CORBA Component Model Tutorial

OMG CCM Implementers Group,
MARS PTC & Telecom DTC
OMG Meeting, Orlando, USA,
June 25th, 2002

OMG TC Document ccm/2002-06-01

Tutorial Objectives

- A guided tour of the CORBA Component Model
  - How to design, implement, package, deploy, execute, and use CORBA components
  - Putting the CCM to work

- Illustrated with a concrete example
  - Well-known Dining Philosophers
  - Demonstrated on various OS, ORB, CCM platforms, and programming languages (C++, Java, OMG IDLscript)

Agenda

- What is the CORBA Component Model?
- Defining CORBA components
- Programming CORBA component clients
- Implementing CORBA components
- Putting CORBA containers to work
- Packaging CORBA components
- Deploying CORBA component applications
- Summary

Tutorial Team

 Speakers
- Philippe Merle - LIFL/INRIA - Philippe.Merle@lifl.fr
- Diego Sevilla Ruiz - Universidad de Murcia - dsevilla@ditec.um.es
- Harald Böhme - Humboldt University - boehme@informatik.hu-berlin.de
- Sylvain Leblanc - LIFL - Sylvain.Leblanc@lifl.fr
- Mathieu Vadet - THALES/LIFL - Mathieu.Vadet@lifl.fr
- Tom Ritter - Fraunhofer Fokus - ritter@fokus.gmd.de
- J. Scott Evans - CPI - evans@dpi.com

 Contributors
- Raphaël Marvie - LIFL – Raphael.Marvie@lifl.fr
- Frank Pilhofer - Alcatel/FPX - fp@fpx.de

With support from the European IST COACH project
What is the CORBA Component Model?

- From CORBA 2.x to the CCM
- Comparison with EJB, COM, and .NET
- CCM Technologies
- Typical Use Case

Why Software Components?

- Time to market
  - Improved application productivity
  - Reduced complexity
  - Reuse of existing code
- Programming by assembly (manufacturing) rather than development (engineering)
  - Reduced skills requirements
  - Focus expertise on domain problems
  - Improving software quality
- Key benefit with client side & server side development

From CORBA 2 . . .

- A distributed object-oriented model
  - Heterogeneity: OMG Interface Definition Language (OMG IDL)
  - Portability: Standardized language mappings
  - Interoperability: GIOP / IIOP
  - Various invocation models: SII, DII, and AMI
  - Middleware: ORB, POA, etc.
    - minimum, real-time, and fault-tolerance profiles
- No standard packaging and deployment facilities !!!
- Explicit programming of non functional properties !!!
  - lifecycle, (de)activation, naming, trading, notification, persistence, transactions, security, real-time, fault-tolerance, ...
- No vision of software architecture

. . . to the CORBA Component Model

- A distributed component-oriented model
  - An architecture for defining components and their interactions
    - From client-side (GUI) to server-side (business) components
  - A packaging technology for deploying binary multi-lingual executables
  - A container framework for injecting lifecycle, (de)activation, security, transactions, persistence, and events
  - Interoperability with Enterprise Java Beans (EJB)
- The Industry’s First Multi-Language Component Standard
  - Multi-languages, multi-OSs, multi-ORBs, multi-vendors, etc.
  - Versus the Java-centric EJB component model
  - Versus the MS-centric .NET component model
CCM Compared to EJB, COM and .NET

- Like SUN Microsystems’s Enterprise Java Beans (EJB)
  - CORBA components created and managed by homes
  - Run in containers managing system services transparently
  - Hosted by application component servers

- Like Microsoft’s Component Object Model (COM)
  - Have several input and output interfaces
  - Both synchronous operations and asynchronous events
  - Navigation and introspection capabilities

- Like Microsoft’s .NET Framework
  - Could be written in different programming languages
  - Could be packaged in order to be distributed

But with CCM

- A CCM application is “really” distributed
  - Could be deployed and run on several distributed nodes simultaneously

- A CORBA component could be segmented into several classes

What is the CCM Specification?

- Abstract Component Model
  - Extensions to IDL and the object model

- Component Implementation Framework
  - Component Implementation Definition Language (CIDL)

- Component Container Programming Model
  - Component implementer and client view
  - Integration with Security, Persistence, Transactions, and Events

- Packaging and deployment facilities

- Interoperability with EJB 1.1

- Component Metadata & Metamodel
  - Interface Repository and MOF extensions
**Relations between OMG Definition Languages**

- **OMG IDL 2.x**
  - Object-oriented collaboration
  - i.e. data types, interfaces, and value types

- **OMG IDL 3.0**
  - Component-oriented collaboration
  - i.e. component types, homes, and event types

- **OMG PSDL**
  - Persistent state definition
  - i.e. [abstract] storage types and homes

- **OMG CIDL**
  - Component implementation description
  - i.e. compositions and segments

**CCM User Roles**

- Component designers
- Component clients
- Composition designers
  - (~ component implementation designers)
- Component implementers
- Component packagers
- Component deployers
- Component end-users

**Component Designers**

- Define component and home types via OMG IDL 3.0 extensions
- Output
  - OMG IDL 3.0 files
  - Client-side OMG IDL mapping
  - Client-side stubs
  - Interface Repository entries

**Component Clients**

- CCM designed for CORBA-2 compliance
  - Component clients could run on “legacy” ORBs
- View components and homes via the client-side OMG IDL mapping
- Use client-side stubs
- Could navigate and introspect components via the generic CCMObject and CCMHome interfaces
Composition Designers

- Specify platform and language independent features required to facilitate code generation
  - Component Implementation Definition Language (CIDL)
  - Persistence State Definition Language (PSDL)

- Output
  - Local server-side OMG IDL mapping
  - Component skeletons
  - Component metadata as XML descriptors

Component Implementers

- Implement business logic operations
  - Defined by local server-side OMG IDL interfaces
  - Could inherit from generated CIDL skeletons
  - Could implement local container callback interfaces
  - Could invoke local container interfaces

- Output
  - Component binaries
  - XML component descriptors enriched

Component Packagers

- Produce component packages containing
  - Component binaries
  - Software & component XML descriptors
  - Default property XML descriptors
  - Probably done using an interactive visual tool

- Output - component archive file (zip file)

- If “no further assembly required”, skip to deployment
Component Assemblers

- Produce assembly packages containing
  - Customized component packages
  - Assembly XML descriptors
    - Component instances and interconnections
    - Logical distribution partitioning
  - Probably done using an interactive visual tool

- Output - component assembly archive file

- Process may be iterated further

Component Deployers

- Deployment/installation tool takes deployer input + component and assembly archives
- Attach virtual component locations to physical nodes
- Start the deployment process
  - Installs components and assemblies to particular nodes on the network
- Output - instantiated and configured components and assemblies now available
  - CCM applications deployed in CCM containers

The CCM Big Picture

Next Tutorial Steps

- Defining CORBA component types
  - Abstract Component Model and OMG IDL 3.0 extensions
- Programming CORBA component clients
  - Client-side OMG IDL mapping
- Implementing CORBA components
  - Component Implementation Framework (CIF)
  - Local server-side OMG IDL mapping
  - Component Implementation Definition Language (CIDL)
- Putting CORBA containers to work
- Packaging CORBA components
  - Associated XML DTDs
- Deploying CORBA component applications
  - Component deployment objects and "basic" process
Defining CORBA Components

- The Abstract Component Model
- OMG IDL 3.0 Extensions
- The Dining Philosophers Example

The Abstract Component Model

- Allows component designers to capture how CORBA components are viewed by other components and clients
  - What a component offers to other components
  - What a component requires from other components
  - What collaboration modes are used between components
    - Synchronous via operation invocation
    - Asynchronous via event notification
  - Which component properties are configurable
  - What the business life cycle operations are (i.e. home)

-Expressed via OMG IDL 3.0 extensions
  - Syntactic construction for well known design patterns
  - Mapped to OMG IDL interfaces for clients and implementers

What is a CORBA Component?

- Component is a new CORBA meta-type
  - Extension of Object (with some constraints)
  - Has an interface, and an object reference
  - Also, a stylized use of CORBA interfaces/objects

- Provides component features (also named ports)
- Could inherit from a single component type
- Could supports multiple interfaces
- Each component instance is created and managed by a unique component home

Component Features

- Attributes = configurable properties
- Facets = offered operation interfaces
- Receptacles = required operation interfaces
- Event sources = produced events
- Event sinks = consumed events
- Navigation and introspection supported
A CORBA Component

Component interface

Facets

Event sinks

Receptacles

Event sources

My Business Component

Component Attributes

- Named configurable properties
  - Vital key for successful re-usability
  - Intended for component configuration
    - e.g., optional behaviors, modality, resource hints, etc.
  - Could raise exceptions
  - Exposed through accessors and mutators

- Could be configured
  - By visual property sheet mechanisms in assembly or deployment environments
  - By homes or during implementation initialization
  - Potentially readonly thereafter

Component Facets

- Distinct named interfaces that provide the component's application functionality to clients
- Each facet embodies a view of the component, corresponds to a role in which a client may act relatively to the component
- A facet represents the component itself, not a separate thing contained by the component
- Facets have independent object references
Component Receptacles

- Distinct named connection points for potential connectivity
  - Ability to specialize by delegation, compose functions
  - The bottom of the Lego, if you will
- Store a simple reference or multiple references
  - But not intended as a relationship service
- Configuration
  - Statically during initialization stage or assembly stage
  - Dynamically managed at runtime to offer interactions with clients or other components (e.g. callback)

Component Events

- Simple publish / subscribe event model
  - “push” mode only
  - Sources (2 kinds) and sinks
- Events are value types
  - Defined with the new `eventtype` meta-type
  - `valuetype` specialization for component events

Component Event Sources

- Named connection points for event production
  - Push a specified `eventtype`
- Two kinds: Publisher & Emitter
  - publishes = multiple client subscribers
  - emits = only one client connected
- Client subscribes or connects to directly component event source
- Container mediates access to `CosNotification` channels
  - scalability, quality of service, transactional, etc.

Component Event Sinks

- Named connection points into which events of a specific type may be pushed
- Subscription to event sources
  - Potentially multiple (n to 1)
- No distinction between emitter and publisher
  - Both push in event sinks
What is a CORBA Component Home?

- Manages a unique component type
  - More than one home type can manage the same component type
  - But a component instance is managed by a single home instance
- Home is a new CORBA meta-type
  - Home definition is distinct from component one
  - Has an interface, and an object reference
- Could inherit from a single home type
- Could supports multiple interfaces
- Is instantiated at deployment time

A CORBA Component Home

Component Home Features

- Allows life cycle characteristics or key type to vary/evolve without changing component definition
- Optional use of primarykey for business component identity and persistency primary key
- Standard factory and finder business logic operations
- Extensible with arbitrary user-defined business logic operations

Primary Keys

- Values exposed to clients to create, find, and destroy component instances
  - Uniquely identifies a component instance within a home
  - Assigned at creation time, or in pre-existing database
  - Must be a value type derived from Components::PrimaryKeyBase (empty, abstract)
- Association between a primary key and a component is defined and maintained by its home
  - Different home types may define different key types (or no key) for the same component type
  - Primary key is not necessarily a part of the component’s state
Other OMG IDL 3.0 Extensions

- The new `import` keyword
  - Importation of OMG IDL scopes
  - To replace `#include`

- The new `typeprefix` keyword
  - To replace `#pragma prefix`

The Dining Philosophers Example

Dining Philosophers as CORBA Components

OMG IDL 3.0 for Dining Philosophers

```plaintext
// Importation of the Components module
// when access to OMG IDL definitions contained
// into the CCM's Components module is required.
import Components;

module DiningPhilosophers
{
    // Sets the prefix of all these OMG IDL definitions.
    // Prefix generated Java mapping classes.
    typeprefix DiningPhilosophers "omg.org";
    ...
};
```
The Fork Interface

exception InUse {};
interface Fork
{
    void get() raises (InUse);
    void release();
};
// The fork component.
component ForkManager
{
    // The fork facet used by philosophers.
    provides Fork the_fork;
};
// Home for instantiating ForkManager components.
home ForkHome manages ForkManager {};

The Fork Manager Component

exception InUse {};
interface Fork
{
    void get() raises (InUse);
    void release();
};
// The fork component.
component ForkManager
{
    // The fork facet used by philosophers.
    provides Fork the_fork;
};
// Home for instantiating ForkManager components.
home ForkHome manages ForkManager {};

The Fork Manager Component Facet

exception InUse {};
interface Fork
{
    void get() raises (InUse);
    void release();
};
// The fork component.
component ForkManager
{
    // The fork facet used by philosophers.
    provides Fork the_fork;
};
// Home for instantiating ForkManager components.
home ForkHome manages ForkManager {};

The Fork Manager Home

exception InUse {};
interface Fork
{
    void get() raises (InUse);
    void release();
};
// The fork component.
component ForkManager
{
    // The fork facet used by philosophers.
    provides Fork the_fork;
};
// Home for instantiating ForkManager components.
home ForkHome manages ForkManager {};
The Philosopher State Types

```c
enum PhilosopherState {
    EATING, THINKING, HUNGRY,
    STARVING, DEAD
};
```

```c
eventtype StatusInfo {
    public string name;
    public PhilosopherState state;
    public unsigned long ticks_since_last_meal;
    public boolean has_left_fork;
    public boolean has_right_fork;
};
```

The Philosopher Component

```c
component Philosopher {
    attribute string name;
    // The left fork receptacle.
    uses Fork left;
    // The right fork receptacle.
    uses Fork right;
    // The status info event source.
    publishes StatusInfo info;
}
```

```c
home PhilosopherHome manages Philosopher {
    factory new(in string name);
};
```

The Philosopher Component Receptacles

```c
component Philosopher {
    attribute string name;
    // The left fork receptacle.
    uses Fork left;
    // The right fork receptacle.
    uses Fork right;
    // The status info event source.
    publishes StatusInfo info;
}
```

```c
home PhilosopherHome manages Philosopher {
    factory new(in string name);
};
```
The Philosopher Component Event Source

```plaintext
component Philosopher
{
  attribute string name;
  // The left fork receptacle.
  uses Fork left;
  // The right fork receptacle.
  uses Fork right;
  // The status info event source.
  publishes StatusInfo info;
};
```

```plaintext
home PhilosopherHome manages Philosopher {
  factory new (in string name);
};
```

The Philosopher Home

```plaintext
component Philosopher
{
  attribute string name;
  // The left fork receptacle.
  uses Fork left;
  // The right fork receptacle.
  uses Fork right;
  // The status info event source.
  publishes StatusInfo info;
};
```

```plaintext
home PhilosopherHome manages Philosopher {
  factory new (in string name);
};
```

The Observer Component

```plaintext
component Observer
{
  // The status info sink port.
  consumes StatusInfo info;
};
```

```plaintext
// Home for instantiating observers.
home ObserverHome manages Observer {};
```

The Observer Component

```plaintext
component Observer
{
  // The status info sink port.
  consumes StatusInfo info;
};
```

```plaintext
// Home for instantiating observers.
home ObserverHome manages Observer {};
```
The Observer Home

component Observer
{
    // The status info sink port.
    consumes StatusInfo info;
}

// Home for instantiating observers.
home ObserverHome manages Observer {};

The Client-Side OMG IDL Mapping

- Each OMG IDL 3.0 construction has an equivalent in terms of OMG IDL 2
- Component and home types are viewed by clients through the CCM client-side OMG IDL mapping
- Permits no change in client programming language mapping
  - Clients still use their favorite IDL-oriented tools like CORBA stub generators, etc.
- Clients do NOT have to be “component-aware”
  - They just invoke interface operations
Main Client-Side OMG IDL Mapping Rules

- A component type is mapped to an interface inheriting from `Components::CCMObject`
- Facets and event sinks are mapped to an operation for obtaining the associated reference
- Receptacles are mapped to operations for connecting, disconnecting, and getting the associated reference(s)
- Event sources are mapped to operations for subscribing and unsubscribing to produced events

An event type is mapped to
- A value type inheriting from `Components::EventBase`
- A consumer interface inheriting from `Components::EventConsumerBase`

A home type is mapped to three interfaces
- One for explicit operations user-defined inheriting from `Components::CCMHome`
- One for implicit operations generated
- One inheriting from both previous interfaces

Client-Side Mapping for ForkManager Component

```c++
component ForkManager
{
  provides Fork the_fork;
};
```

Is mapped to

```c++
interface ForkManager :
  ::Components::CCMObject
{
  Fork provide_the_fork();
};
```

Client-Side Mapping for Fork Home

```c++
home ForkHome
  manages ForkManager {};
  Is mapped to

interface ForkHomeExplicit :
  ::Components::CCMHome {};
interface ForkHomeImplicit :
  ::Components::KeylessCCMHome {
    ForkManager create();
  };
interface ForkHome :
  ForkHomeExplicit,
  ForkHomeImplicit {};
```
Client-Side Mapping for StatusInfo Event Type

eventtype StatusInfo { . . . };

valuetype StatusInfo :
::Components::EventBase { . . . };

interface StatusInfoConsumer :
::Components::EventConsumerBase {
  void push_StatusInfo(in StatusInfo the_StatusInfo);
};

Client-Side Mapping for Observer Component

component Observer {
  consumes StatusInfo info;
};

interface Observer :
::Components::CCMObject {
  StatusInfoConsumer get_consumer_info();
};

Client-Side Mapping for Observer Home

home ObserverHome manages Observer {};

interface ObserverHomeExplicit :
::Components::CCMHome {};
interface ObserverHomeImplicit :
::Components::KeylessCCMHome {
  Observer create();
};
interface ObserverHome :
ObserverHomeExplicit, ObserverHomeImplicit {};

Client-Side Mapping for Philosopher Component

component Philosopher {
  attribute string name;
  uses Fork left;
  uses Fork right;
  publishes StatusInfo info;
};

interface Philosopher :
::Components::CCMObject {
  attribute string name;
.../...
Client-Side Mapping
for Philosopher Component

```c
void connect_left(in Fork cnx) raises(...);
Fork disconnect_left() raises(...);
Fork get_connection_left();

void connect_right(in Fork cnx) raises(...);
Fork disconnect_right() raises(...);
Fork get_connection_right();

Components::Cookie subscribe_info(
in StatusInfoConsumer consumer) raises(...);
StatusInfoConsumer unsubscribe_info(
in Components::Cookie ck) raises(...);
```

Client-Side Mapping
for Philosopher Home

```c
home PhilosopherHome manages Philosopher {
  factory new(in string name);
}

interface PhilosopherHomeExplicit : ::Components::CCMHome {
  Philosopher new(in string name);
}
interface PhilosopherHomeImplicit : ::Components::KeylessCCMHome {
  Philosopher create();
}
interface PhilosopherHome :
  PhilosopherHomeExplicit,
  PhilosopherHomeImplicit {};
```

The Client Programming Model

- Component-aware and -unaware clients
- Clients see two design patterns
  - Factory – Client finds a home and uses it to create a new component instance
  - Finder - Client searches an existing component instance through Name Service, Trader Service, or home finder operations
- Optionally demarcation of transactions
- Could establish initial security credentials
- Invokes operations on component instances
  - Those defined by the client-side mapping

CORBA Component Home Finder

- A brokerage of homes to clients
  - Home implementations register with home finder
  - Clients request homes from home finder
- Home finder makes determination of what is the “best” home for a client, based on the client’s request and any available environmental or configuration data
- A home finder constitutes a domain of home/container/implementation visibility
Using CORBA Components with OMG IDLscript

```
# Obtains the component home finder.
chf = CORBA.ORB.resolve_initial_references("ComponentHomeFinder")

# Finds a home by its home type.
forkHome = chf.find_home_by_type(ForkHome.id())

# Creates a fork manager component.
forkManager = forkHome.create()

# Obtains the fork facet.
fork = forkManager.provide_the_fork()

# Uses the fork facet.
fork.get()

fork.release()
```

Connecting CORBA Components with OMG IDLscript

```
# Obtaining CORBA components to be interconnected.
kant = Philosopher("corbaname:...")
observer = Observer("corbaname:...")

# Connects kant and observer.
ck = kant.subscribe_info(observer.get_consumer_info())

# Disconnects kant and observer.
kant.unsubscribe_info(ck)
```

Navigation and Introspection

- Navigation from any facet to component base reference with `CORBA::Object::get_component()`
  - Returns nil if target isn't a component facet
  - Returns component reference otherwise
- Navigation from component base reference to any facet via generated facet-specific operations
- Navigation and introspection capabilities provided by `CCMObject`
  - Via the `Navigation` interface for facets
  - Via the `Receptacles` interface for receptacles
  - Via the `Events` interface for event ports

Implementing CORBA Components

- Component Implementation Framework (CIF)
- Local Server-Side OMG IDL Mapping
Component Implementation Framework

- CIF defines a programming model for constructing component implementations
  - How components should be implemented

- Facilitates component implementation
  - “only” business logic should be implemented
    - Not activation, identify, port management and introspection
  => Local server-side OMG IDL mapping
    - Interactions between implementations and containers

- Manages segmentation and persistency
  => Component Implementation Definition Language

Executors and Home Executors

- Programming artifacts implementing a component’s or component home’s behavior
  - Local CORBA objects with interfaces defined by the local server-side OMG IDL mapping

- Component executors could be monolithic
  - All component attributes, supported interfaces, facet operations, and event sinks implemented by one class

- Component executors could also be segmented
  - Component features split into several classes
  - Implements ExecutorLocator interface

- Home executors are always monolithic

Executors Are Hosted by Container

- Container intercepts invocations on executors for managing activation, security, transactions, persistency, and so

- Component executors must implement a local callback lifecycle interface used by the container
  - SessionComponent for transient components
  - EntityComponent for persistent components

- Component executors could interact with their containers and connected components through a local context interface
A Monolithic Component Executor

- Component container
- Monolithic executor
- Container context
- Main component executor interface
- Facet or event sink executor interface
- SessionComponent or EntityComponent
- Component-oriented context interface
- Container-oriented context interface
- Context use
- Container interposition

A Segmented Component Executor

- Component container
- Main segment
- Seg2
- Seg3
- Seg4
- Container context
- ExecutorLocator

Monolithic versus Segmented Approach

- Monolithic approach
  - Poor life cycle control of facet executors
  - But simplicity of implementation
  - Should be used for hand-coded implementation

- Segmented approach
  - Fine grain life cycle control of facet executors
  - But complexity of implementation
  - Should be used for CIDL based implementation

The Server-Side OMG IDL Mapping

- Component Client
- Client Application
- OMG IDL 3.0
- Client-side OMG IDL 2.x
- Implemented by
- User written
- Compiler
- Generated files
- Component Designer
- OMG IDL 3.0 Compiler
- Local server-side OMG IDL 2.x
- Component Executor
- ORB
- Component Skeleton
**Main Server-Side OMG IDL Mapping Rules**

- A component type is mapped to three local interfaces
  - The main component executor interface
    - Inheriting from `Components::EnterpriseComponent`
  - The monolithic component executor interface
    - Operations to obtain facet executors and receive events
  - The component specific context interface
    - Operations to access component receptacles and event sources

- A home type is mapped to three local interfaces
  - One for explicit operations user-defined
    - Inheriting from `Components::HomeExecutorBase`
  - One for implicit operations generated
  - One inheriting from both previous interfaces

**Implementing CORBA Components**

- Dining Philosophers Example
  - In Java
  - In C++

**Implementation Rules**

- General
  - Local server-side equivalent IDL interfaces are implemented according to the used language mapping
  - Choice between monolithic and locator implementation
  - entry point = factory for each home type

- Java specific
  - Executor classes inherit from `org.omg.CORBA.LocalObject`
  - Entry points = static methods of home executor classes

- C++ specific
  - Entry points = extern “C” functions that can be found in shared library

**Local Server-Side Equivalent IDL for ForkManager Component**
Local Server-Side Equivalent IDL for ForkManager Component

// Executor interface for the the_fork facet.
local interface CCM_Fork : Fork
{
    // No declarations added.
}

// Component-specific context interface.
local interface CCM_ForkManager_Context :
    // Container context interface.
    ::Components::CCMContext
{
    // Empty because no receptacles or event sources.
}

Fork Facet Implementation in Java:
Just Business Operations

public class ForkImpl
    extends org.omg.CORBA.LocalObject
    implements CCM_Fork
{
    private boolean available_ = true;
    
    public void get() throws InUse {
        // Check if there is no current philosopher.
        if (!available_) throw new InUse();
        available_ = false;
    }

    public void release() {
        available_ = true;
    }
}

Fork Facet Implementation in C++:
Just Business Operations

class Fork_impl : virtual public CCM_Fork
{
    bool available_;

    public:
        Fork_impl() { available_ = true; }

    void get() {
        if (!available_) throw InUse();
        available_ = false;
    }

    void release() {
        available_ = true;
    }
};
Local Server-Side Equivalent IDL for ForkManager Component

// Monolithic executor interface.
local interface CCM_ForkManager :
   // Executors base interface.
   ::Components::EnterpriseComponent
{
   // Requested by container.
   CCM_Fork get_the_fork();

   // No attributes.
};

ForkManager Executor Monolithic in Java

public class MonolithicForkManagerImpl
   extends ForkImpl    // Fork implementation.
   implements CCM_ForkManager, // Is monolithic.
                  // Is a session executor
   org.omg.Components.SessionComponent
{
   // Required by CCM_ForkManager interface.
   public CCM_Fork get_the_fork() {
      // Itself as it extends ForkImpl.
      return this;
   }

   // Also SessionComponent operations.
};

Local Server-Side Equivalent IDL for ForkManager Component

ForkManager Executor Monolithic in C++

// IDL implied by the IDL to C++ mapping.
local interface MyFork :
   CCM_ForkManager, CCM_Fork,
   Components::SessionComponent
{};

// C++
class ForkManager_impl :
   virtual public MyFork,
   virtual public Fork_impl
{
   public:
      // Required by CCM_ForkManager interface.
      CCM_Fork_ptr get_the_fork() {
         return CCM_Fork::_duplicate(this);
      }

      // Also SessionComponent operations.
};
Local Server-Side Equivalent IDL for ForkManager Component

// Main component executor interface.
local interface CCM_ForkManager_Executor :
    // Executors base interface.
    ::Components::EnterpriseComponent
{
    // Empty because no attributes.
};

Segmented ForkManager Executor With Two Segments in Java

public class MainSegForkManagerImpl
    extends org.omg.CORBA.LocalObject
    implements CCM_ForkManager_Executor,
               org.omg.Components.SessionComponent
{
    // SessionComponent to implement.
}

Segmented ForkManager Executor With Two Segments in C++

class MainSegForkManager_impl :
    virtual public CCM_ForkManager_Executor,
    virtual public Components::SessionComponent
{
    // SessionComponent to implement.
};

Local Server-Side Equivalent IDL for ForkManager Component

// Container callback implemented by the main segment.
local interface ExecutorLocator :
    ::Components::EnterpriseComponent
{
    // Obtain the specified port executor segment.
    Object obtain_executor(in string name)
        raises(CCMException);
    // Release a port executor.
    void release_executor(in Object obj)
        raises(CCMException);
    // Notify configuration completion.
    void configuration_complete()
        raises(CCMException);
};
public class ForkManagerExecutorLocatorImpl
extends org.omg.CORBA.LocalObject
implements ExecutorLocator
{
private CCM_Fork fork_;
private CCM_ForkManager_Executor mgr_;

public ForkManagerExecutorLocatorImpl()
{
fork_ = new ForkImpl();
mgr_ = new MainSegForkManagerImpl();
}

public org.omg.CORBA.Object obtain_executor(String name)
throws org.omg.Components.CCMException
{
if (name.equals("ForkManager")) return mgr_; 
if (name.equals("the_fork")) return fork_;
throw new org.omg.Components.CCMException();
}
public void release_executor(org.omg.CORBA.Object obj)
throws org.omg.Components.CCMException
{
// Nothing to do.
}
public void configuration_complete()
throws org.omg.Components.CCMException
{
// Nothing to do.
}
}

class ForkManagerExecutorLocator_impl :
virtual public ExecutorLocator
{
CCM_Fork_var fork_; 
CCM_ForkManager_Executor_var mgr_; 
public:
ForkManagerExecutorLocator_impl() 
{
fork_ = new Fork_impl;
mgr_ = new MainSegForkManager_impl;
}
CORBA::Object_ptr obtain_executor(const char* name){
if (strcmp(name, "ForkManager")==0)
    return CORBA::Object::_duplicate(mgr_);
else if (strcmp(name, "the_fork")==0)
    return CORBA::Object::_duplicate(fork_);
throw Components::CCMException();
}
void release_executor(CORBAX::Object_ptr obj) {
    // Nothing to do.
}
void configuration_complete() {
    // Nothing to do.
}
Local Server-Side Equivalent IDL for ForkHome Home

Monolithic executor

ForkHome Executor in Java

```java
public class ForkHomeImpl extends org.omg.CORBA.LocalObject implements CCM_ForkHome {
    public org.omg.Components.EnterpriseComponent create () {
        return new ForkManager...Impl();
    }
}
```

ForkHome Executor in C++

```c++
class ForkHome_impl : virtual public CCM_ForkHome {
    public org.omg.Components.EnterpriseComponent_ptr create () {
        return new ...ForkManager...Impl();
    }
};
extern "C" {
    Components::HomeExecutorBase_ptr create_ForkHome () {
        return new ForkHome_impl;
    }
}
```
Local Server-Side Equivalent IDL for Observer Component

// Container callback implemented by the component
local interface SessionComponent :
::Components::EnterpriseComponent
{
    // The context is fixed by the container.
    void set_session_context(SessionContext ctx)
        raises(CCMException);
    // Called when component is activated.
    void ccm_activate()
        raises(CCMException);
    // Called when component is deactivated.
    void ccm_passivate()
        raises(CCMException);
    // Called when component is removed.
    void ccm_remove()
        raises(CCMException);
};

Observer Executor Monolithic in Java

public class ObserverImpl
    extends org.omg.CORBA.LocalObject
implement CCM_Observer,
    org.omg.Components.SessionComponent
{
    // Required for monolithic interface.
    public void push_info(StatusInfo event) {
        ... update GUI ...
    }
}
Observer Executor
Monolithic in Java (contd)

// Required for SessionComponent interface.
public void set_session_context(SessionContext ctx)
    throws CCMException
{
    ...
}

public void ccm_activate() throws CCMException
{
    ... display GUI ...
}

public void ccm_passivate() throws CCMException
{
    ... hide GUI ...
}

public void ccm_remove() throws CCMException
{
    ... free GUI ...
}

Observer Executor
Monolithic in C++

// IDL implied by the IDL to C++ mapping.
local interface MyObserver :
    CCM_Observer,
    Components::SessionComponent {};

class Observer_impl :
    virtual public MyObserver
{
    // Required for monolithic interface.
    void push_info(StatusInfo * event) {
        ... update GUI ...
    }

Local Server-Side Equivalent IDL
for ObserverHome Home

// Required for SessionComponent interface.
void set_session_context(
    Components::SessionContext_ptr ctx)
{
    ...
}

void ccm_activate()
{
    ... display GUI ...
}

void ccm_passivate()
{
    ... hide GUI ...
}

void ccm_remove()
{
    ... free GUI ...
}
Local Server-Side Equivalent IDL for ObserverHome Home

local interface CCM_ObserverHomeExplicit :
   ::Components::HomeExecutorBase
{};

local interface CCM_ObserverHomeImplicit {
   ::Components::EnterpriseComponent create()
   raises ::Components::CreateFailure;
};

local interface CCM_ObserverHome :
   CCM_ObserverHomeExplicit,
   CCM_ObserverHomeImplicit
{};

ObserverHome Executor In Java

public class ObserverHomeImpl
   extends org.omg.CORBA.LocalObject
   implements CCM_ObserverHome
{
   // Required by CCM_ObserverHome interface.
   public org.omg.Components.EnterpriseComponent create()
   { return new ObserverImpl(); }

   // Called at deployment time.
   public static org.omg.Components.HomeExecutorBase create_home()
   { return new ObserverHomeImpl(); }
}

Observer Home
Observer

ObserverHome Executor In C++

class ObserverHome_impl :
   virtual public CCM_ObserverHome
{
   Components::EnterpriseComponent_ptr create () { return new Observer_impl; }
};
extern "C"
{ Components::HomeExecutorBase_ptr
   create_ObserverHome () { return new ObserverHome_impl; }
}

Local Server-Side Equivalent IDL for Philosopher Component

observer Home
Observer

ObserverHome Executor
In Java

public class ObserverHomeImpl
   extends org.omg.CORBA.LocalObject
   implements CCM_ObserverHome
{
   // Required by CCM_ObserverHome interface.
   public org.omg.Components.EnterpriseComponent create()
   { return new ObserverImpl(); }

   // Called at deployment time.
   public static org.omg.Components.HomeExecutorBase create_home()
   { return new ObserverHomeImpl(); }
}

Observer Home
Observer

ObserverHome Executor
In C++

class ObserverHome_impl :
   virtual public CCM_ObserverHome
{
   Components::EnterpriseComponent_ptr create () { return new Observer_impl; }
};
extern "C"
{ Components::HomeExecutorBase_ptr
   create_ObserverHome () { return new ObserverHome_impl; }
}

Monolithic Executor

Philosopher
name = XXX

Local Server-Side Equivalent IDL for Philosopher Component

observer Home
Observer

ObserverHome Executor
In Java

public class ObserverHomeImpl
   extends org.omg.CORBA.LocalObject
   implements CCM_ObserverHome
{
   // Required by CCM_ObserverHome interface.
   public org.omg.Components.EnterpriseComponent create()
   { return new ObserverImpl(); }

   // Called at deployment time.
   public static org.omg.Components.HomeExecutorBase create_home()
   { return new ObserverHomeImpl(); }
}

Observer Home
Observer

ObserverHome Executor
In C++

class ObserverHome_impl :
   virtual public CCM_ObserverHome
{
   Components::EnterpriseComponent_ptr create () { return new Observer_impl; }
};
extern "C"
{ Components::HomeExecutorBase_ptr
   create_ObserverHome () { return new ObserverHome_impl; }
}
Local Server-Side Equivalent IDL for Philosopher Component

// Main component executor interface.
local interface CCM_Philosopher_Executor :
    ::Components::EnterpriseComponent
{
    attribute string name;
};

// Monolithic executor interface.
local interface CCM_Philosopher :
    CCM_Philosopher_Executor
{
};

Philosopher name = XXX

Philosopher Executor Monolithic in Java

public class PhilosopherImpl
    extends org.omg.CORBA.LocalObject
    implements CCM_Philosopher,
                org.omg.CORBA.SessionComponent,
                java.langRunnable
{
    // Transient state.
    private String name_; 
    // Constructor.
    public PhilosopherImpl(String n) { name_ = n; }

    // Required by the CCM_Philosopher Executor interface.
    public void name(String n) { name_ = n; }
    public String name() { return name_; }

};

Philosopher Executor Monolithic in Java (2)

// The philosopher behavior state machine.
private java.lang.Thread behavior_; 

// The philosopher CCM context.
private CCM_Philosopher_Context the_context_; 

public void set_session_context(SessionContext ctx)
    throws CCMException
    {
        the_context_ = (CCM_Philosopher_Context)ctx;
    }

public void ccm_activate() throws CCMException
    {
        behavior_ = new Thread(this); behavior_.start();
    }

public void ccm_passivate() throws CCMException
    {
        behavior_.stop();
    }

public void ccm_remove() throws CCMException
    {
        ...
    }
Philosopher Executor
Monolithic in Java (3)

```java
public void run() { // The state machine.
    // Pushes the current status to all observers.
    the_context_.push_info(...);
    ...
    // Takes the left fork.
    the_context_.get_connection_left().get();
    ...
    // Takes the right fork.
    the_context_.get_connection_right().get();
    ...
    // Releases the left fork.
    the_context_.get_connection_left().release();
    ...
    // Releases the right fork.
    the_context_.get_connection_right().release();
    ...
}
```

Philosopher Executor
Monolithic in C++

```cpp
// IDL implied by the IDL to C++ mapping.
local interface MyPhilosopher : 
    Components::SessionComponent {};

// C++
class Philosopher_impl : virtual public MyPhilosopher {
    CCM_Philosopher_Context_var ctx_; 
    CORBA::String_var name_; 
    public:
        // Constructor.
        Philosopher_impl(const char* nn) { name_ = nn; }
        void name(const char* nn) { name_ = nn; }
        char* name() { return CORBA::string_dup(name_); }

    void set_session_context(
        Components::SessionContext_ptr ctx) 
    { ctx_ = CCM_Philosopher_Context::_narrow (ctx); }

    void ccm_activate() 
    { ... start philosopher thread ... }

    void ccm_passivate() 
    { ... stop philosopher thread ... }

    void ccm_remove() 
    { ... }
}
```

Philosopher Executor
Monolithic in C++ (2)

```cpp
void set_session_context(
    Components::SessionContext_ptr ctx) 
{ ctx_ = CCM_Philosopher_Context::_narrow (ctx); }

void ccm_activate() 
{ ... start philosopher thread ... }

void ccm_passivate() 
{ ... stop philosopher thread ... }

void ccm_remove() 
{ ... }
```

Philosopher Executor
Monolithic in C++ (3)

```cpp
void timer() { // The state machine.
    ... 
    ctx_->push_info(...);
    ...
    ctx_->get_connection_left()->get();
    ...
    ctx_->get_connection_right()->get();
    ...
    ctx_->get_connection_left()->release();
    ...
    ctx_->get_connection_right()->release();
    ...
}
```
Local Server-Side Equivalent IDL for PhilosopherHome Home

Monolithic executor

Philosopher Home Executor in Java

public class PhilosopherHomeImpl extends org.omg.CORBA.LocalObject implements CCM_PhilosopherHome

{ // Required by CCM_PhilosopherHomeImplicit interface.
  public org.omg.Components.EnterpriseComponent create() { return new PhilosopherImpl("unnamed"); }

  // Required by CCM_PhilosopherHomeExplicit interface.
  public org.omg.Components.EnterpriseComponent_ptr _cxx_new(const char * name)
  { return new PhilosopherImpl(name); }

  // Called at deployment time.
  public static org.omg.Components.HomeExecutorBase_ptr create_home() { return new PhilosopherHomeImpl(); }

}

Philosopher Home Executor In C++

class PhilosopherHome_impl : virtual public CCM_PhilosopherHome

{ Components::EnterpriseComponent_ptr create()
  { return new Philosopher_impl("unnamed"); }

  Components::EnterpriseComponent_ptr _cxx_new(const char * name)
  { return new Philosopher_impl(name); }

  extern "C" {
    Components::HomeExecutorBase_ptr create_PhilosopherHome()
    { return new PhilosopherHomeImpl(); }
  }

}
Implementing CORBA Components with CIDL

Component Implementation Definition Language (CIDL)

- Describes component composition
  - Aggregate entity which describes all the artifacts required to implement a component and its home

- Manages component persistence state
  - With OMG Persistent State Definition Language (PSDL)
  - Links storage types to segmented executors

- Generates executor skeletons providing
  - Segmentation of component executors
  - Default implementations of callback operations
  - Component’s state persistency

Basic CIDL Composition Features

- Component lifecycle category
  - Service, session, process, entity

- Name of home executor skeleton to generate

- Component home type implemented
  - Implicitly the component type implemented

- Name of main executor skeleton to generate

CIDL Composition for Observer Component

```c++
#include <philo.idl>
// or import DiningPhilosophers;

composition service ObserverComposition
{
    home executor ObserverHomeServiceImpl
    {
        implements DiningPhilosophers::ObserverHome;
        manages ObserverServiceImpl;
    };
};
```
OMG CIDL Compilation Process

- Component Designer
- OMG IDL 3.0
- OMG CIDL
- Component Executor
- OMG IDL 3.0 Compiler
- Component Implementer
- Local Server-side OMG IDL
- Component Skeleton

Advanced CIDL Composition Features

- Associated abstract storage home type for component persistency
- Multiple executor segments
  - Implement a subset of the component's facets
  - Could have an associated abstract storage home
- Component features stored automatically
  - Attribute values, references connected to receptacles and event sources are delegated to storage
- Proxy homes

CIDL Composition for ForkManager Component

```c
#include <philo.idl>
// or import DiningPhilosophers;

composition session ForkManagerComposition
{
  home executor ForkHomeSessionImpl
  {
    implements DiningPhilosophers::ForkHome;
    manages ForkManagerSessionImpl
    {
      segment Seg
      {
        provides facet the_fork;
      }
    }
  }
};
```

OMG PSDL for Dining Philosophers

```c
#include <philo.idl>

abstract storagetype Person {
  state string name;
  state DiningPhilosophers::PhilosopherState
    philosopher_state;
  . . .
};

abstract storagehome PersonHome of Person {
  factory create();
};

storagetype PersonBase implements Person {
  storagehome PersonHomeBase of PersonBase
    implements PersonHome;
};
```
CIDL Composition for Dining Philosophers

```cpp
#include <philo.psd1>

composition process PhilosopherComposition {
  home executor PhilosopherHomeProcessImpl {
    implements DiningPhilosophers::PhilosopherHome;
    bindsTo PersonHome;
    manages PhilosopherProcessImpl;
  };
}
```

OMG CIDL & PSDL Compilation Process

Relationship Between Artifacts

Putting CORBA Containers to Work

- The Container Model
- Container Managed Policies
The Container Model

- A framework for component application servers
- Mostly built on the Portable Object Adaptor
  - Automatic activation / deactivation
  - Resource usage optimization
- Provides simplified interfaces for CORBA Services
  - Security, transactions, persistence, and events
- Uses callbacks for instance management
- Empty container for user-defined frameworks also

The Container Architecture

Container View

- A container encapsulates one or several POAs
- A container manages one kind of component
  - entity: persistent, primary key, and explicit destruction
  - process: persistent, no key, and explicit destruction
  - session: exists during a session with the client
  - service: exists during an invocation
  - EJBsession, EJBentity: for EJBs
  - Empty: user-defined policy
- References are exported through Component HomeFinder, Naming, or Trader services

Component Categories
Container Managed Policies

- Specified by the deployer using an XML vocabulary
- Implemented by the container, not the component
- Policy declarations defined for:
  - Servant Lifetime
  - Transaction
  - Security
  - Events
  - Persistence

Servant Lifetime Policies

- **method** – valid for all categories
  - activated before each invocation
  - passivated after each invocation
- **transaction** – valid for all except service
  - activated on the first invocation of a new transaction
  - passivated after the last invocation of the transaction
- **component** – valid for all except service
  - activated before first invocation
  - passivated explicitly
- **container** – valid for all except service
  - activated on the first invocation
  - passivated when the container needs to reclaim memory

Transactions

- Container-managed at the operation level
  - NOT_SUPPORTED
  - REQUIRED
  - SUPPORTS
  - REQUIRES_NEW
  - MANDATORY
  - NEVER
- Self-managed using the `Components::Transaction::UserTransaction` API which is mapped to CORBA transactions

Security

- Most security is declarative using the component descriptors (security element)
- Container supports access to and testing of credentials at run time
- Security Permissions defined at the operation level
  - CLIENT_IDENTITY
  - SYSTEM_IDENTITY
  - SPECIFIED_IDENTITY (=userid)
- Based on CORBA Security V2
Events

- Subset of the CORBA Notification service
  - Events represented as valuetypes to components
  - Push Model
  - Container maps valuetypes to Structured Events
  - Container manages channel creation

- Quality of service properties left to configuration
- Event Policies declared in component descriptors
  - non-transactional
  - default
  - transactional

Persistence

- Supported for Entity container types only
- Container persistence policies:
  - Self managed
  - Container managed
- Both modes can use PSS or their own persistence mechanism

The Container Server Architecture

- Container Manager
- Session Container
- EJB Container
- Other Container
- Entity Container

Packaging CORBA Components
A Day in the Life of a Component

- A component is specified
  - OMG IDL 3.0, PSDL, and CIDL
- A component is implemented
  - Component Implementation Framework
- A component must be packaged
- A component may be assembled with other components
- Components and assemblies are be deployed

Packaging and Deployment

- “Classic” CORBA: No standard means of ...
  - Configuration
  - Distribution
  - Deployment

Packaging and Deployment of Components

- Components are packaged into a self-descriptive package
- Packages can be assembled
- Assemblies can be deployed
- Helped by XML descriptors

CCM Applications Deployment

- It is necessary for an application to
  - List component instances
  - Define logical location and partitioning
  - Specify connections between components
- It is necessary for a component to
  - Specify its elements
    - interfaces, implementations
  - Describe system requirements
    - OS, ORB, JVM, library releases, ...
  - Specify its initial configuration
- It is necessary for a connection to
  - Associate related component ports

The Packaging and Deployment Model

- Describes distributed CORBA component-based applications for automatic deployment

- Packaging technology
  - Self descriptive “ZIP” archives with XML descriptors
  - For heterogeneous components

- Allows interoperability between deployment tools and containers
  - Off-line by data exchange formats
  - On-line by OMG IDL interfaces
Component Package

- Archive (ZIP file) containing
  - One component, consisting of
    - One or more implementations
      - E.g. for different OSs, ORBs, processors, QoS, ...
    - OMG IDL file of the component, home and port types
    - CORBA Component Descriptor (.ccd) for required container policies
    - Property File Descriptor (.cpf) defining default attribute values
    - Software Package Descriptor (.csd) describing package contents
  - Self-contained and self-descriptive, reusable unit
  - Usually done by the component implementer

Component Assembly Package

- A component assembly is a template for a deployed set of interconnected components
- Described by an assembly descriptor in terms of component files, partitioning, and connections
- May be deployed as it as well as imported into a design tool to be reused or extended
- A “ZIP” archive containing descriptor, component archive files, and property files

Component Packaging Artifacts

- IDL/CIDL File
- Stubs, Skeletons
- Packaging Tool
- Implementation
- User’s Code
- User’s Code
- Component
- Descriptors
- Default Properties
- Home Properties
- Component Properties
- Assembly Tool
- Assembly Tool
- Component Assembly Package
- Deployment Tool
- Deployment Tool
- CORBA Component Package
- softpkg Descriptor

Component Assembly Package

- Archive (ZIP file) containing
  - One or more component packages, either
    - Including a package’s contents
    - Including the original package
    - Referencing the package by URL
  - Property File Descriptors defining initial attribute values
  - Component Assembly Descriptor (.cad)
    - Defines home instances to be created
    - Defines component instances to be created
    - Defines connections between ports to be made
  - Self-contained and self-descriptive unit
  - For automatic and easy “one step” deployment
  - No programming language experience necessary
Component Assembly Artifacts

XML Descriptors Overview

- Software Package Descriptor (.csd)
  - Describes contents of a component software package
  - Lists one or more implementation(s)

- CORBA Component Descriptor (.ccd)
  - Technical information mainly generated from CIDL
  - Some container managed policies filled by user

- Component Assembly Descriptor (.cad)
  - Describes initial virtual configuration
    - homes, component instances, and connections

- Component Property File Descriptor (.cpf)
  - Name/value pairs to configure attributes

Relationship Between CCM XML Descriptors

Software Package Descriptor (.csd)

- Descriptive general elements
  - title, description, author, company, webpage, license

- Link to OMG IDL file

- Link to default property file

- Implementation(s)
  - Information about Implementation
    - Operating System, processor, language, compiler, ORB
    - Dependencies on other libraries and deployment requirements
    - Customized property and CORBA component descriptor

- Link to implementation file
  - Shared library, Java class, executable

- Entry point
Software Package Descriptor Example

```xml
<?xml version='1.0'?><!DOCTYPE softpkg><softpkg name="PhilosopherHome"><idl id="IDL:DiningPhilosophers/PhilosopherHome:1.0"><fileinarchive name="philo.idl"/></idl><implementation id="*"><code type="DLL"><fileinarchive name="philo.dll"/><entrypoint>create_DiningPhilosophers_PhilosopherHome</entrypoint></code></implementation></softpkg>
```

Software Package Descriptor for Observer Component

```xml
<?xml version='1.0'?><!DOCTYPE softpkg SYSTEM "softpkg.dtd"><softpkg name="Observer" version="1,0,0,0"><pkgtype>CORBA Component</pkgtype><title>Observer</title><author><name>Philippe Merle</name><company>INRIA</company><webpage href="http://www.inria.fr"/></author><description>The CCM dining philosophers example</description><license href="http://www.objectweb.org/license.html"/><idl id="IDL:DiningPhilosophers/Observer:1.0"><link href="http://www.objectweb.org/philo.idl"/></idl><descriptor type="CORBA Component"><fileinarchive name="observer.ccd"/></descriptor><propertyfile><fileinarchive name="observer.cpf"/></propertyfile><implementation id="Observer_impl"><os name="WinNT" version="4,0,0,0"/><os name="Linux" version="2,2,17,0"/><processor name="x86"/><compiler name="JDK"/><programminglanguage name="Java"/><code type="Java class"><fileinarchive name="ObserverHomeImpl.class"/><entrypoint>ObserverHomeImpl.create_home</entrypoint></code><runtime name="Java VM" version="1,2,2,0"/><runtime name="Java VM" version="1,3,0,0"/><dependency>...</dependency></implementation>
```
Software Package Descriptor for Observer Component

<dependency type="ORB" action="assert">
  <name>OpenORB</name>
</dependency>

<dependency type="Java Class" action="install">
  <valuetypefactory repid="IDL:DiningPhilosophers/StatusInfo:1.0">
    <fileinarchive name="DiningPhilosophers/StatusInfoDefaultFactory.class"/>
  </valuetypefactory>
</dependency>

Software Package Descriptor for Observer Component

<implementation id="observer_0x1">
  <os name="Win2000"/>
  <processor name="x86"/>
  <compiler name="VC++"/>
  <programminglanguage name="C++"/>
  <dependency type="DLL" localfile name="jtc.dll"/>
  <dependency type="DLL" localfile name="ob.dll"/>
  <descriptor type="CORBA Component">
    <fileinarchive name="observer.ccd"/>
  </descriptor>
  <code type="DLL">
    <fileinarchive name="PhilosophersExecutors.dll"/>
    <entrypoint>create_ObserverHome</entrypoint>
  </code>
</implementation>

CORBA Component Descriptor (.ccd)

- Structural information generated by CIDL
  - Component / home types and features
  - Ports and supported interfaces
  - Component category and segments
- Container policies filled by the packager
  - Threading
  - Servant lifetime
  - Transactions
  - Security
  - Events
  - Persistence
  - Extended POA policies
- Link to component and home property files

CORBA Component Descriptor Example

<corbacomponent>
  <corbaversion>3.0</corbaversion>
  <componentrepid IDL:DiningPhilosophers/Philosopher:1.0 />
  <homerepid IDL:DiningPhilosophers/PhilosopherHome:1.0 />
  <componentkind session servant lifetime=component />
  <threading policy="multithread"/>
  <configurationcomplete set="true"/>
  <homefeatures name="PhilosopherHome" repid="IDL:...PhilosopherHome:1.0"/>
  <componentfeatures name="Philosopher" repid="IDL:...Philosopher:1.0"/>
  <ports>
    <publishes publishesname="info" eventtype="IDL:DiningPhilosophers/StatusInfo:1.0"/>
    <uses usesname="left" repid="IDL:DiningPhilosophers/Fork:1.0"/>
    <uses usesname="right" repid="IDL:DiningPhilosophers/Fork:1.0"/>
  </ports>
</corbacomponent>
CORBA Component Descriptor
for Philosopher Component

<?xml version="1.0"?>
<!DOCTYPE corbacomponent SYSTEM "corbacomponent.dtd">
<corbacomponent>
  <corbaversion>3.0</corbaversion>
  <componentrepid repid="IDL:DiningPhilosophers/Philosopher:1.0"/>
  <homerpid repid="IDL:DiningPhilosophers/PhilosopherHome:1.0"/>
  <componentkind>
    <process><servant lifetime="container" /></process>
  </componentkind>
  <security rightsfamily="CORBA"
        rightscombinator="secanyrights" />
  <threading policy="multithread" />
  <configurationcomplete set="true" />
</corbacomponent>

CORBA Component Descriptor
for Philosopher Component

<homefeatures name="PhilosopherHome"
              repid="IDL:DiningPhilosophers/PhilosopherHome:1.0"/>
<componentfeatures name="Philosopher"
                   repid="IDL:DiningPhilosophers/Philosopher:1.0">
  <ports>
    <uses usesname="right"
          repid="IDL:DiningPhilosophers/Fork:1.0" />
    <uses usesname="left"
          repid="IDL:DiningPhilosophers/Fork:1.0" />
    <publishes emitsname="info"
               eventtype="StatusInfo">
      <eventpolicy policy="normal" />
    </publishes>
  </ports>
</componentfeatures>
<intface name="Fork"
         repid="IDL:DiningPhilosophers/Fork:1.0"/>

CORBA Component Descriptor
for Philosopher Component

<segment name="philosopherseg" segmenttag="1">
  <segmentmember facettag="1" />
  <containermanagedpersistence>
    <storagehome id="PSDL:PersonHome:1.0"/>
    <pssimplemplementation id="OpenORB-PSS" />
    <accessmode mode="READ_WRITE" />
    <psstransaction policy="TRANSACTIONAL" >
      <psstransactionisolationlevel level="SERIALIZABLE" />
    </psstransaction>
    <params>
      <param name="x" value="1" />
    </params>
  </containermanagedpersistence>
</segment>

Property File Descriptor (.cpf)

- Used to set home and component properties
  - However, it could be used for anything
- Contains zero or more name/value pairs to configure attributes
- Referenced by...
  - Software Package Descriptors to define default values for component attributes
  - CORBA Component Descriptors to define default values for component or home attributes
  - Assembly Descriptors to configure initial values for home or component instances
Property Files

IDL/CIDL File

User's Code

Programming Language Tools

State, Behaviors

Implementation

Component Descriptor

Packaging Tool

Assembly Tool

Deployment Tool

IDL/CIDL Compiler

Property File For Philosopher Kant

<?xml version="1.0"?>
<!DOCTYPE properties SYSTEM "properties.dtd">
<properties>
  <simple name="name" type="string">
    <description>Philosopher name</description>
    <value>Kant</value>
    <defaultvalue>Unknown</defaultvalue>
  </simple>
</properties>

Component Assembly Descriptor (.cad)

- References one or more Component Software Descriptors
- Defines home instances and their collocation and cardinality constraints
- Defines components to be instantiated
- Defines that homes, components or ports are to be registered in the ComponentHomeFinder, Naming or Trading Service
- Defines connections to be made between component ports, e.g. receptacles to facets and event sinks to event sources

Dining Philosophers as CORBA Components

Philosopher

name = Kant

Fork

Observer

Philosopher

name = Aristotle

Philosopher

name = Descartes

Fork

Observer
Component Assembly Descriptor for Dining Philosophers

<?xml version="1.0"?>
<!DOCTYPE componentassembly SYSTEM "componentassembly.dtd">

<componentassembly id="demophilo">
  <description>Dinner assembly descriptor</description>
  <componentfiles>
    <componentfile id="PhilosopherComponent">
      <fileinarchive name="philosopher.csd"/>
    </componentfile>
    <componentfile id="ObserverComponent">
      <fileinarchive name="observer.csd"/>
    </componentfile>
    <componentfile id="ForkManagerComponent">
      <fileinarchive name="forkmanager.csd"/>
    </componentfile>
  </componentfiles>
</componentassembly>

Assembly Descriptor Example (2)

<partitioning>
  <homeplacement id="ObserverHome">
    <componentfileref idref="ObserverComponent"/>
    <registerwithnaming name="Dinner/ObserverHome"/>
  </homeplacement>
  <homeplacement id="PhilosopherHome">
    <componentfileref idref="PhilosopherComponent"/>
    <registerwithnaming name="Dinner/PhilosopherHome"/>
  </homeplacement>
  <homeplacement id="ForkHome">
    <componentfileref idref="ForkComponent"/>
    <registerwithnaming name="Dinner/ForkHome"/>
  </homeplacement>
</partitioning>
Component Assembly Descriptor
Connections for Dining Philosophers

<connections>
  <connectinterface>
    <usesport>
      <usesidentifier>left</usesidentifier>
      <componentinstantiationref idref="Kant"/>
    </usesport>
    <providesport>
      <providesidentifier>the_fork</providesidentifier>
      <componentinstantiationref idref="ForkManager1"/>
    </providesport>
  </connectinterface>
</connections>

Component Assembly Descriptor
Connections for Dining Philosophers

<connectevent>
  <publishesport>
    <publishesidentifier>info</publishesidentifier>
    <componentinstantiationref idref="Kant"/>
  </publishesport>
  <consumesport>
    <consumesidentifier>info</consumesidentifier>
    <componentinstantiationref idref="Freud"/>
  </consumesport>
</connectevent>

Component Packaging

IDL
User Code
IDL/CIDL Compiler
Generated Code
Component Descriptor
Default Properties
Compiler
Shared Library or Executable
Packaging Tool
Component Package .zip

Component Assembly

Instance Creation
Port Connections
Component Package
Assembly Tool
Assembly Archive .aar (ZIP)
Properties
Deployment Tool...
Deploying CORBA Component Applications

- Component Deployment Objects
- Component Deployment Process
- Deployment Scenario

Deployment

- An Assembly Archive is deployed by a deployment tool
- The deployment tool might interact with the user to assign homes and components to hosts and processes
- The deployment application interacts with installation objects on each host

Deployment Objects

- ComponentInstallation
  - Singleton, installs component implementations
- AssemblyFactory
  - Singleton, creates Assembly objects
- Assembly
  - Represents an assembly instantiation
  - Coordinates the creation and destruction of component assemblies and components
- ServerActivator
  - Singleton by host, creates ComponentServer objects
- ComponentServer
  - Creates Container objects
- Container
  - Installs CMHome objects

The Component Deployment Process
The Component Deployment Process

- **Deployment Tool**
- **AssemblyFactory**
- **Assembly**
- **ServerActivator**
- **ComponentServer**
- **Container**
- **CCMHome**
- **ComponentInstallation**
- **CCMObject**

Deployment API: Assembly

```plaintext
module Components {
    enum AssemblyState {
        INACTIVE, INSERVICE
    }
    exception CreateFailure {};
    exception RemoveFailure {};

    interface Assembly {
        void build () raises (CreateFailure);
        void tear_down () raises (RemoveFailure);
        AssemblyState get_state ();
    }
}
```

Deploying the Philosophers Example

- Run Deployment Application
  - Use ComponentInstallation to upload component implementations
  - Use AssemblyFactory to create an Assembly
  - Call build() operation on Assembly Interface
    - starts ComponentServers for each home
    - creates Containers and installs homes
    - creates component instances
    - interconnects component ports
    - calls configuration_complete

- One-step installation!

Deployment Scenario
Deployment Scenario: Deployment Tool

Deployment Scenario: Assembly Creation

Deployment Scenario: Component Server Instantiation

Deployment Scenario: Container Instantiation
Deployment Scenario: Home Installation

Component Assembly Descriptor +

Home for A

Container

Code for Component A

Assembly

Home for B

Container

Code for Component B

Deployment Scenario: Component Instantiation

Component Assembly Descriptor +

Home for A

Assembly

Home for B

Deployment Scenario: Component Configuration

Component Assembly Descriptor +

Home for A

Assembly

Home for B

A instance

B instance

Summary
Conclusion

- 1st open standard for Distributed Component Computing
  - Component-based software engineering process
  - Advanced component model
  - Server-side container framework
  - Packaging and distributed deployment
  - EJB interworking
  - Component meta models

- Heart of CORBA 3.0
  - Available specification since the Yokohama meeting
  - ~ 500 pages added

Next CCM Steps at OMG

- Deployment and Configuration RFP
  - OMG TC Doc orbos/2002-01-19

- CORBA Component Model Revision Task Force
  - Chartered at Yokohama meeting (April 26th 2002)

- UML Profile for CCM RFP
  - In preparation / discussion
  - Revision of the UML Profile for CORBA for including IDL 3.0 extension, PSDL, and CIDL

- EDOC to CCM Mapping RFP
  - Should be prepared

Open Source CCM Implementations

- OpenCCM from LIFL & ObjectWeb
  - Java on ORBacus 4.1 & OpenORB 1.2.1
  - http://www.objectweb.org/OpenCCM/

- MicoCCM from FPX & Alcatel
  - C++ on MICO
  - http://www.fpx.de/MicoCCM/

- CIF from Humboldt University
  - C++ on ORBacus 4.1
  - http://sourceforge.net/projects/cif

Commercial CCM Implementations

- Qedo from Fraunhofer FOKUS
  - C++ on MICO & ORBacus 4.1
  - http://qedo.berlios.de

- EJCCM from CPI Inc.
  - Java on OpenORB 1.3.x
  - http://www.ejccm.org

- K2 from ICMP
  - C++ on various ORBs
  - http://www.icmgworld.com
More Information

- CORBA 3.0: New Components Chapters
  - OMG TC Document ptc/2001-11-03

- CORBA 3 Fundamentals and Programming
  - Dr. John Siegel, published at John Wiley and Sons

- “The CCM Page”, Diego Sevilla Ruiz
  - http://www.ditec.um.es/~dsevilla/ccm/