descriptor packaging the Administrator component into one deployable package.

32.2.5.7 Messenger Assembly Descriptors

An *assembly* is a component implementation that consists of a set of subcomponent instances connected through their ports.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component's composition
  - Implement a C++ executor for each component and facet
- Describe the application's deployment
  - Describe each component's libraries and ports
  - **Connect component instances through their ports**
  - Deploy each component into a component container
- Build the application
- Run the application

Figure 32-24 Road Map

We package the Messenger, Receiver, and Application components into one top-level component we refer to as the Messenger Assembly. Our Messenger Assembly consists of one Messenger component instance, two Receiver component instances, and one Administrator component instance.

The Messenger Assembly's deployment is described by three descriptor files: a CORBA Component Descriptor describing the assembly's exposed properties and ports; a Component Implementation Descriptor describing the assembly's implementation in terms of its subcomponent instances and the connections between them; and a Component Package Descriptor that

packages the assembly into a deployable component. The relationships between the descriptor files are illustrated in the diagram.

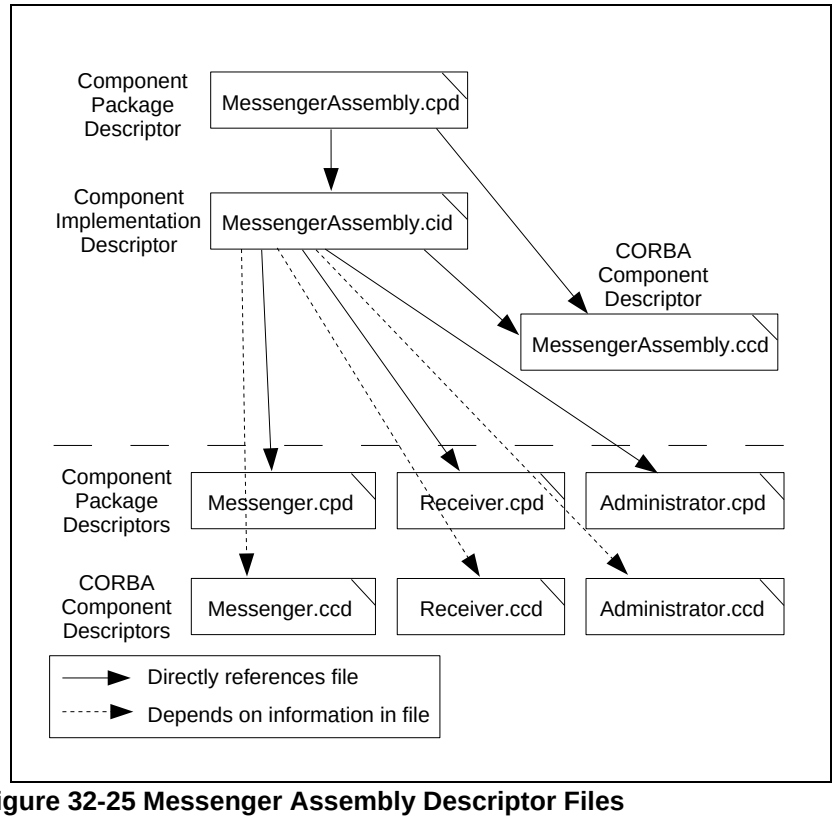

Figure 32-25 Messenger Assembly Descriptor Files

A *CORBA Component Descriptor* describes the Messenger Assembly's public interface. An assembly may expose ports and attributes of its subcomponents to the outside world. The Messenger Assembly exposes the Messenger component's subject attribute, but does not expose any Messenger, Receiver, or Administrator ports.

A *Component Implementation Descriptor* describes the Messenger Assembly's implementation. Its implementation is composed of one instance of the Messenger component, two instances of the Receiver component, and one instance of the Administrator component. The Component Implementation Descriptor describes how the component instances' facets and event publishers connect to receptacles and event consumers to comprise the assembly.

A *Component Package Descriptor* can describe many alternate implementations of the assembly. Our example provides one implementation of the Messenger Assembly.

Messenger Assembly - CORBA Component Descriptor

A *CORBA Component Descriptor* describes a component's ports and attributes. An assembled component may expose ports or attributes of its subcomponents. Our Messenger Assembly merely exposes the Messenger component's subject attribute.

The Messenger Assembly's CORBA Component Descriptor follows.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentInterfaceDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Assembly</label>

  <property>
    <name>subject</name>
    <type>
      <kind>tk_string</kind>
    </type>
  </property>

</Deployment:ComponentInterfaceDescription>
```

The `<property>` element indicates that the assembly exposes an attribute called "subject", whose type is a string. The assembly's Component Implementation Descriptor file, described in the next section, defines how the assembly's subject attribute is mapped to the subject attribute of its Messenger subcomponent.

We can think of the Messenger Assembly as a component whose *implied* IDL3 interface is the following:

```
component MessengerAssembly {
  attribute string subject;
};
```

Messenger Assembly - Component Implementation Descriptor

The Messenger Assembly's Component Implementation Descriptor describes the subcomponent instances that comprise the assembly and the connections between their ports. The Component Implementation Descriptor describes the subcomponent instances and connections as shown in the deployment diagram.

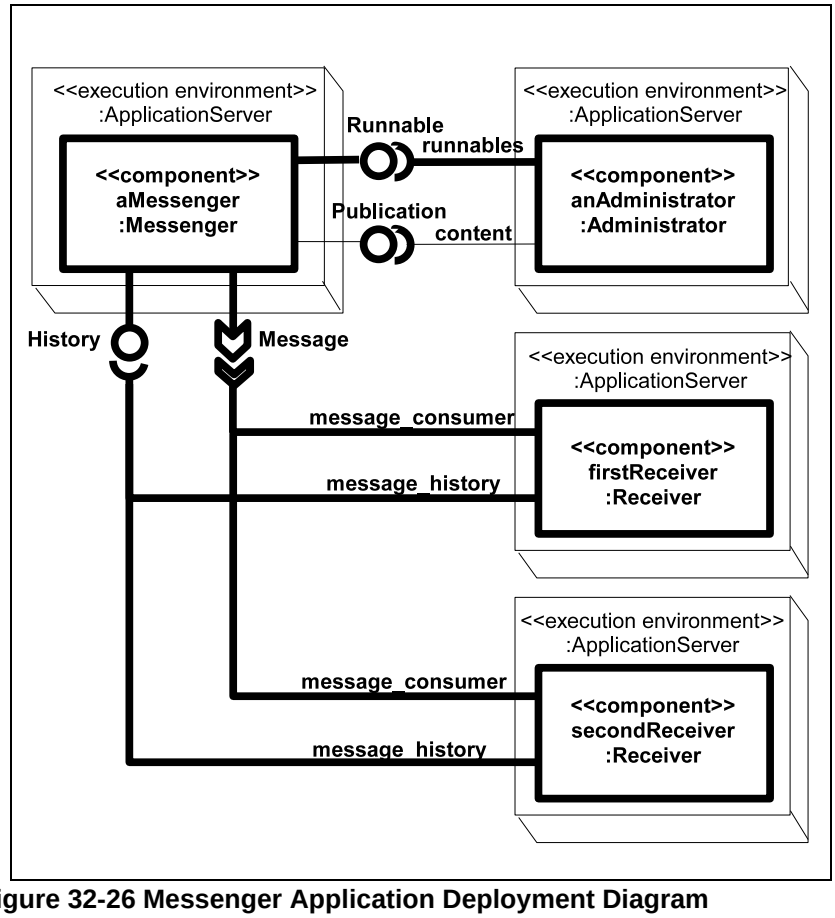

Figure 32-26 Messenger Application Deployment Diagram

The Messenger Assembly's Component Implementation Descriptor is as follows, with comments interspersed.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentImplementationDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
```

```
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
 <label>Messenger Assembly</label>
 <implements href="MessengerAssembly.ccd"/>
```

The `<implements>` element references the Messenger Assembly's CORBA Component Descriptor documented in the previous section.

```
 <assemblyImpl>
```

An `<assemblyImpl>` element indicates that this is an assembly-based component, meaning that is composed of subcomponent instances.

```
  <instance xmi:id="a_Messenger">
    <name>Messenger_Instance</name>
    <package href="Messenger.cpd"/>
  </instance>
  <instance xmi:id="first_Receiver">
    <name>First_Receiver_Instance</name>
    <package href="Receiver.cpd"/>
  </instance>
  <instance xmi:id="second_Receiver">
    <name>Second_Receiver_Instance</name>
    <package href="Receiver.cpd"/>
  </instance>
  <instance xmi:id="a_Administrator">
    <name>Administrator_Instance</name>
    <package href="Administrator.cpd"/>
  </instance>
```

The `<instance>` elements create the Messenger instance, the two Receiver instances, and the Administrator instance. Each `<instance>` refers to the Component Package Descriptor of its component type. The `xml::id` attributes of the instances are used by `<connection>` elements to connect the instances' ports.

```
  <connection>
    <name>Messenger_to_First_Receiver_Publish</name>
    <internalEndpoint>
      <portName>message_publisher</portName>
      <instance xmi:idref="a_Messenger"/>
    </internalEndpoint>
    <internalEndpoint>
      <portName>message_consumer</portName>
      <instance xmi:idref="first_Receiver"/>
```

```
    </internalEndpoint>
  </connection>
```

This connection connects the Messenger instance's `message_publisher` port to one Receiver instance's `message_consumer` port. The connection's `<name>` is a unique identifier for the connection within the assembly. The value in each `<portName>` element must match the port name in the Messenger's and Receiver's CORBA Component Descriptor files. The `<instance>` element's `xml::idref` attribute matches the `<instance>` element's `xml::id` attribute above.

```
<connection>
  <name>Messenger_to_First_Receiver_History</name>
  <internalEndpoint>
    <portName>message_history</portName>
    <instance xmi:idref="a_Messenger"/>
  </internalEndpoint>
  <internalEndpoint>
    <portName>message_history</portName>
    <instance xmi:idref="first_Receiver"/>
  </internalEndpoint>
</connection>
```

This connection connects the Messenger's `message_history` facet to a Receiver instance's `message_history` receptacle.

The `message_publisher`-to-`message_consumer` connection and the `message_history`-to-`message_history` connection are illustrated by the highlights in the deployment diagram.

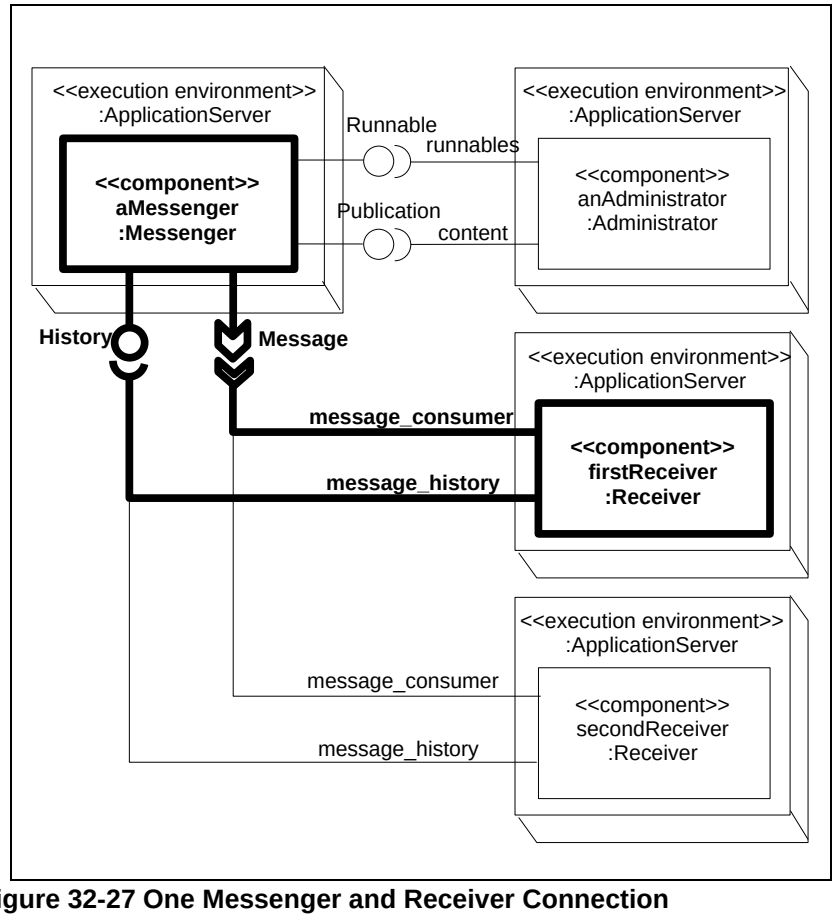

Figure 32-27 One Messenger and Receiver Connection

```
<connection>
  <name>Messenger_to_Second_Receiver_Publisher</name>
  <internalEndpoint>
    <portName>message_publisher</portName>
    <instance xmi:idref="a_Messenger"/>
  </internalEndpoint>
  <internalEndpoint>
    <portName>message_consumer</portName>
    <instance xmi:idref="second_Receiver"/>
  </internalEndpoint>
```

```
</connection>
<connection>
  <name>Messenger_to_Second_Receiver_History</name>
  <internalEndpoint>
    <portName>message_history</portName>
    <instance xmi:idref="a_Messenger"/>
  </internalEndpoint>
  <internalEndpoint>
    <portName>message_history</portName>
    <instance xmi:idref="second_Receiver"/>
  </internalEndpoint>
</connection>
```

These two connections connect the second Receiver instance to the Messenger instance. The Messenger instance's `message_publisher` port connects to the Receiver instance's `message_consumer` port and the Messenger instance's `message_history` facet connects to the Receiver instance's `message_history` receptacle.

```
<connection>
  <name>Messenger_to_Administrator_Control</name>
  <internalEndpoint>
    <portName>control</portName>
    <instance xmi:idref="a_Messenger"/>
  </internalEndpoint>
  <internalEndpoint>
    <portName>runnables</portName>
    <instance xmi:idref="a_Administrator"/>
  </internalEndpoint>
</connection>
<connection>
  <name>Messenger_to_Administrator_Content</name>
  <internalEndpoint>
    <portName>content</portName>
    <instance xmi:idref="a_Messenger"/>
  </internalEndpoint>
  <internalEndpoint>
    <portName>content</portName>
    <instance xmi:idref="a_Administrator"/>
  </internalEndpoint>
</connection>
```

These two connections connect the Administrator instance to the Messenger instance. The Messenger instance's `control` facet connects to the Administrator instance's `runnables` receptacle and the Messenger instance's `content` facet connects to the Administrator instance's `content` receptacle.

```
    <externalProperty>
        <name>Subject Mapping</name>
        <externalName>subject</externalName>
        <delegatesTo>
            <propertyName>subject</propertyName>
            <instance xmi:idref="a_Messenger"/>
        </delegatesTo>
    </externalProperty>
```

The `<externalProperty>` element maps the Messenger Assembly's exposed `subject` attribute to the Messenger component instance's `subject` attribute. The Messenger Assembly doesn't implement its own subject attribute; it must map to an attribute of one of its subcomponents.

```
  </assemblyImpl>
</Deployment:ComponentImplementationDescription>
```

In summary, the Messenger Assembly's Component Implementation Descriptor describes the four subcomponent instances that comprise the assembly and describes the connections that connect their ports together.

Messenger Assembly - Component Package Descriptor

The Messenger Assembly's Component Package Descriptor is the top-level descriptor that represents the assembly as a deployable component. That should sound familiar; the Messenger, Receiver, and Administrator Component Package Descriptors serve exactly the same purpose.

The Messenger Assembly's Component Package Descriptor is nearly identical to the Messenger component's.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentPackageDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Assembly Package</label>
  <realizes href="MessengerAssembly.ccd"/>
  <implementation>
    <name>Messenger Application</name>
    <referencedImplementation href="MessengerAssembly.cid"/>
  </implementation>
</Deployment:ComponentPackageDescription>
```

The `<realizes>` element references the assembly's CORBA Component Descriptor, which describes the assembly's implied IDL3 interface. The `<referencedImplementation>` element references the assembly's Component Implementation Descriptor, which describes the assembly's implementation in terms of its subcomponents.

Messenger Assembly - Summary

The Messenger Assembly's descriptors files describe the Messenger Assembly' composition in terms of its subcomponent instances the connections between them. The table summarizes the Messenger Assembly descriptor files.

Table 32-8 Messenger Assembly Descriptor Files

File	Description
`MessengerAssembly.ccd`	CORBA Component Descriptor for the Messenger Assembly's implied IDL3 interface
`MessengerAssembly.cid`	Component Implementation Descriptor describing the Messenger Assembly's implementation in terms of its subcomponent instances and connections
`MessengerAssembly.cpd`	Component Package Descriptor packaging the Messenger Assembly component into one deployable package.

32.2.5.8 Application Descriptors

The application's deployment descriptors describe how the assembly's component instances are deployed onto logical nodes and how each logical node is mapped to a physical component container.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component's composition
  - Implement a C++ executor for each component and facet
- Describe the application's deployment
  - Describe each component's libraries and ports
  - Connect component instances through their ports
  - **Deploy each component into a component container**
- Build the application
- Run the application

Figure 32-28 Road Map

The application's UML Deployment Diagram illustrates the deployment.

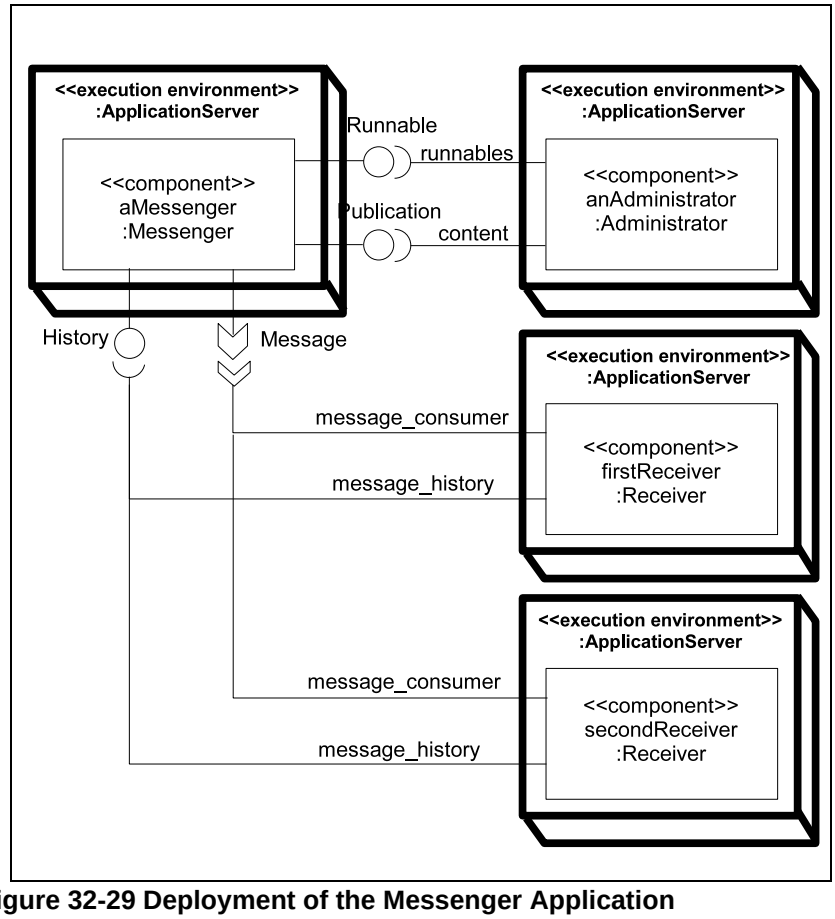

Figure 32-29 Deployment of the Messenger Application

The Application's deployment is described by five descriptor files: a Package Configuration Descriptor that wraps the Messenger Assembly's package descriptor; a Top-level Package Descriptor that represents the entire application; a Component Deployment Plan descriptor that maps the Message Assembly's subcomponent instances to logical deployment nodes; a Component Domain Descriptor that describes each of the logical nodes; and a node map that maps logical deployment nodes to physical component server processes.

The *Package Configuration Descriptor* describes one possible configuration of a component package by indicating deployment requirements and/or configuration properties of the component package.

The *Top-level Package Descriptor* references the Package Configuration Descriptor that is the root of the deployed application. It always has the name `package.tpd`.

The *Component Deployment Plan* contains the bulk of the application's deployment information. It maps each component instance onto a logical deployment node, achieving a separation of concerns between the Component Implementation Descriptor's instance connections and the Component Deployment Plan's node mappings.

The *Component Domain Descriptor* describes the target deployment environment in terms of its nodes, interconnects, and bridges.

The *Node Map* maps each logical node onto a physical component server process.

The relationships between the application descriptor files are illustrated in the diagram.

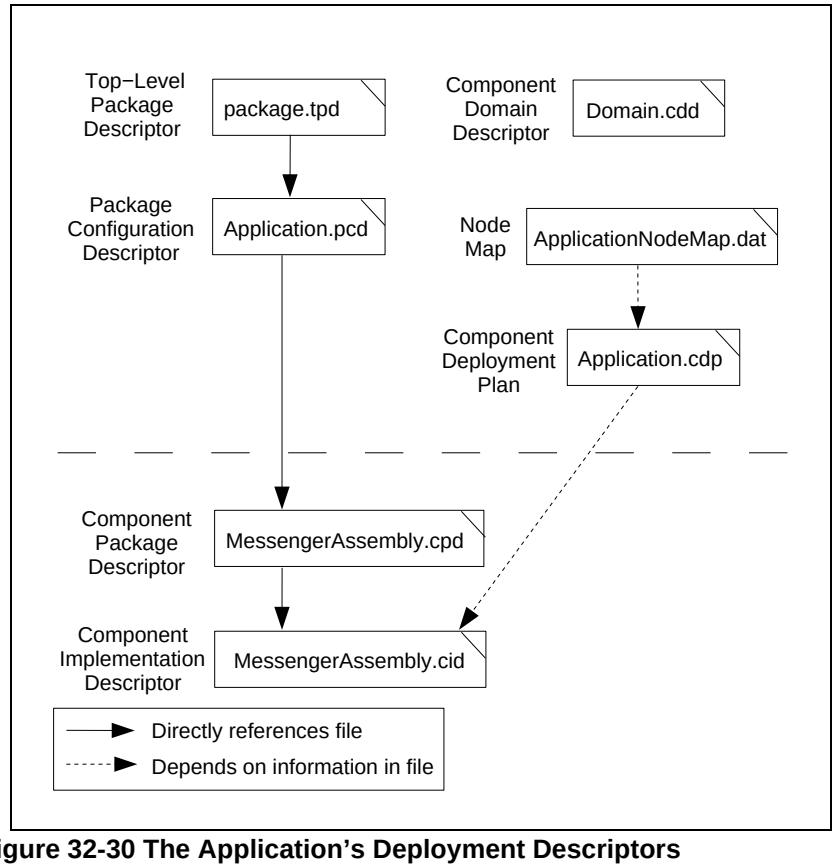

Figure 32-30 The Application's Deployment Descriptors

Application - Package Configuration Descriptor

The Package Configuration Descriptor describes one configuration of a deployable component package. It may define deployment requirements or configure attribute values. Our Package Configuration Descriptor references the Messenger Assembly's Component Package Descriptor and configures a default value for the Messenger Assembly's exposed `subject` attribute. The descriptor is as follows, with comments interspersed:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:PackageConfiguration
xmlns:Deployment="http://www.omg.org/Deployment"
```

```
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Application Configuration</label>
  <basePackage href="MessengerAssembly.cpd"/>
```

The `<basePackage>` element references the Messenger Assembly's Component Package Descriptor. Either a `<basePackage>` element or a `<specializedConfig>` element is mandatory. A `<specializedConfig>` element can reference another Package Configuration Descriptor and override its requirements and/or configuration values.

```
  <configProperty>
    <name>subject</name>
    <value>
      <type>
        <kind>tk_string</kind>
      </type>
      <value>
        <string>Typewriter practice</string>
      </value>
    </value>
  </configProperty>
```

The `<configProperty>` element defines a default value for the Messenger Assembly's `subject` attribute.

Note *CIAO does not yet support the setting of a default attribute value through a* `<configProperty>` *element, so this value is currently ignored.*

```
</Deployment:PackageConfiguration>
```

A Package Configuration Descriptor may also contain `<selectRequirement>` requirement elements. In future implementations of CIAO, these elements would be matched against `<capability>` elements in the Component Implementation Description.

Application - Top-level Package Descriptor

Each application has exactly one Top-level Package Descriptor. It is always named `package.tpd`, and it points to the application's Package

Configuration Descriptor file. The Top-level Package Descriptor for our application is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:TopLevelPackageDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <package href="Application.pcd"/>
</Deployment:TopLevelPackageDescription>
```

A Top-level Package Descriptor has exactly one `<package>` element that points to the application's Package Configuration Descriptor.

Application - Component Deployment Plan

The application's Component Deployment Plan describes how the Messenger Assembly's component instances are deployed onto logical processing nodes. Through this descriptor, the application deployer can vary the component instance-to-node mapping independently from the connections between component instances.

The Component Deployment Plan for our application is as follows, with comments interspersed:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:DeploymentPlan
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Deployment Plan</label>
  <instance xmi:id="Messenger_Instance_ID">
    <name>Messenger_Instance</name>
    <node>Messenger_Node</node>
  </instance>
```

This `<instance>` element indicates that the `Messenger_Instance` is deployed onto the `Messenger_Node`. The value of the `<name>` element must match the Messenger instance's `<name>` element in the Messenger Assembly's Component Implementation Descriptor file.

```
  <instance xmi:id="First_Receiver_Instance_ID">
    <name>First_Receiver_Instance</name>
```

```
    <node>First_Receiver_Node</node>
  </instance>
  <instance xmi:id="Second_Receiver_Instance_ID">
    <name>Second_Receiver_Instance</name>
    <node>Second_Receiver_Node</node>
  </instance>
```

These two `<instance>` elements indicate that each of the two Receiver instances is deployed on its own logical node. Again, the value of each `<name>` element must match the Receiver instances' `<name>` elements in the Messenger Assembly's Component Implementation Descriptor file.

```
  <instance xmi:id="Administrator_Instance_ID">
    <name>Administrator_Instance</name>
    <node>Administrator_Node</node>
  </instance>
```

This `<instance>` element indicates that the Administrator instance is deployed onto the `Administrator_Node`.

```
</Deployment:DeploymentPlan>
```

In summary, the Component Deployment Plan maps each component instance onto a logical processing node. Each instance name must match an instance name in the assembly's Component Implementation Descriptor.

Application - Component Domain Descriptor

The Component Domain Descriptor describes the target environment in terms of its nodes, interconnects, and bridges. The Messenger application's Component Domain Descriptor is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Deployment:Domain
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Application Domain</label>
  <node>
    <name>Messenger_Node</name>
    <label>Messenger's Node</label>
  </node>
  <node>
    <name>First_Receiver_Node</name>
```

```
    <label>First Receiver's Node</label>
  </node>
  <node>
    <name>Second_Receiver_Node</name>
    <label>Second Receiver's Node</label>
  </node>
  <node>
    <name>Administrator_Node</name>
    <label>Administrator's Node</label>
  </node>
</Deployment:Domain>
```

This Component Domain Descriptor describes the four nodes of the Messenger application. The `<name>` of each `<node>` matches the node name in the Component Deployment Plan. The `<label>`, as always, is optional and may be used for display purposes by a tool. Our sample application doesn't use this descriptor, but we define it for completeness.

Node Map

The node map is a text file that maps each of the Component Deployment Plan's logical nodes onto a physical component server process by mapping each logical node to a `NodeManager` object reference. The node map for the Messenger application is as follows:

```
Administrator_Node   corbaloc:iiop:localhost:10000/NodeManager
First_Receiver_Node  corbaloc:iiop:localhost:20000/NodeManager
Second_Receiver_Node corbaloc:iiop:localhost:30000/NodeManager
Messenger_Node       corbaloc:iiop:localhost:40000/NodeManager
```

The contents of this file determine where each component executes. Our deployment environment consists of four `NodeManager` processes running on the `localhost`, each listening on a different port. Each `NodeManager` is a component server, capable of dynamically loading a component's libraries and making connections between components. The `NodeManager` is documented in 32.2.7 and 32.4.

The node map enables a great deal of deployment flexibility. We could deploy the Messenger application across a network simply by running our NodeManager processes across the network and changing our node map's `NodeManager` object references to reflect that.

We could also deploy several component instances on one component server simply by mapping several logical nodes to the same `NodeManager` object reference.

Messenger Application - Summary

The Messenger Application's descriptors describe how each subcomponent instance is deployed onto a physical component server process. The table summarizes the Messenger application's descriptor files.

Table 32-9 Messenger Application Descriptor Files

File	Description
`Application.pcd`	Package Configuration Descriptor that configures the top-level component's deployment attributes.
`package.tpd`	Top-level Package Descriptor that represents the application.
`Application.cdp`	Component Deployment Plan that maps component instances to logical nodes.
`Domain.cdd`	Component Domain Descriptor that describes the target deployment environment.
`ApplicationNodeMap.dat`	Text file that maps each logical node to a physical component server object.

The following sections discusses the execution of the Messenger application.

32.2.6 Building the Messenger Application

The Messenger application consists of three component types: Messenger, Receiver, and Administrator. Each component type is composed of three libraries: a stub library, a servant library, and an executor library. We manage

these libraries with three Make Project Creator (MPC) files, one file for each component type.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component's composition
  - Implement a C++ executor for each component and facet
- Describe the application's deployment
  - Describe each component's libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- **Build the application**
- Run the application

Figure 32-31 Road Map

The example's source code, build files, and XML descriptor files ares in the `$CIAO_ROOT/examples/DevGuideExamples/Messenger` directory.

32.2.6.1 Setting Up Your Environment

There are several environment variables used by ACE, TAO, and CIAO during both the compilation and execution of applications. Information about ACE and TAO environment variables is available at 3.2. CIAO's environment variables are described below. Syntax for Windows is shown in parentheses.

- `CIAO_ROOT`

  The base path for all CIAO-related code, normally `$TAO_ROOT/CIAO` (`%TAO_ROOT%\CIAO`).

- `XERCESCROOT`

  The base directory of the Xerces C++ installation. See 32.3.1 and 32.3.2 for more information on Xerces C++.

- Library Path

  The library path must include the directory containing the Xerces C++ dynamic libraries, `$XERCESCROOT/bin` (`%XERCESCROOT%\bin`). You should add this location to your `LD_LIBRARY_PATH` environment variable

or its equivalent. (On Windows, add this directory to your PATH so DLLs can be located at run time.)

32.2.6.2 Creating the Messenger's MPC File

The CIAO source tree contains a generate_component_mpc.pl script to generate the beginning of a component's MPC file.

UNIX and UNIX-like Systems

The script is $CIAO_ROOT/bin/generate_component_mpc.pl

Windows Systems

The script is %CIAO_ROOT%\bin\generate_component_mpc.pl.

General Usage

The general usage of the generate_component_mpc.pl script is as follows:

```
$CIAO_ROOT/bin/generate_component_mpc.pl <component name>
```

For example:

```
$CIAO_ROOT/bin/generate_component_mpc.pl Messenger
```

creates an MPC file called Messenger.mpc.

The script also prints the following text to indicate that it generated the export header file for each Messenger library:

```
The following commands have been executed:
   generate_export_file.pl MESSENGER_STUB > Messenger_stub_export.h
   generate_export_file.pl MESSENGER_SVNT > Messenger_svnt_export.h
   generate_export_file.pl MESSENGER_EXEC > Messenger_exec_export.h
```

We deploy our component executors in dynamic libraries. On Windows platforms, classes exported from a dynamic library must define an export macro. On UNIX-like platforms, the export macros define to nothing. However, the 4.x versions of the gcc compiler support the C++ export keyword, which may reduce code size by reducing the number of exported symbols. In either case, the export macros enable cross-platform development.

See 5.12 for more information on the generate_export_file.pl script.

The Messenger's MPC File

The generated MPC file is as follows:

```
project(Messenger_stub): ccm_stub {

  sharedname = Messenger_stub
  idlflags += -Wb,stub_export_macro=MESSENGER_STUB_Export
  idlflags += -Wb,stub_export_include=Messenger_stub_export.h
  idlflags += -Wb,skel_export_macro=MESSENGER_SVNT_Export
  idlflags += -Wb,skel_export_include=Messenger_svnt_export.h
  dynamicflags   = MESSENGER_STUB_BUILD_DLL

  IDL_Files {
    Messenger.idl
  }

  Source_Files {
    MessengerC.cpp
  }
}

project(Messenger_svnt) : ciao_servant {
  after +=  Messenger_stub
  sharedname  = Messenger_svnt
  libs    += Messenger_stub
  idlflags +=  -Wb,export_macro=MESSENGER_SVNT_Export
  idlflags +=  -Wb,export_include=Messenger_svnt_export.h
  dynamicflags = MESSENGER_SVNT_BUILD_DLL

  CIDL_Files {
    Messenger.cidl
  }

  IDL_Files {
    MessengerE.idl
  }

  Source_Files {
    MessengerEC.cpp
    MessengerS.cpp
    Messenger_svnt.cpp
  }
}

project(Messenger_exec) : ciao_executor {
  after   += Messenger_svnt
  sharedname = Messenger_exec
  libs      += Messenger_stub Messenger_svnt
  idlflags +=  -Wb,export_macro=MESSENGER_EXEC_Export
```

```
  idlflags +=  -Wb,export_include=Messenger_exec_export.h
  dynamicflags   = MESSENGER_EXEC_BUILD_DLL

  IDL_Files {
  }

  Source_Files {
    Messenger_exec.cpp
  }
}
```

That's a reasonable start towards our final `Messenger.mpc` file. We edit the file as follows, with comments interspersed. First, we discuss the project for the Messenger's stub library, `Messenger_stub`.

```
project(Messenger_stub): ccm_stub {
```

Because we deploy the application using the DAnCE facility, we change the `ciao_client` base project dependency to `ciao_client_dnc`.

```
  sharedname  = Messenger_stub
  idlflags += -Wb,stub_export_macro=MESSENGER_STUB_Export
  idlflags += -Wb,stub_export_include=Messenger_stub_export.h
  idlflags += -Wb,skel_export_macro=MESSENGER_SVNT_Export
  idlflags += -Wb,skel_export_include=Messenger_svnt_export.h
  dynamicflags = MESSENGER_STUB_BUILD_DLL
```

We make no changes to the `sharedname`, `idlflags`, or `dynamicflags`.

```
  IDL_Files {
    Runnable.idl
    Publication.idl
    Message.idl
    History.idl
    Messenger.idl
  }
```

The Messenger component's interfaces, event types, and component declaration are spread across five IDL files. The

generate_component_mpc.pl script does not know this. Thus, we add four IDL files to the IDL_Files section.

```
Source_Files {
  RunnableC.cpp
  PublicationC.cpp
  MessageC.cpp
  HistoryC.cpp
  MessengerC.cpp
}
```

We add stub source code files for each of the Messenger component's IDL files.

```
}
```

Next, we discuss the project for the Messenger's servant library, Messenger_svnt.

```
project(Messenger_svnt): ciao_servant_dnc {
```

Again, because we deploy the application using the DAnCE facility, we change the ciao_servant base project dependency to ciao_servant_dnc.

```
after += Messenger_stub
sharedname  = Messenger_svnt
libs += Messenger_stub
idlflags += -Wb,export_macro=MESSENGER_SVNT_Export
idlflags += -Wb,export_include=Messenger_svnt_export.h
dynamicflags = MESSENGER_SVNT_BUILD_DLL
```

We make no changes to the after, sharedname, libs, idlflags, or dynamicflags.

```
// project must be a ciao_servant or ciao_server to use CIDL files
CIDL_Files {
  Messenger.cidl
```

```
  }

  IDL_Files {
    MessengerE.idl
  }
```

We make no changes to the `CIDL_Files` or the `IDL_Files`.

```
  Source_Files {
    RunnableS.cpp
    PublicationS.cpp
    MessageS.cpp
    HistoryS.cpp
    MessengerS.cpp
    MessengerEC.cpp
    Messenger_svnt.cpp
  }
```

We add skeleton source code files for the Messenger component's IDL files.

```
}
```

Finally, we discuss the project for the Messenger's executor library, `Messenger_exec`.

```
project(Messenger_exec): ciao_component_dnc {
```

Once more, because we deploy the application using the DAnCE facility, we change the `ciao_component` base project dependency to `ciao_component_dnc`.

```
  after += Messenger_svnt
  sharedname  = Messenger_exec
  libs     += Messenger_stub Messenger_svnt
  idlflags += -Wb,export_macro=MESSENGER_EXEC_Export
  idlflags += -Wb,export_include=Messenger_exec_export.h
  dynamicflags = MESSENGER_EXEC_BUILD_DLL
```

We make no changes to the `after`, `sharedname`, `libs`, `idlflags`, or `dynamicflags`.

```
IDL_Files {
}

Source_Files {
  MessengerES.cpp
  Messenger_exec_i.cpp
  Publication_exec_i.cpp
  History_exec_i.cpp
  Runnable_exec_i.cpp
}
```

We make quite a few changes to the executor library's `Source_Files` section. First, we add `MessengerES.cpp`, the Messenger executor's skeleton file. Then we change `Messenger_exec.cpp` to `Messenger_exec_i.cpp` to reflect the fact that we've renamed the CIDL-generated executor implementation file as described in 32.2.3. Finally, we add the facet executor implementation files.

```
}
```

32.2.6.3 Creating the Administrator's and Receiver's MPC Files

The Receiver's and Administrator's MPC files are similar to the Messenger's. We generate each file using the generate_component_mpc.pl script.

```
generate_component_mpc.pl Receiver
generate_component_mpc.pl Administrator
```

We modify the generated MPC files by hand, just as we did for the Messenger. We'll examine the modified `Receiver.mpc` file and highlight significant differences between `Receiver.mpc` and `Messenger.mpc`. Comments are interspersed. First, we discuss the project for the Receiver's stub library, `Receiver_stub`.

```
project(Receiver_stub): ccm_stub {

  after += Messenger_stub
  sharedname  = Receiver_stub
  libs += Messenger_stub
```

The Receiver's stub library is dependent on the Messenger's stub library and must be built after the Messenger's stub library.

```
  idlflags += -Wb,stub_export_macro=RECEIVER_STUB_Export
  idlflags += -Wb,stub_export_include=Receiver_stub_export.h
  idlflags += -Wb,skel_export_macro=RECEIVER_SVNT_Export
  idlflags += -Wb,skel_export_include=Receiver_svnt_export.h
  dynamicflags = RECEIVER_STUB_BUILD_DLL

  IDL_Files {
    Receiver.idl
  }

  Source_Files {
    ReceiverC.cpp
  }
}
```

Next, we discuss the project for the Receiver's servant library, `Receiver_svnt`.

```
project(Receiver_svnt): ciao_servant_dnc {


  after += Receiver_stub Messenger_svnt
  sharedname  = Receiver_svnt
  libs += Receiver_stub Messenger_stub Messenger_svnt
```

The Receiver's servant library is dependent on the Messenger's stub and servant libraries and must be built after the Messenger's servant library.

```
  idlflags += -Wb,export_macro=RECEIVER_SVNT_Export
  idlflags += -Wb,export_include=Receiver_svnt_export.h
  dynamicflags = RECEIVER_SVNT_BUILD_DLL

  CIDL_Files {
    Receiver.cidl
  }

  IDL_Files {
    ReceiverE.idl
  }

  Source_Files {
    ReceiverS.cpp
    ReceiverEC.cpp
    Receiver_svnt.cpp
```

```
  }
}
```

Finally, we discuss the project for the Receiver's executor library, `Receiver_exec`.

```
project(Receiver_exec): ciao_component_dnc {


  after += Receiver_svnt
  sharedname  = Receiver_exec
  libs += Receiver_stub Receiver_svnt Messenger_stub
```

The Receiver's executor library is dependent on the Messenger's stub library.

```
  idlflags += -Wb,export_macro=RECEIVER_EXEC_Export
  idlflags += -Wb,export_include=Receiver_exec_export.h
  dynamicflags = RECEIVER_EXEC_BUILD_DLL

  IDL_Files {
  }

  Source_Files {
    ReceiverES.cpp
    Receiver_exec_i.cpp
  }
```

The Receiver's executor library has just one executor implementation file.

```
}
```

The modified `Administrator.mpc` file is similar. The primary difference between the Administrator component and the Receiver component is that the Administrator is not an event consumer, and thus the Administrator's servant library does not depend on the Messenger's servant library.

32.2.6.4 Running MPC

Execute the Make Project Creator to generate the Messenger's build files for your platform. For example:

UNIX and UNIX-like Systems

```
cd $CIAO_ROOT/examples/DevGuideExamples/Messenger
```

```
$ACE_ROOT/bin/mwc.pl -type gnuace
```

Windows Systems

```
cd %CIAO_ROOT%\examples\DevGuideExamples\Messenger
perl %ACE_ROOT%\bin\mwc.pl -type vc71
```

See Chapter 4 for more information on MPC.

32.2.6.5 Building

Build the Messenger application using the target build environment.

32.2.7 Running the Messenger Application

Finally, we can execute the application.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component's composition
  - Implement a C++ executor for each component and facet
- Describe the application's deployment
  - Describe each component's libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application
- **Run the application**

Figure 32-32 Road Map

Recall that we deploy the Messenger application on four nodes as illustrated in the diagram.

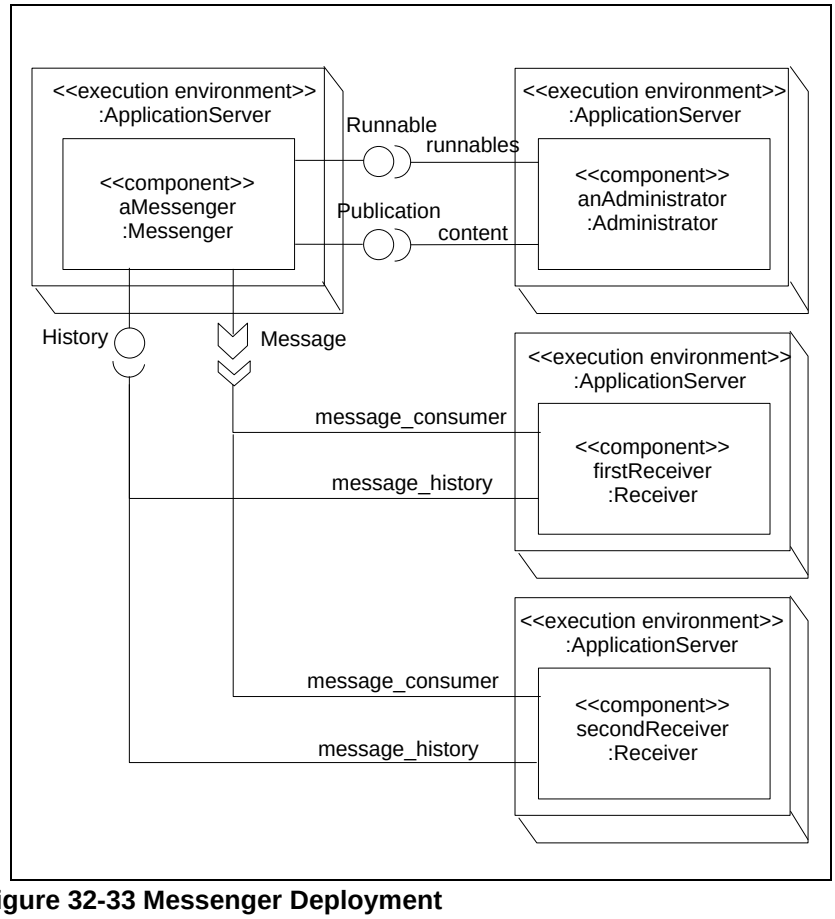

Figure 32-33 Messenger Deployment

Each component type -- the Messenger, Receiver, and Administrator -- consists of a set of dynamic libraries. We have not created any executables. Deployment descriptor files, as described in 32.2.5, define how the component instances are created and connected together and how those component libraries are deployed onto physical nodes.

32.2.7.1 Setting Up Your Environment

Please see 32.2.6.1 for information on setting up the environment variables required to execute a CIAO application.

32.2.7.2 DAnCE Executables

CIAO's Deployment And Configuration Engine (DAnCE), which implements the OMG "Deployment and Configuration of Component-based Distributed Applications" specification (OMG Document ptc/03-07-08), contains a set of executables to dynamically load component libraries, create component instances, and make connections between them. The table summarizes the DAnCE executables that we'll use to deploy the application. The executables are described in more detail in 32.4.

Table 32-10 DAnCE Executables

Name	Description
`dance_node_manager`	A daemon process that launches component servers
`ciao_componentserver`	The component server
`dance_execution_manager`	A process that maps component instances to component servers
`dance_repository_manager`	A process that parses a set of deployment descriptors and sends deployment information to the ExecutionManager

32.2.7.3 Deploying the Messenger with DAnCE

First, we run four dance_node_manager daemon processes. In our example, each Node Manager process executes on the `localhost`. The four processes listen on ports 10000, 20000, 30000, and 40000, respectively. Each Node Manager launches a Node Application process when an application is deployed upon it.

```
$DANCE_ROOT/bin/dance_node_manager \
   -ORBListenEndpoints iiop://localhost:10000 \
   -s "$CIAO_ROOT/bin/ciao_componentserver"

$DANCE_ROOT/bin/dance_node_manager \
   -ORBListenEndpoints iiop://localhost:20000 \
   -s "$CIAO_ROOT/bin/ciao_componentserver"

$DANCE_ROOT/bin/dance_node_manager \
   -ORBListenEndpoints iiop://localhost:30000 \
   -s "$CIAO_ROOT/bin/ciao_componentserver"

$DANCE_ROOT/bin/dance_node_manager \
   -ORBListenEndpoints iiop://localhost:40000 \
   -s "$CIAO_ROOT/bin/ciao_componentserver"
```

Then, we start an Execution Manager that reads the node map file `ApplicationNodeMap.dat` and writes its own object reference to a file called `em.ior`. The Execution Manager executable must be run in the directory containing the application's deployment descriptors.

```
cd $CIAO_ROOT/examples/DevGuideExamples/Messenger/descriptors
$DANCE_ROOT/bin/dance_execution_manager \
   -o em.ior -i ApplicationNodeMap.dat
```

Finally, we start a Repository Manager that connects to the Execution Manager and reads the application's Top-level Package Descriptor and its Component Deployment Plan. The Repository Manager executable must also be run in the directory containing the application's deployment descriptors.

```
cd $CIAO_ROOT/examples/DevGuideExamples/Messenger/descriptors
$DANCE_ROOT/bin/dance_repository_manager \
   -p package.tpd -d Application.cdp -k file://em.ior
```

The Execution Manager deploys a component instance in each of the Node Manager windows.

The Messenger does not begin publishing automatically. One of the Node Manager windows contains the Administrator component instance. That window displays the following menu:

```
What do you want to do to the Messenger(s)?
1.  Start
2.  Stop
3.  Change Publication Period
4.  Change Publication Text
Please enter a selection:
```

Use the menu to start and stop publishing and change attributes of the publication. A more industrial-strength application might launch a GUI of some kind. The Administrator's menu illustrates that a component implementation can contain user interface elements.

32.2.7.4 Debugging

It can be a challenge to debug a component executor implementation. Thorough unit testing uncovers many problems before the component is tested in deployment. However, it may be necessary to debug a component in its component server process.

We launch a component executor in the debugger by changing the Node Manager's launch command for its Node Application. It is easiest to illustrate with an example.

UNIX and UNIX-like Systems

```
$DANCE_ROOT/bin/dance_node_manager
   -ORBListenEndpoints iiop://localhost:10000
   -d 180
   -s "gdb --args $CIAO_ROOT/bin/ciao_componentserver"
```

Windows Systems

```
%DANCE_ROOT%\bin\dance_node_manager.exe
   -ORBListenEndpoints iiop://localhost:10000
   -d 180
   -s "devenv /debugexe %CIAO_ROOT%\bin\ciao_componentserver.exe"
```

The Node Manager's launch command launches the component server process in the debugger. The `-d 180` command-line option delays the launching of the NodeApplication process by 180 seconds, or three minutes. In that three minutes you must set a breakpoint in your component executor to stop program execution. Once a breakpoint is reached you may debug as usual.

32.3 Building CIAO

Before building CIAO, build the ACE and TAO libraries as described in Chapter 2 and Appendix A.

CIAO requires three external libraries: Xerces C++; Boost; and Utility. The DAnCE deployment framework requires Xerces C++. The CIDL compiler requires Boost and Utility. These build instructions describe how to obtain and build those libraries. For more information, see `$CIAO_ROOT/CIAO-INSTALL.html`.

Building CIAO entails several steps:

1. Build ACE and TAO, including the following targets:

```
ACE
ACEXML_Parser
TAO_IDL
```

```
TAO
IFR_Client
IORInterceptor
IORTable
Naming_Service
RTCORBA
RTPortableServer
Valuetype
Security
Utils
```

2. Obtain and build the Xerces C++ library
3. Obtain and build the Boost library
4. Obtain the Utility library
5. Set up the build environment
6. Enable the CIDL compiler in MPC's global features file.
7. Generate build files with MPC
8. Build CIAO's CIDL compiler
9. Build CIAO's libraries and DAnCE executables

32.3.1 Building CIAO with Visual C++

The CIAO libraries and DAnCE executables can be built with either Visual C++ 7.1/8/9. However, the CIDL compiler can only be built with Visual C++ 7.1. This section contain directions for building CIAO and the CIDL compiler with Visual C++ 7.1.

Obtain and Build the Xerces C++ Library

The source code for Xerces C++ can be obtained from `<http://xml.apache.org/xerces-c>`. At publication time, the latest version of Xerces C++ was version 2.6. Download and unzip the Xerces C++ 2.6 source code. The remainder of this section assumes that Xerces C++ is installed in a directory called `C:\xerces-c-src-2_6_0`.

The Xerces C++ site also contains many prebuilt distributions of the Xerces C++ library. If you find a binary distribution that matches your platform and compiler then you can avoid building Xerces C++.

Set the `XERCESCROOT` environment variable to your root Xerces C++ directory, as follows:

```
set XERCESCROOT=C:\xerces-c-src_2_6_0
```

Open the Visual Studio solution file called `xerces-all.sln` in the `%XERCESCROOT%\Projects\Win32\VC7\xerces-all` directory. If Visual Studio asks if you want to convert the projects to the current version of Visual Studio, answer "`yes`". If there is a Visual Studio solution file for Visual Studio .NET (Visual C++ 7.1), use it instead. Build the `XercesLib` target.

The Xerces C++ project does not install its include files and libraries in the directories where CIAO is expecting them. We manually rectify this. First, we copy the Xerces C++ source files to an `include` directory. This copies both header and source files, but that is not a problem.

```
cd %XERCESCROOT%
xcopy /E /I  src include
```

Then, we create directories for the Xerces C++ libraries.

```
mkdir %XERCESCROOT%\lib
mkdir %XERCESCROOT%\bin
```

Finally, we copy the Xerces C++ libraries to the appropriate directories

```
cd %XERCESCROOT%\lib
copy ..\Build\Win32\VC7\Debug\xerces-c_2D.lib
copy xerces-c_2D.lib xerces-cd.lib

cd %XERCESCROOT%\bin
copy ..\Build\Win32\VC7\Debug\xerces-c_2_6D.dll
```

Obtain and Build the Boost library

CIAO's CIDL compiler uses the Boost `regex` and `filesystem` libraries and the `spirit` parser framework. The `spirit` parser framework consists only of header files.

CIAO's Windows build requires Boost version 1.30.2. The Boost 1.30.2 source tree and the latest version of the Boost Jam build system can be downloaded from the Boost web site, <`http://www.boost.org`>.

Install Boost 1.30.2 and the latest Boost Jam in the directories of your choice. For this example, we assume that Boost 1.30.2 is installed in `C:\Boost-1.30.2` and Boost Jam is installed in `C:\Boost-Jam`.

You can edit the Boost Jamfile in `C:\Boost-1.30.2\Jamfile` to limit the build to the `filesystem` and `regex` libraries. For example:

```
# Boost Jamfile
project-root ;
# please order by name to ease maintenance
#subinclude libs/date_time/build ;
subinclude libs/filesystem/build ;
#subinclude libs/python/build ;
subinclude libs/regex/build ;
#subinclude libs/signals/build ;
#subinclude libs/test/build ;
#subinclude libs/thread/build ;
```

Build the Boost 1.30.2 `regex` and `filesystem` libraries with Boost Jam as follows:

```
cd C:\Boost-1.30.2
vsvars32.bat
C:\Boost-Jam\bjam.exe "-sTOOLS=vc7.1"
```

Create a directory called `C:\Boost-1.30.2\lib`. Copy the `regex` and `filesystem` library files to `C:\Boost-1.30.2\lib` and rename them so CIAO's build can find them.

```
mkdir C:\Boost-1.30.2\lib
cd C:\Boost-1.30.2\lib

copy
C:\Boost-1.30.2\libs\regex\build\bin\libboost_regex.lib\vc7.1\debug\runtime-
link-dynamic\libboost_regex_debug.lib
copy
C:\Boost-1.30.2\libs\filesystem\build\bin\libboost_filesystem.lib\vc7.1\debu
g\runtime-link-dynamic\libboost_filesystem.lib

rename libboost_regex_debug.lib boost_regex_debug.lib
rename libboost_filesystem.lib boost_filesystem_debug.lib
```

Obtain the Utility Library

CIAO's CIDL compiler uses the Utility library. Download the Utility 1.2.2 library from the following location:

```
http://www.dre.vanderbilt.edu/cidlc/prerequisites/Utility-1.2.2.tar.bz2
```

There is nothing to build. The remaining instructions assume that the Utility library has been unzipped into a directory called `C:\Utility-1.2.2`.

Set Up the Build Environment

Set `CIAO_ROOT`, `XERCESCROOT`, and `UTILITY_ROOT` environment variables and update your `PATH`. Setting `CIAO_ROOT` is not strictly necessary on Windows, but it makes using CIAO more convenient. Setting `XERCESCROOT` and `UTILITY_ROOT` is necessary.

For example:

```
set CIAO_ROOT=%TAO_ROOT%\CIAO
set XERCESCROOT=C:\xerces-c-src_2_6_0
set UTILITY_ROOT=C:\Utility-1.2.2
set PATH=%PATH%;%XERCESCROOT%\bin;%CIAO_ROOT%\bin
```

The `%XERCESCROOT%\bin` directory contains the Xerces DLLs. The `%CIAO_ROOT%\bin` directory contains the CIAO CIDL compiler.

Update the include and library directories in Visual Studio. Add the Boost root directory to Visual Studio's include directories:

```
C:\Boost-1.30.2
```

Add the Boost `lib` directory that we created to Visual Studio's library directories:

```
C:\Boost-1.30.2\lib
```

Enable CIAO and the CIDL Compiler in MPC's Default Features File

Create or edit the `%ACE_ROOT%\bin\MakeProjectCreator\config\default.features` file and enable CIAO and the CIDL compiler.

```
ciao          = 1
cidl          = 1
```

Generate Build Files with MPC

Generate CIAO's Visual Studio project files with MPC:

```
cd %CIAO_ROOT%
```

```
perl %ACE_ROOT%\bin\mwc.pl -recurse -type vc71
```

This command generates a Visual Studio solution file for each MPC workspace file found in the build tree.

Build CIAO's CIDL Compiler

Build CIAO's CIDL compiler by using the Visual Studio workspace `%CIAO_ROOT%\CIDLC\CIDLC.sln`.

You may use the **Batch Build** command in Visual Studio to build the CIDL compiler's libraries and executables. Alternatively, you may find that it is easier to build the libraries and executables from the command line, as follows:

```
cd %CIAO_ROOT%\CIDLC
devenv CIDLC.sln /build debug
```

Build CIAO's Libraries and DAnCE Executables

Build CIAO's libraries and DAnCE executables by using the Visual Studio workspace `%CIAO_ROOT%\CIAO.sln`.

You may use the **Batch Build** command in Visual Studio to build the libraries and configurations in which you are interested. Alternatively, you may find that it is easier to build just the configurations in which you are interested from the command line. For example:

```
cd %CIAO_ROOT%
devenv CIAO.sln /build debug
```

32.3.2 Building CIAO on UNIX with GNU Make and gcc

CIAO may be built with `gcc` versions 3.3 and later.

Obtain and Build the Xerces C++ Library

The source code for Xerces C++ can be obtained from `<http://xml.apache.org/xerces-c>`. At publication time, the latest version of Xerces C++ is version 2.6. Download and unzip the Xerces C++ 2.6 source code. The remainder of this section assumes that Xerces C++ is installed in a directory called `$HOME/xerces-c-src-2_6_0`.

The Xerces C++ site also contains many prebuilt distributions of the Xerces C++ library. If you find a binary distribution that matches your platform and compiler then you can avoid building Xerces C++.

Set the `XERCESCROOT` environment variable to your root Xerces C++ directory, as follows:

```
export XERCESCROOT=$HOME/xerces-c-src_2_6_0
```

Build the Xerces C++ libraries as follows:

```
cd $XERCESCROOT/src/xercesc
autoconf
./runConfigure -plinux -cgcc -xg++ -minmem -nsocket -tnative -rpthread
make
```

This execution of the Xerces C++ `runConfigure` script command uses the `gcc` compiler, targets the `linux` platform, and builds with `pthreads`. For more information, type enter the following at the command line:

```
./runConfigure -help
```

The Xerces C++ project does not install its include files in the directories where CIAO is expecting them. We manually rectify this by creating a symbolic link to an `include` directory.

```
cd $XERCESCROOT
ln -s src include
```

Obtain and Build the Boost library

CIAO's CIDL compiler uses the Boost `regex` and `filesystem` libraries and the `spirit` parser framework. The `spirit` parser framework consists only of header files.

CIAO's UNIX versions can use the latest version of Boost, which is version 1.32 at publication time. Using a later version of Boost allows more flexibility in the choice of compiler version. The Boost source tree and the latest version of the Boost Jam build system can be downloaded from the Boost web site, <`http://www.boost.org`>.

Install Boost 1.32 and the latest Boost Jam in the directories of your choice. For this example, we assume that Boost 1.32 is installed in `$HOME/boost_1_32_0` and Boost Jam is installed in `$HOME/boost-jam`.

Build the Boost 1.32 libraries with Boost Jam as follows:

```
cd $HOME/boost_1_32_0
$HOME/boost-jam/bjam -sTOOLS=gcc
```

Obtain the Utility Library

CIAO's CIDL compiler uses the Utility library. Download the Utility 1.2.2 library from the following location:

```
http://www.dre.vanderbilt.edu/cidlc/prerequisites/Utility-1.2.2.tar.bz2
```

There is nothing to build. The remaining instructions assume that the Utility library has been unzipped into a directory called `$HOME/Utility-1.2.2`.

Set Up the Build Environment

Set `CIAO_ROOT`, `XERCESCROOT`, `UTILITY_ROOT`, `BOOST_ROOT`, `BOOST_INCLUDE`, and `BOOST_LIB` environment variables and update your `PATH`. Setting `CIAO_ROOT` is not strictly necessary on Windows, but it makes using CIAO more convenient. Setting `XERCESCROOT`, `UTILITY_ROOT`, and the Boost environment variables is necessary.

For example:

```
export CIAO_ROOT=$TAO_ROOT/CIAO
export XERCESCROOT=$HOME/xerces-c-src_2_6_0
export UTILITY_ROOT=$HOME/Utility-1.2.2
export BOOST_ROOT=$HOME/boost_1_32_0
export BOOST_INCLUDE=$BOOST_ROOT
export BOOST_LIB=$BOOST_ROOT/libs
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$XERCESCROOT/lib
```

Enable CIAO and the CIDL Compiler in MPC's Default Features File

Create or edit the `$ACE_ROOT/bin/MakeProjectCreator/config/default.features` file and enable CIAO and the CIDL compiler.

```
ciao          = 1
cidl          = 1
```

Generate Build Files with MPC

Generate CIAO's GNU Makefiles with MPC:

```
cd $CIAO_ROOT
$ACE_ROOT/bin/mwc.pl -recurse -type gnuace
```

Build CIAO's CIDL Compiler

Build CIAO's CIDL compiler by executing `make` in the `$CIAO_ROOT/CIDLC` directory.

```
cd $CIAO_ROOT/CIDLC
make ciao=1 cidl=1
```

Build CIAO's Libraries and DAnCE Executables

Build CIAO's libraries and DAnCE executables by executing `make` in the `$CIAO_ROOT/ccm`, `$CIAO_ROOT/ciao`, and `$CIAO_ROOT/ciao` directories.

```
cd $CIAO_ROOT/ccm
make ciao=1 cidl=1

cd $CIAO_ROOT/ciao
make ciao=1 cidl=1

cd $CIAO_ROOT/DAnCE
make ciao=1 cidl=1
```

32.4 DAnCE Executable Reference

CIAO's Deployment And Configuration Engine (DAnCE), which implements the OMG "Deployment and Configuration of Component-based Distributed Applications" specification (OMG Document ptc/03-07-08), contains a set of executables to dynamically load component libraries, create component instances, and make connections between them.

The DAnCE executables are described in the following subsections.

32.4.1 Overview

The DAnCE executables are as follows:

UNIX and UNIX-like Systems

Table 32-11 DAnCE Executables

Name	Path
dance_node_manager	$DANCE_ROOT/bin/dance_node_manager
ciao_componentserver	$CIAO_ROOT/bin/ciao_componentserver
dance_execution_manager	$DANCE_ROOT/bin/dance_execution_manager
dance_repository_manager	$DANCE_ROOT/bin/dance_repository_manager

Windows Systems

Table 32-12 DAnCE Executables

Name	Path
dance_node_manager	%DANCE_ROOT%\bin\dance_node_manager
ciao_componentserver	%CIAO_ROOT%\bin\ciao_componentserver
dance_execution_manager	%DANCE_ROOT%\bin\dance_execution_manager
dance_repository_manager	%DANCE_ROOT%\bin\dance_repository_manager

32.4.2 Node Manager and Node Application

The `dance_node_manager` is a daemon process that launches `ciao_componentserver` processes as directed by the `dance_execution_manager`. Each object reference in the Messenger's `ApplicationNodeMap.dat` configuration file refers to a `NodeManager` object in the `NodeManager` daemon process. Recall that the Messenger's `ApplicationNodeMap.dat` file is as follows:

```
Administrator_Node   corbaloc:iiop:localhost:10000/NodeManager
First_Receiver_Node  corbaloc:iiop:localhost:20000/NodeManager
Second_Receiver_Node corbaloc:iiop:localhost:30000/NodeManager
Messenger_Node       corbaloc:iiop:localhost:40000/NodeManager
```

The node map file expects to find four different `NodeManager` objects on the `localhost`, each in an ORB listening on a different port. Presumably, each

NodeManager object lives in a different process. Each NodeManager launches a ciao_compontnserver process as a container for the component instance or instances mapped to it. For example:

```
$DANCE_ROOT/bin/dance_node_manager \
   -ORBListenEndpoints iiop://localhost:10000 \
   -s "$CIAO_ROOT/bin/ciao_componentserver"
```

The dance_node_manager executable recognizes the following command-line options

Table 32-13 dance_node_manager Command-Line Options

Option	Description	Default
-s *server_executable*	Componentserver executable to be launched.	REQUIRED
-o *ior_file*	Export the NodeManager IOR to a file. Supersedes registration with Naming Service.	off
-d *spawn_delay*	Delay spawning of the NodeApplication by *spawn_delay* seconds. This can be helpful for debugging.	0
-n	Register the NodeManager with the Naming Service in the root naming context with the name retrieved by calling ACE_OS::hostname(). Superseded by export of IOR file.	off
-?	Display usage information.	n/a

32.4.3 Execution Manager

The dance_execution_manager reads the node map file and maps each component instance to the dance_node_manager responsible for it. For example:

```
$DANCE_ROOT/bin/dance_execution_Manager \
   -o em.ior -i ApplicationNodeMap.dat
```

The `dance_execution_manager` executable recognizes the following command-line options

Table 32-14 Execution Manager Command-Line Options

Option	Description	Default
`-i` *`node_map_file`*	Path of the node map file.	`deployment.dat`
`-o` *`ior_file`*	Export the `ExecutionManager` IOR to a file. Supersedes registration with Naming Service.	off
`-n`	Register the `ExecutionManager` with the Naming Service in the root naming context with the name `ExecutionManager`. Superseded by export of IOR file.	off
`-?`	Display usage information.	n/a

32.4.4 Repository Manager

The `RepositoryManager` parses the XML Deployment and Configuration files and passes the relevant deployment information to the `dance_execution_manager`. For example:

```
$DANCE_ROOT/bin/dance_repository_manager \
   -p package.tpd -d Application.cdp -k file://em.ior
```

The `RepositoryManager` executable recognizes the following command-line options

Table 32-15 Repository Manager Command-Line Options

Option	Description	Default
`-k` *`execution_manager_ior`*	The Execution Manager's IOR.	`file://exec_mgr.ior`
`-p` *`package_url`*	The Top-level Package Descriptor's URL	REQUIRED
`-d` *`plan_url`*	The Component Deployment Descriptor's URL	REQUIRED

32.5 CIDL Compiler Reference

A CIDL `composition` embedded in a CIDL file describes a component implementation. CIAO includes a CIDL compiler, `cidlc`, that generates local

IDL interfaces for component homes and executors and C++ classes for servants and default executor implementations.

Note *The generated C++ code is only usable by CIAO. The C++ output from CIDL compilers cannot be interchanged among CORBA implementations. However, the code generated by CIAO's CIDL compiler is platform-independent, making it possible to use CIAO in cross-compilation environments.*

CIAO's CIDL compiler maps CIDL files to equivalent IDL and C++ according to the CORBA Component Model specification.

32.5.1 CIDL Executables

UNIX and UNIX-like Systems

The CIDL compiler executable is `$CIAO_ROOT/bin/cidlc`.

Windows Systems

The CIDL compiler executable is `%CIAO_ROOT%\bin\cidlc.exe`.

General Usage

The general usage of the CIAO CIDL compiler is as follows:

```
cidlc <options> -- CIDL-file
```

The CIDL file name must be listed after the "--", which is listed after the options. For example:

```
cidlc -I . -I $CIAO_ROOT/ccm -I $CIAO_ROOT/ciao -I $TAO_ROOT \
      -I $TAO_ROOT/tao -I $TAO_ROOT/orbsvcs -- Messenger.cidl
```

32.5.2 Output Files Generated

The CIDL compiler generates three files for each CIDL file. One of these files is an IDL2 file containing the component executor's local IDL2 interfaces. The component developer compiles that file with the TAO IDL compiler. The remaining two files are C++ files containing the component servant's class definition and implementation. The generation of these files ensures that the

generated code is portable and optimized for a wide variety of C++ compilers. The diagram illustrates the generated files.

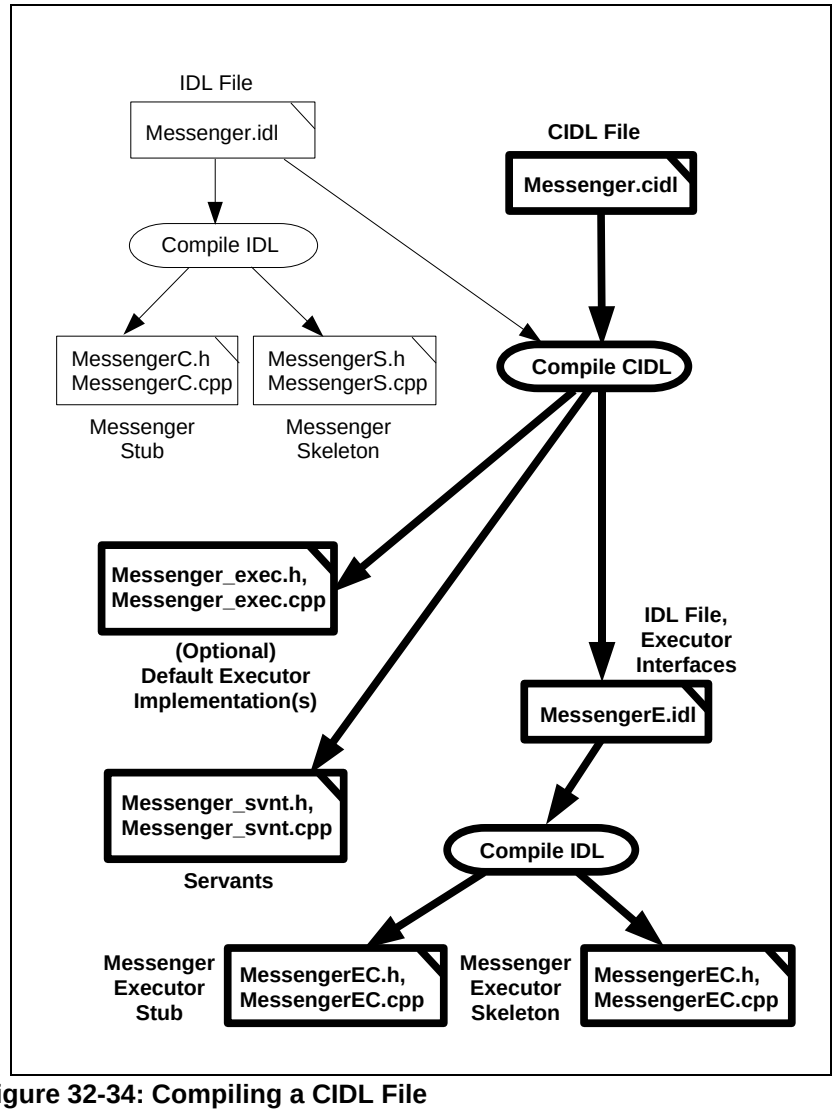

Figure 32-34: Compiling a CIDL File

For a CIDL file named `Messenger.cidl`, running the command

```
cidlc -I . -I $CIAO_ROOT/ccm -I $CIAO_ROOT/ciao -I $TAO_ROOT \
      -I $TAO_ROOT/tao -I $TAO_ROOT/orbsvcs -- Messenger.cidl
```

generates the following files (we show how to customize these names later):

Table 32-16 IDL and C++ Files Generated

File Name	Description
`MessengerE.idl`	IDL2 for the component executor, to be run through the `tao_idl` compiler.
`Messenger_svnt.h`	Component and facet servant class definition.
`Messenger_svnt.cpp`	Component and facet servant class implementations.

32.5.3 Using CIDL Compiler Options

We discuss CIDL compiler command line options in 32.5.4 through 32.5.9. To see a complete list of the CIDL compiler's options, enter the following:

```
cidlc --help
```

In addition to the CIDL compiler options listed by the `--help` argument, the CIDL compiler also recognizes the `-I` preprocessor argument for specifying an element of the include path.

32.5.4 Preprocessing Options

The CIDL compiler does not run the full C preprocessor. It recognizes only the `-I include_path` preprocessor command-line option and the `#include` preprocessor directive. All other preprocessor directives are ignored.

Each CIDL file must be compiled with, at a minimum, the following include path:

```
-I $CIAO_ROOT/ccm -I $CIAO_ROOT/ciao -I $TAO_ROOT -I $TAO_ROOT/tao -I
$TAO_ROOT/orbsvcs
```

Each CIDL file indirectly includes a standard IDL file called `Components.idl`, which in turn includes several other IDL files.

The table provides details of the preprocessing options.

Table 32-17 Preprocessing Options

Option	Description	Default
`---preprocess-only`	Run the preprocessor on the IDL file, but do not generate any IDL or C++ code.	generate IDL and C++ code

Table 32-17 Preprocessing Options

Option	Description	Default
`-I` *`include-path`*	Add `include_path` to the list of paths searched for include files.	none

32.5.5 General Options

The CIDL compiler has an option that allows you to turn on a verbose mode that displays detailed information about the CIDL file compilation steps. There are also two options for displaying usage information. The options are summarized in the table.

Table 32-18 General CIDLC Options

Option	Description	Default
`--trace-semantic-actions`	Turn on verbose mode.	off
`--help`	Output usage information to `stderr`.	n/a
`--help-html`	Output usage information in HTML format to `stderr`.	n/a

32.5.6 Servant File Options

The CIDL compiler generates a complete servant class implementation for each component and each facet. The component developer has some control over the servant's usage of event type factories and the names of the generated files.

The table summarizes the servant-related CIDL compiler options.

Table 32-19 Servant File Options

Option	Description	Default
`--suppress-register-factory`	Suppress automatic registration of a value type factory for each event type. By default, a value type factory is automatically registered for each event type. If factory registration is suppressed, then the developer must manually register a value type factory for each event type.	off, meaning a value type factory is automatically registered for each event type
`--svnt-hdr-file-suffix` *`suffix`*	Use this suffix instead of the default to construct the name of the servant's header file.	`_svnt.h`

Table 32-19 Servant File Options

Option	Description	Default
`--svnt-hdr-file-regex` *regex*	Use this regular expression to construct the name of the servant's header file	n/a
`--svnt-src-file-suffix` *suffix*	Use this suffix instead of the default to construct the name of the servant's source file.	`_svnt.cpp`
`--svnt-src-file-regex` *regex*	Use this regular expression to construct the name of the servant's source file	n/a
`--svnt-export-macro` *macro*	Replace the servant's default export macro with this export macro	see below
`--svnt-export-include` *file*	Replace the servant's default export include file with this file	see below

The CIDL compiler assumes that the servant is part of a dynamic library. On Windows platforms, classes exported from dynamic libraries must define an export macro. On UNIX-like platforms, the export macros define to nothing. However, future versions of the `gcc` compiler support the C++ `export` keyword, which may reduce code size by reducing the number of exported symbols. In either case, the export macros enable cross-platform development.

The CIDL compiler assumes that a component servant's export macro is called `<COMPONENT>_SVNT_Export` and that the macro is defined in a header file called `<Component>_svnt_export.h`. For example, the Messenger component's servant export macro is assumed to be

```
MESSENGER_SVNT_Export
```

and it is assumed to be defined in a C++ header file called

```
Messenger_svnt_export.h
```

If that is not the case, then use the `--svnt-export-macro` command-line argument to indicate the correct name of the export macro and the `--svnt-export-include` command-line argument to indicate the correct name of the export header file.

See 32.2.6, "Building the Messenger Application," for more information on component export macros.

32.5.7 Local Executor File Options

The CIDL compiler generates an IDL file containing the component implementation's local executor interfaces. The component developer implements the component and its facets by implementing these local executor interfaces.

The table summarizes the executor-related CIDL compiler options.

Table 32-20 Local Executor File Options

Option	Description	Default
`--lem-file-suffix` *`suffix`*	Suffix for the generated executor IDL file.	`E`
`--lem-file-regex` *`regex`*	Regular expression to use when constructing the name of the local executor IDL file.	n/a
`--lem-force-all`	Force generation of local executor mapping for all IDL types, whether used by the composition or not. By default, the CIDL compiler generates local executor interfaces only for those components used by the composition.	off

32.5.8 Starter Executor Implementation File Options

The CIDL compiler can generate a default executor implementation for each component and facet. These default executor implementation files contain empty C++ member function definitions that you fill in with your implementation code. This can be a great time saver.

Note *Running the CIDL compiler with the starter implementation options overwrites any existing implementation files of the same names. Any modifications will be lost unless you rename the starter implementation files after they are generated (recommended).*

The table summarizes the implementation-related CIDL compiler options.

Table 32-21 Executor Implementation File Options

Option	Description	Default
`--gen-exec-impl`	Generate a default executor implementation class for each component and facet.	off

Table 32-21 Executor Implementation File Options

Option	Description	Default
`--exec-hdr-file-suffix` *suffix*	Use this suffix instead of the default to construct the name of the default executor implementation's header file.	`_exec.h`
`--exec-hdr-file-regex` *regex*	Use this regular expression to construct the name of the default executor implementation's header file.	n/a
`--exec-src-file-suffix` *suffix*	Use this suffix instead of the default to construct the name of the default executor implementation's source file.	`_exec.cpp`
`--exec-src-file-regex` *regex*	Use this regular expression to construct the name of the default executor implementation's source file.	n/a
`--exec-export-macro` *macro*	Replace the default executor implementation's default export macro with this export macro.	see below
`--exec-export-include` *file*	Replace the default executor implementation's default export include file this file.	see below

You are strongly advised to rename the generated default executor implementation files before modifying them. Otherwise, the CIDL compiler will likely overwrite your changes. For example, rename `Messenger_exec.h` and `Messenger_exec.cpp` to `Messenger_exec_i.h` and `Messenger_exec_i.cpp`.

The CIDL compiler assumes that the executor implementation is part of a dynamic library. On Windows platforms, classes exported from a dynamic library must define an export macro.

The CIDL compiler assumes that a component executor's export macro is called `<COMPONENT>_EXEC_Export` and that the macro is defined in a header file called `<Component>_exec_export.h`. For example, the Messenger component's executor export macro is assumed to be

```
MESSENGER_EXEC_Export
```

and it is assumed to be defined in a C++ header file called

```
Messenger_exec_export.h
```

If that is not the case, then use the `--exec-export-macro` command-line argument to indicate the correct name of the export macro and the `--exec-export-include` command-line argument to indicate the correct name of the export header file.

32.5.9 Descriptor File Options

The CIDL compiler generates a CORBA Component Descriptor for each component. However, the generated descriptor file is not usable to deploy a CCM application using CIAO's DAnCE facility. The generated descriptor file is formatted in accordance with the deprecated "Packaging and Deployment" chapter of the OMG CORBA Component Model specification (OMG Document formal/02-06-65) rather than the updated OMG "Deployment and Configuration of Component-based Distributed Applications" specification (OMG Document ptc/03-07-08). Thus, we ignore the generated CORBA Component Descriptor files in our deployment.

The table summarizes the descriptor-related CIDL compiler options:

Table 32-22 Descriptor File Options

Option	Description	Default
`--desc-file-suffix` *suffix*	Use this suffix instead of the default to construct the name of the descriptor file.	`.ccd`
`--desc-file-regex` *regex*	Use this regular expression to construct the name of the descriptor file	n/a

32.6 IDL3-to-IDL2 Compiler Reference

CIAO includes an IDL3-to-IDL2 compiler that generates IDL2-compatible interfaces for ORB implementations that do not recognize IDL3 keywords such as "`component`" and "`provides`". This enables a client developed with a non-CCM-aware ORB to communicate with a CCM component. For example, a Java client built with an ORB such as JacORB can use CIAO's IDL3-to-IDL2 output files as its interface to the Messenger component. Simply compile the Messenger's IDL3 files with CIAO's IDL3-to-IDL2 compiler, and then compile the IDL2 output with JacORB's IDL compiler.

Note *The generated IDL2 code is usable by any ORB.*

CIAO's IDL3-to-IDL2 compiler maps IDL3 files to equivalent IDL2 according to the Equivalent IDL sections of the CORBA Component Model specification.

For an example of using the IDL3-to-IDL2 compiler, please see the `Administrator_Client_IDL2.mpc` project in the `$CIAO_ROOT/examples/DevGuideExamples/Messenger` directory.

32.6.1 IDL3-to-IDL2 Source Code

The source code for the IDL3-to-IDL2 compiler is in the `$CIAO_ROOT/tools/IDL3_to_IDL2` directory. Build the code in that directory to create the `tao_idl3_to_idl2` executable.

32.6.2 IDL3-to-IDL2 Executable

UNIX and UNIX-like Systems

The IDL3-to-IDL2 compiler executable is `$CIAO_ROOT/bin/tao_idl3_to_idl2.`

Windows Systems

The IDL3-to-IDL2 compiler executable is `%CIAO_ROOT%\bin\tao_idl3_to_idl2.exe.`

General Usage

The general usage of the CIAO IDL3-to-IDL2 compiler is as follows:

```
tao_idl3_to_idl2 -I $CIAO_ROOT -I $CIAO_ROOT/ciao -I $CIAO_ROOT/ciao -I
$TAO_ROOT \
      -I $TAO_ROOT/tao -I $TAO_ROOT/orbsvcs \
      <options> <idl3 files>
```

For example:

```
tao_idl3_to_idl2 -I $CIAO_ROOT -I $CIAO_ROOT/ciao -I $CIAO_ROOT/ciao -I
$TAO_ROOT \
      -I $TAO_ROOT/tao -I $TAO_ROOT/orbsvcs \
      -I . Messenger.idl
```

The lengthy include path is necessary to enable the IDL3-to-IDL2 compiler to find CIAO's CCM-related IDL files.

32.6.3 Output Files Generated

The IDL3-to-IDL2 compiler generates one IDL2 output file for each input file. A developer typically compiles that output file with another ORB's IDL compiler. The diagram illustrates the generated files.

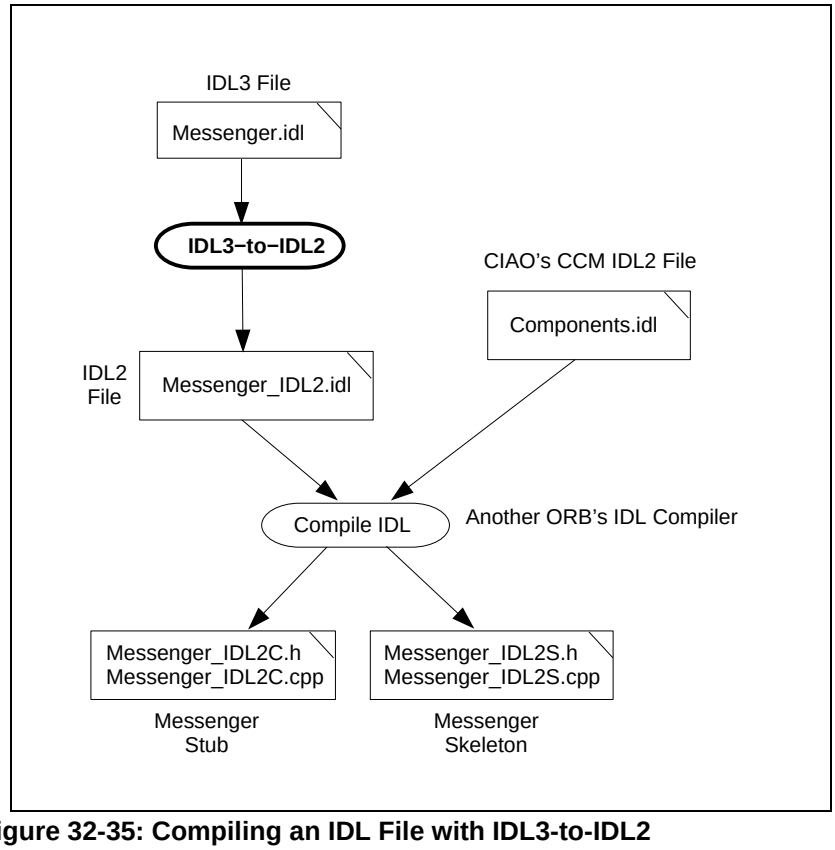

Figure 32-35: Compiling an IDL File with IDL3-to-IDL2

For an IDL3 file named `Messenger.idl`, running the command

```
tao_idl3_to_idl2 -I $CIAO_ROOT -I $CIAO_ROOT/ciao -I $CIAO_ROOT/ciao -I
$TAO_ROOT \
      -I $TAO_ROOT/tao -I $TAO_ROOT/orbsvcs \
      -I . Messenger.idl
```

generates the following file:

Table 32-23 IDL and C++ Files Generated

File Name	Description
`Messenger_IDL2.idl`	Equivalent IDL2 for Messenger's IDL3 file, to be run through another ORB's IDL compiler.

The generated `Messenger_IDL2.idl` file includes CIAO's `Components.idl` file, which contains IDL2 CCM declarations. When compiling the `Messenger_IDL2.idl` file with a non-CCM-aware ORB's IDL compiler, the `Components.idl` file and the files it includes must be in the include path of that ORB's IDL compiler.

In our example, we compile the IDL3 file `Messenger.idl`, which follows:

```
// file Messenger.idl
#include <Components.idl>
#include <Runnable.idl>
#include <Publication.idl>
#include <Message.idl>
#include <History.idl>

component Messenger {
  attribute string subject;

  provides Runnable control;
  provides Publication content;

  publishes Message message_publisher;
  provides History message_history;
};

home MessengerHome manages Messenger {};
```

Note that you must also compile the included IDL3 files `Runnable.idl`, `Publication.idl`, `Message.idl`, and `History.idl` with the IDL3-to-IDL2 compiler.

The compiler generates the IDL2 file `Messenger_IDL2.idl`:

```
// file Messegner_IDL2.idl
#include "Components.idl"
#include "Runnable_IDL2.idl"
#include "Publication_IDL2.idl"
#include "Message_IDL2.idl"
#include "History_IDL2.idl"
```

```
interface Messenger : Components::CCMObject
{

  attribute string subject;
  Runnable provide_control ();
  Publication provide_content ();
  History provide_message_history ();

  Components::Cookie subscribe_message_publisher (
    in MessageConsumer consumer)
    raises (Components::ExceededConnectionLimit);

  MessageConsumer unsubscribe_message_publisher (in Components::Cookie ck)
    raises (Components::InvalidConnection);
};

interface MessengerHomeExplicit : Components::CCMHome
{
};

interface MessengerHomeImplicit : Components::KeylessCCMHome
{
  Messenger create ()
    raises (Components::CreateFailure);
};

interface MessengerHome : MessengerHomeExplicit, MessengerHomeImplicit
{
};
```

32.6.4 IDL3-to-IDL2 Compiler Options

We discuss IDL3-to-IDL2 compiler command line options in 32.6.5 through 32.6.6. To see a complete list of the IDL3-to-IDL2 compiler's options, enter the following:

```
tao_idl3_to_idl2 -u
```

32.6.5 Preprocessing Options

The IDL3-to-IDL2 compiler uses the same preprocessor as the `tao_idl` compiler. For more information on the preprocessor options and directives, please see 5.5. The most commonly used of these options is the `-I` option, which specifies a directory for the include path. For example:

```
tao_idl3_to_idl2 -I $CIAO_ROOT -I $CIAO_ROOT/ciao -I $CIAO_ROOT/ciao -I
$TAO_ROOT \
      -I $TAO_ROOT/tao -I $TAO_ROOT/orbsvcs \
      -I . Messenger.idl
```

32.6.6 General Options

The IDL3-to-IDL2 compiler's other remaining options are summarized below. Each option's function is identical to its matching option of the IDL compiler.

Table 32-24 General IDL3-to-IDL2 Options

Option	Description	Default
`-o` *`output-directory`*	Subdirectory in which to place the generated stub and skeleton files.	Current directory
`-t` *`dir`*	Directory used by the IDL compiler for temporary files.	In UNIX, uses the value of the `TMPDIR` environment variable, if set, or `/tmp` by default. In Windows, uses the value of the `TMP` environment variable, if set, or the `TEMP` environment variable, if set, or the `WINNT` directory (on NT).
`-v`	Verbose flag. IDL compiler will print progress messages after completing major phases.	No progress messages displayed.
`-d`	Print the Abstract Syntax Tree (AST) to `stdout`.	AST is not displayed.
`-w`	Suppress warnings.	All warnings displayed.
`-V`	Print version information for front end and back end.	No version information displayed.
`-Cw`	Output a warning if two identifiers in the same scope differ in spelling only by case.	Error output is default.
`-Ce`	Output an error if two indentifiers in the same scope differ in spelling only by case.	Error output is default.
`-g` *`gperf-path`*	Specify a path for the `gperf` program	`$ACE_ROOT/bin/gperf`

32.7 Future Topics

Several CCM and CIAO-related topics are beyond the scope of this chapter. They include the following:

- Component navigation
- Keyed component homes
- Home finders
- Life cycle categories `service`, `process`, and `entity`
- The IDL3 `supports` keyword

In addition, there are several capabilities that are expected to be addressed in future versions of CIAO. These include the following:

- Static application deployment
- Deployment of real-time applications
- Container-managed persistent using the Persistent State Service (PSS) and Persistent State Definition Language (PSDL) (OMG Document formal/02-09-06)
- Integration with Enterprise Java Beans
- Using the Real-Time Event Service or OMG Notification Service as the event delivery infrastructure
- Quality-of-Service