Limitations of Object-Based Middleware

Object-Oriented programming is a standardised technique, but…

- Lack of defined interfaces between objects
- It is hard to specify dependencies between objects
- Internal configuration of objects
- Long time to install new applications

**Components**

- Components are functional entities – they must define what they offer and what they need
- Application construction: composition of objects with defined interfaces
- Semi-automatic deployment of the application: Run-time environment configuration

Components in CORBA

**CORBA Component Model (CCM)**

- Distributed, component-oriented model
- **Components** are binary code fragments for functional entities, maybe implemented in different languages
- Application deployment by definition of functional entities and interactions between entities
- Standardised environment for
  - Component management: creation, activation, deactivation, …
  - CORBA services for components (Security, Persistence, Event, Transactions)
- Interoperability with Enterprise Java Beans (EJB)

"CORBA Component Model is a specification for creating server-side scalable, language-neutral, transactional, multi-user and secure enterprise-level applications"

The CORBA Component Model

- Extends the CORBA Object Model
- Provides standard run-time environment for components
  - application servers
  - containers
- Component: reusable physical entity
- Container: standardised execution environment for components
- Application server: generic server process
- Adoption of POA
- Uses CORBA services: Transactions, Security, Event, …

**CORBA Component**

- Component is a new CORBA meta-type.
  - Extension of type `Object` (with constraints)
  - Has an interface, and an object reference
  - Also, a stylized use of CORBA interfaces/objects
- Provides component features (also named **Ports**)
- Could support multiple interfaces
- Each component instance is created and managed by a unique component home

A component describes:

- What functionality a component offers to other components
- What functionality a component needs from other components
- What kind of interaction are used between components: synchronous or asynchronous
- Which component properties can be configured
- Which manager is responsible for managing component instances
A CORBA Component

Offered operation interfaces

Facets

Component interface

Event sources

Event sinks

Attributes

Required operation interfaces

OFFERED

REQUIRED

consumer events

produced events

configurable properties

Building CCM Applications is Assembling Component Instances

Multiple named interfaces that provide the component’s application functionality to clients - one of these must be the same as the supported interface

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

Facets

Receptacles

Connection points between components, where one uses an interface of another

No inherent life cycle dependencies or ownership relationship implied - no operations are inherently transitive across receptacle connection

“Plug in” to the receptacle one or more references to instances of the required component type

- Explicit client relationship
- Ability to specialize by delegation, compose functions
- IDL compiler generates operations to connect to and disconnect from the receptacle
## Events

- Decoupled communication between components
  - Receptacle supports direct communication between components
  - In contrast, events are indirect: Event channel contacts set of consumers for each event
- Simple event model based on channels
- Subset of CORBA Notification Service ("push" model)
- Components can declare that they
  - Produce a kind of event (event source)
    - Emit (1:1)
    - Publish (1:N)
  - Accept a kind of event (event sinks)

## Attributes

- Named configurable properties
- Allow component configuration on an instance basis
- Determine behavior (within range of possible behaviors) for particular component instance
- "Configurator" objects can configure components by establishing attribute values
  - Attributes exposed through accessors and mutators
  - Signal when completed and then validity checked
- Can be configured
  - By visual property sheet mechanisms in assembly or deployment environments
  - By homes or during implementation initialization
  - Potentially readonly thereafter

## Homes

- A home manages a specified component type
  - Home definition is distinct from component definition
  - More than one home type can manage the same component type (but any given component instance has only one home)
  - Allows life cycle characteristics or key type to vary/evolve without changing component definition
- Instantiated at deployment time

## A CORBA Component Home

![Diagram of a CORBA Component Home](image)
A HomeFinder is used to find out the reference for the home interface belonging to a component type. All homes can be registered here (is not done automatically). 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 to service a client, based on the client’s request and any available environmental or configuration data.

```plaintext
resolve_initial_reference("ComponentHomeFinder")
gives back reference to HomeFinder.
```

### IDL Example

```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;
}

home PhilosopherHome manages
    Philosopher
    {
        factory new (in string name);
    };
```
**Philosopher State Types**

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

**Philosopher Component**

```csharp
class PhilosopherComponent
{
    public string name;
    public enum PhilosopherState state;
    public long ticks_since_last_meal;
    public bool has_left_fork;
    public bool has_right_fork;
}
```

```csharp
class PhilosopherHome
{
    factory new (in string name);
}
```

**Fork Component**

```csharp
class ForkComponent
{
    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 {};
}
```

**Observer Component**

```csharp
class ObserverComponent
{
    interface Observer
    {
        void get() raises (InUse);
        void release();
    } // The observer component.
    component Observer
    {
        // The status info sink port.
        consumes StatusInfo info;
    } // Home for instantiating observers.
    home ObserverHome manages Observer {};
}
```
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
- Invokes operations on component instances