CORBA Component Model
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
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
What is the CCM Specification?

- 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
From CORBA Component Design to Packaging

Component Designer

OMG IDL, PSDL & CIDL

Component Implementer

Local server-side OMG IDL

Component Executor Code

Component Client

OMG IDL PSDL & CIDL Compiler

Stubs, Skeletons

Programming Language Tools

Client-side OMG IDL

XML Component Descriptor

Binary Component

Component Packager
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
Component Clients

- 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 overload 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

IDL/CIDL File

User's Code

Programming Language Tools

Stubs, Skeletons Implementation

Component Descriptor

Packaging Tool

CORBA Component Package

CORBA Component Package

softpkg Descriptor

CORBA Component Package

Deployment Tool

designers

implementer

packager

assembler

deployer
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

My Business Component

Component interface

Facets

Event sinks

OFFERED

Attributes

REQUIRED

Event sources

Receptacles

Attributes

Event sinks

Facets

Component interface
Building CCM Applications = Assembling CORBA Component Instances
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

MyBusinessHome

c1

...
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
The Dining Philosophers Example

- Thinking
- Hungry
- Starving
- Eating
- Dead

- Kant

- Descartes

- Aristotle

- Thinking
- Hungry
- Starving
- Eating
- Dead
Dining Philosophers as CORBA Components

Philosopher
name = Kant

Philosopher
name = Descartes

Philosopher
name = Aristotle

Fork

Fork

Fork

Observer

Component
Base ref.
Facet
Receptacle
Event Sink
Event Source
// 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.
    typeofprefix 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 { };
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 {};}
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 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
The Client-Side OMG IDL Mapping

Component Designer

User written
Compiler
Generated files

Component
Client

Client Application

uses

Component

OMG IDL 3.0

uses

implemented by

Client-side OMG IDL 2.x

OMG IDL

3.0

 Compiler

Client Stub

ORB

uses

Component

uses

uses

uses

uses

uses

uses

uses
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
Main Client-Side OMG IDL Mapping Rules

- 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

```plaintext
component ForkManager {
    provides Fork the_fork;
};
```

Is mapped to

```plaintext
interface ForkManager :
    ::Components::CCMObject {
    Fork provide_the_fork();
};
```
Client-Side Mapping for Fork Home

```
home ForkHome
    manages ForkManager {};

interface ForkHomeExplicit :
    ::Components::CCMHome {};

interface ForkHomeImplicit :
    ::Components::KeylessCCMHome {
        ForkManager create();
    };

interface ForkHome :
    ForkHomeExplicit,
    ForkHomeImplicit {};
```

Is mapped to

ForkHome

Fork Manager
Client-Side Mapping for StatusInfo Event Type

eventtype StatusInfo { . . . };

Is mapped to

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

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

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
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 in C++

- Dining Philosophers Example
exception InUse {};

interface Fork {
    void get () raises (InUse);
    void release ()
};

component ForkManager {
    provides Fork the_fork;
};
Server Side equivalent IDL for ForkManager

// Executor interface for the the_fork facet.
local interface CCM_Fork : Fork {
}

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

// Monolithic executor interface.
local interface CCM_ForkManager :
    CCM_ForkManager_Executor {
    // Requested by container.
    CCM_Fork get_the_fork();
};
// Component-specific context interface.
local interface CCM_ForkManager_Context :
    ::Components::CCMContext
{
    // Empty because no receptacles or event sources.
};
Different ForkManager Implementations

- Fork facet implementation
  - class `Fork_impl`

- Monolithic approach
  - By inheritance: `ForkManager_1_impl`
  - By delegation: `ForkManager_2_impl`

- Executor locator approach
  - Segmented: `ForkManager_3_Executor_impl` and `ForkManager_3_Locator_impl`
class Fork_impl : virtual public CCM_Fork
{
    bool available_; 

public:
    Fork_impl () { available_ = true; } 

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

public void release()
{
    available_ = true;
}
};
ForkManager Implementation (1): Monolithic, Inheritance of Facet

// IDL implied by the IDL to C++ mapping.
local interface MyFork : CCM_ForkManager, CCM_Fork {};  
// C++
class ForkManager_1_impl :
    virtual public MyFork,
    virtual public Fork_impl
{
    public:
        // facet implemented by myself
        CCM_Fork_ptr get_the_fork ()
        {
            return CCM_Fork::_duplicate (this);
        }
};
ForkManager Implementation (2): Monolithic, Delegation of Facet

class ForkManager_2_impl :
    virtual public CCM_ForkManager
{
    CCM_Fork_var the_fork_;

    public:
    ForkManager_2_impl () {
        the_fork_ = new Fork_impl;
    }
    CCM_Fork_ptr get_the_fork () {
        return CCM_Fork::_duplicate (the_fork_);
    }
};
ForkManager Implementation (3): Locator based

class ForkManager_3_Executeor_impl :
    virtual public CCM_ForkManager_Executor
    { /* empty */ };

class ForkManager_3_Locator_impl :
    virtual public Components::ExecutorLocator
    {
    CCM_ForkManager_Executor_var _executor;
    CCM_Fork_var _the_fork;
    public:
    ForkManager_3_Locator_impl ()
    {
        _executor = new ForkManager_3_Executeor_impl;
        _the_fork = new Fork_impl;
    }
ForkManager Implementation (3): Locator based (contd)

/* MyFork_3_Locator_impl continued */

CORBA::Object_ptr
obtain_executor (const char * name) {
    if (strcmp (name, "ForkManager") == 0)
        return CORBA::Object::_duplicate (_executor);
    else
        return CORBA::Object::_duplicate (_the_fork);
}

void release_executor (CORBA::Object_ptr obj)
{ /* empty */ }

void configuration_complete ()
{ /* empty */ }
};
local interface CCM_ForkHomeExplicit :
::Components::HomeExecutorBase {
    // Empty
};

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

local interface CCM_ForkHome :
    CCM_ForkHomeExplicit,
    CCM_ForkHomeImplicit {};

Server Side equivalent IDL for ForkHome
class **ForkHome_impl** :

    virtual public **CCM_ForkHome**

    {
        // from the implicit interface
        Components::EnterpriseComponent_ptr create ()
        {
            return new **ForkManager_1_impl**;
            // or: return new ForkManager_2_impl;
            // or: return new ForkManager_3_Locator_impl;
        }
    }

    extern “C” {
        Components::HomeExecutorBase_ptr create_**ForkHome** ()
        {
            return new **ForkHome_impl**;
        }
    }
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**
#include <philo.idl>
// or import DiningPhilosophers;

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

Component Designer

OMG IDL 3.0

Composition Designer

OMG CIDL

Component Implementer

Component Executor

includes imports

OMG IDL 3.0 Compiler

Local Server-side OMG IDL

Component Skeleton

OMG CIDL Compiler

Component Executor Skeleton

inherited by and completed

delegates to

partially implemented

User written

Compiler

Generated files
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
Relationship Between Artifacts

```
component C {};
home H manages C {};
```

```
home executor HE {}
  implements H;
  bindsTo SH;
  manages E;
};
```

```
abstract storagetype ST{}
  abstract storagehome SH manages ST {};
```

Diagram:

- Home
  - implements Executor
  - manages Component

- Executor
  - implements Home
  - manages Storage Home

- Storage Home
  - manages Storage Object

- Component
  - implements Executor

Legend:

- IDL
- CIDL
- PSDL
- Explicitly defined
- Implicitly defined
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 Packaging Artifacts

- IDL/CIDL File
- User's Code
- Programming Language Tools
- IDL/CIDL Compiler
- Stubs, Skeletons
- Implementation
- Component Descriptor
- Packaging Tool
- Default Properties
- Home Properties
- Component Properties
- CORBA Component Package
- Assembly Tool
- CORBA Component Package
- Component Assembly Package
- Assembly Descriptor
- Deployment Tool
- softpkg Descriptor
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 is 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 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

IDL/CIDL File

User's Code

Programming Language Tools

IDL/CIDL Compiler

Stubs, Skeletons

IDL/CIDL File

User's Code

Programming Language Tools

Stubs, Skeletons

Component Descriptor

Packaging Tool

Default Properties

Home Properties

Component Properties

CORBA Component Package

Assembly Tool

Component Assembly Package

Deployment Tool

CORBA Component Package
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
  - Final CCM specification approved begin 2002
  - ~ 500 pages added
CCM Implementations

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

- Qedo Fraunhofer FOKUS/Humboldt University
  - C++ on MICO & ORBacus
  - http://sourceforge.net/projects/cif

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

- CORBA 3.0.3
  - OMG TC Document formal/2004-03-12
- CORBA Component Model, v3.0
  - OMG TC Document formal/2002-06-65
- CORBA 3 Fundamentals and Programming
  - Dr. John Siegel, published at John Wiley and Sons
- CORBA Komponenten
  - Bertram Neubauer, Tom Ritter, Frank Stoinski
- “The CCM Page”, Diego Sevilla Ruiz
  - http://www.ditec.um.es/~dsevilla/ccm/