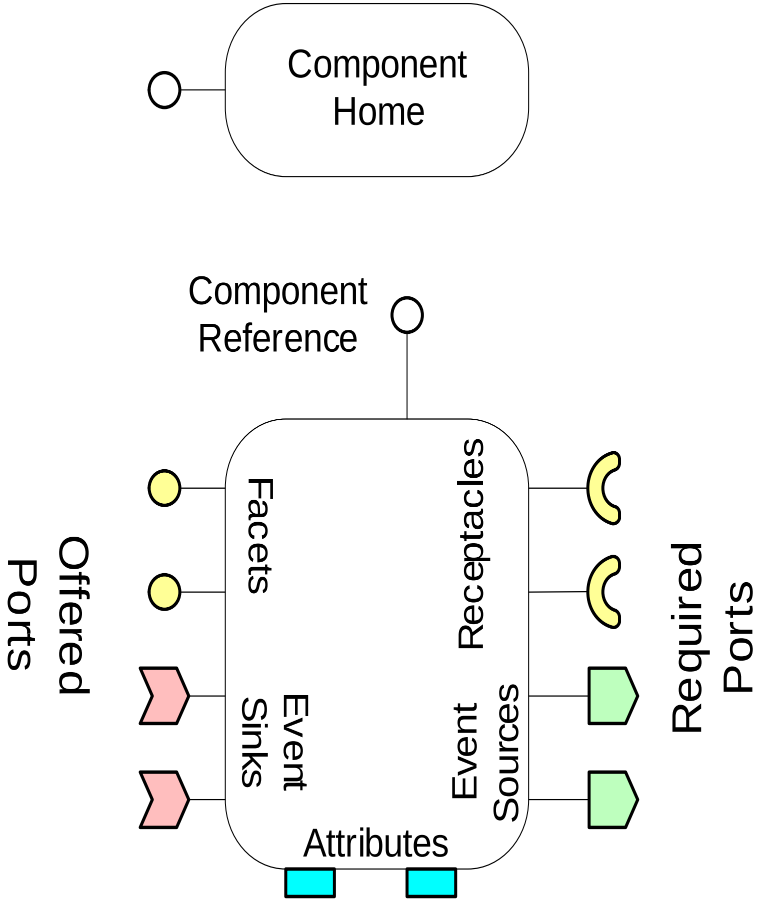
Chap7. Component-based Development

Part 7.2 CCM Component Definition

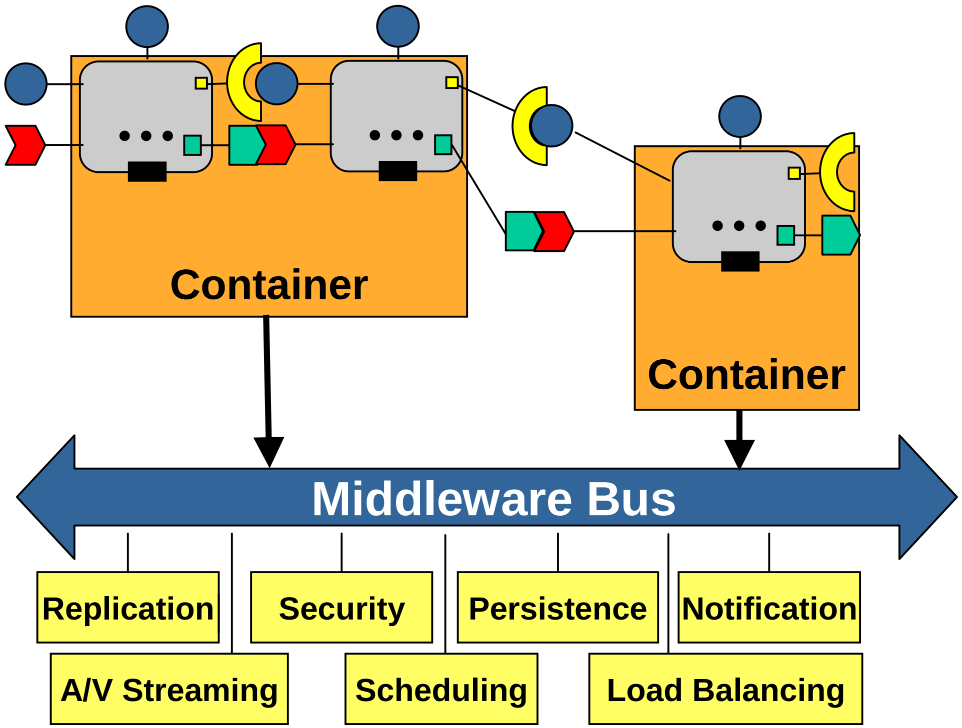
1. CCM Component Model
2. Extended IDL
3. Equivalence
4. Component Implementation Definition Language (CIDL)
5. Implementing a CCM Component
6. Extending the Basic Example

1. CCM Component Model

Component Features



Components Assembly



2. Extended IDL

Overview

-The CCM (CORBA 3.x) introduces new IDL constructs that support component types. This comes in addition of features already available for interface definitions (CORBA 2.x).

-However component instances are accessed through regular CORBA object references. That is made possible by defining what is called the *Component Equivalent Interface*.

-Component equivalent interface is a regular CORBA interface, generated automatically, that carries all the operations associated with the component.

- These include custom operations from supported interfaces as well as generic operations derived from and associated with the components ports (e.g., facets, receptacles, etc.)

Components Definition

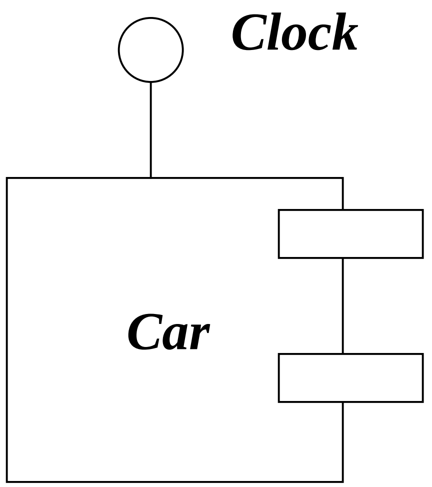
-Component types are declared using the keyword ***component***.

-The equivalent interface supported by the component may inherit from some user-defined interfaces. This relationship is expressed using a ***supports*** clause on the component declaration.

÷That's the single way component definitions may introduce new operations.

÷ A support clause may refer to a single interface or to several interfaces related by inheritance.

-*Example*



//IDL Code

module vehicle {

interface Clock {
Time getTime ();
void ResetTime (in Time t);
};
***component Car supports Clock** {};*
};

Components Facets

-Facets correspond to the interfaces provided by a component. Facets are declared using the keyword ***provides***.

-*Notation*

component XXX {

  provides <interface_type> <facet_name>;

 };

-*Example*

module motors {

  interface Engine{};

  interface Panel {};

 component Car supports Clock{

   provides Engine _engine;

   provides Panel _panel;

 };

};

Components Receptacles

-Correspond to the interfaces required by a component to function in a given environment.

-A receptacle is defined by using the keyword ***uses*** followed by the name of the receptacle.

-There are 2 kinds of receptacles: *simplex* receptacle and *multiplex* receptacle.

Simplex Receptacle

-Can be connected to only one object.

-*Notation*

component XXX {
  uses <interface_type> <receptacle_name>;
};

-*Example*

interface Customer {};
  component Account {
    uses Customer owner;
  };

Multiplex receptacle

÷Can be connected to several objects.

Notation

component XXX {
  uses multiple <interface_type> <receptacle_name>;
 };

Example

component Account {
    uses ***multiple*** *Customer owner;*
};

Event Sources and Sinks

-Event-driven communication is used as alternative to invocation-based communication, in order to decouple an object from its environment.

Event Type

-Notifications values are defined using CORBA ***valuetype*** type, which is derived from Components::EventBase; ***eventtype*** is a specialization of value type dedicated to asynchronous component communication.

Notation

```
eventtype<name> {
    //attributes
 };
```

Example

```
module stockbrocker {
  eventtype AlertSignal{
      public string reason;
   };
 …
};
```

Publishers

-The keyword ***publishes*** is used to define an event source named *publisher* that allows only 1-to-n communication,which makes it equivalent to a multiplex receptacle.

Notation

```
component XXX {
  publishes <event_type> <source_name>;
};
```

Example

```
module stockbroker {
  eventtype AlertSignal{
      public string reason;
   };

  component Broker {
      publishes AlertSignal alert_source;
   };
 };
```

Emitters

-Correspond to event sources involved in point-to-point communications with only one consumer; they are defined using keyword ***emits***.

Notation

```
component XXX {
  emits <event_type> <source_name>;
};
```

Example

```
module stockbrocker {
   eventtype StockLimit {
       public long stock_value;
    };

   component Broker {
      emits StockLimit limitAlert;
    };
};
```

Event Sink

-An event sink (or consumer) is declared using the keyword ***consumes***.

Notation

```
component XXX {
 consumes <event_type> <sink_name>;
};
```

Example

```
module stockbrocker {
  eventtype AlertSignal {
      public string reason;
   };
  component Trader {
      consumes AlertSignal alert_sink;
   };
};
```

Attributes

-Attributes ports are defined and used for component configuration.

÷They are defined in the same way as for interface, but are primarily and typically used for configuration purposes. The equivalent interface will carry pairs of get and set methods.

```
component Broker {
    attribute string broker_name;
    emits StockLimit limitAlert;
  };
```

Component Homes

- A CORBA component is managed by a special entity called a *home*, which provides life cycle and additional services on behalf of the component.
  - ÷Homes provide factory operations that are used to create instances of the components they manage. They also provide some operations that are used to locate and retrieve pre-existing component instances.

- A home manages component instances of a specific type.
  - ÷Multiple home types can manage the same component type; however a component instance is associated to a unique home instance.

- A home is declared using the **home** keyword.

home *BrockerHome* ***manages*** *Brocker { };*

- Equivalent interfaces are generated for homes as well.

3. Equivalence

-The ***cidl*** compiler generates from the IDL 3.x definition equivalent IDL 2.x code, and the supporting ***Component Implementation Framework (CIF)*** necessary to develop and deploy the component.

Component Equivalent Interface

-A *component equivalent interface* is generated for every component.

-Component equivalent interface is a regular CORBA interface, that carries equivalent operations associated with the features (e.g., facets, receptacles, events etc.) of the component.

Example:

```
component Car supports Clock{
    …
}
```

The equivalent interface for Car component would be:

```
interface Car:Components::CCMObject, Clock{
  //equivalent operations definitions for ports and interfaces
};
```

Facets

Notation

provides <interface_type> <facet_name> ();

Equivalence

<interface_type> provide_<facet_name> ();

-Clients of a component instance can invoke corresponding method to obtain a reference to the facet.

Example

-The equivalent interface for *Car* component, would be as follows:

```
interface Car:Components::CCMObject, Clock{
  Engine provide_engine();
  Panel provide_panel();
};
```

```
module motors {
  interface Engine{};
  interface Panel {};

  component Car supports Clock{
    provides Engine _engine;
    provides Panel _panel;
  };
};
```

Receptacles

Simplex Receptacles

Notation

uses <interface_type> <receptacle_name>;

Equivalence

-Equivalent IDL will contain methods that clients can use to connect/disconnect to the given receptacle.

```
void connect_<receptacle_name> (in <interface_type> cnxn)
                    raises (Components::AlreadyConnected,Components::InvalidConnection);

<interface_type> disconnect_<receptacle_name>()  raises(Components::NoConnection);

<interface_type> get_connection_<receptacle_name> ();
```

Example

```
interface Customer {};
 component Account {
  uses Customer owner;
 };
```

```
interface Account {

  //connections operations for receptacle owner;
   void connect_owner(in Customer conxn) raises(Components::AlreadyConnected,
                                                Components::InvalidConnection);
   Customer disconnect_owner() raises(Components::NoConnection);

   Customer get_connection_owner();
};
```

Multiplex Receptacles

Notation

uses multiple <interface_type> <receptacle_name>;

Equivalence

```
struct <receptacle_name>Connection {
  <interface_type> objref;
  Components::Cookie ck;
};

sequence <<receptacle_name>Connection> <receptacle_name>Connections;

Components::Cookie connect_<receptacle_name> (in <interface_type> cnxn)
                raises (Components::ExceededConnectionLimit,Components::InvalidConnection);

<interface_type> disconnect_<receptacle_name>(in Components::Cookie ck)
                raises(Components::NoConnection);

<receptacle_name>Connections get_connections_<receptacle_name> ();
```

Event Sources and Sinks

Publisher

Notation

publishes <event_type> <source_name>;

Equivalence

Components::Cookie subscribe_<source_name> (in <event_type>Consumer consumer)
raises(Components::ExceededConnectionLimit);
<event_type>Consumer unsubscribe_<source_name> (in Components::Cookie ck);

Example

-The equivalent interface generated for the event supplier broker component will include the following:

```
interface Broker:Components::CCMObject {
  Components::Cookie subscribe_alert_source(in AlertSignalConsumer consumer)
                                    raises(Components::ExceededConnectionLimit);
  AlertSignalConsumer unsubscribe_alert_source(in Components::Cookie ck)
                                    raises (Components::InvalidConnection);
 };
```

```
module stockbroker {
  eventtype AlertSignal{
     public string reason;
  };

  component Broker {
     publishes AlertSignal
                  alert_source;
  };
};
```

Emitter

Notation

emits <event_type> <source_name>;

Equivalence

void subscribe_<source_name> (in <event_type>Consumer consumer)
raises(Components::AlreadyConnected);
<event_type>Consumer unsubscribe_<source_name> () raises (Components::NoConnection);

Consumer

Notation

consumes <event_type> <sink_name>;

Equivalence

<event_type>Consumer get_consumer_<sink_name>();

Example

Example

```
module stockbrocker {
  eventtype AlertSignal {
      public string reason;
   };
  component Trader {
      consumes AlertSignal alert_sink;
   };
};
```

-The equivalent interface generated for event consumer *Trader* component is as follows:

```
interface Trader:Components::CCMObject {
    AlertSignal get_consumer_alert_sink();
};
```

4. Component Implementation Definition Language (CIDL)

-CIDL is used to describe internal aspects and characteristics of component irrelevant to clients, but essential for code generation and deployment in containers such as a component's category.

-In contrast, IDL is used to describe external characteristics of a component such as its interfaces, which are relevant to clients.

-CIDL:

- Describes a component's *composition*
  - Aggregate entity that associates *interfaces* with all artifacts required to implement a particular component & its home *executors*
- Can also manage component persistence state
  - Via OMG *Persistent State Definition Language* (PSDL)
  - (Not part of Lightweight CCM)

Component Categories

-There are four categories of CORBA components:

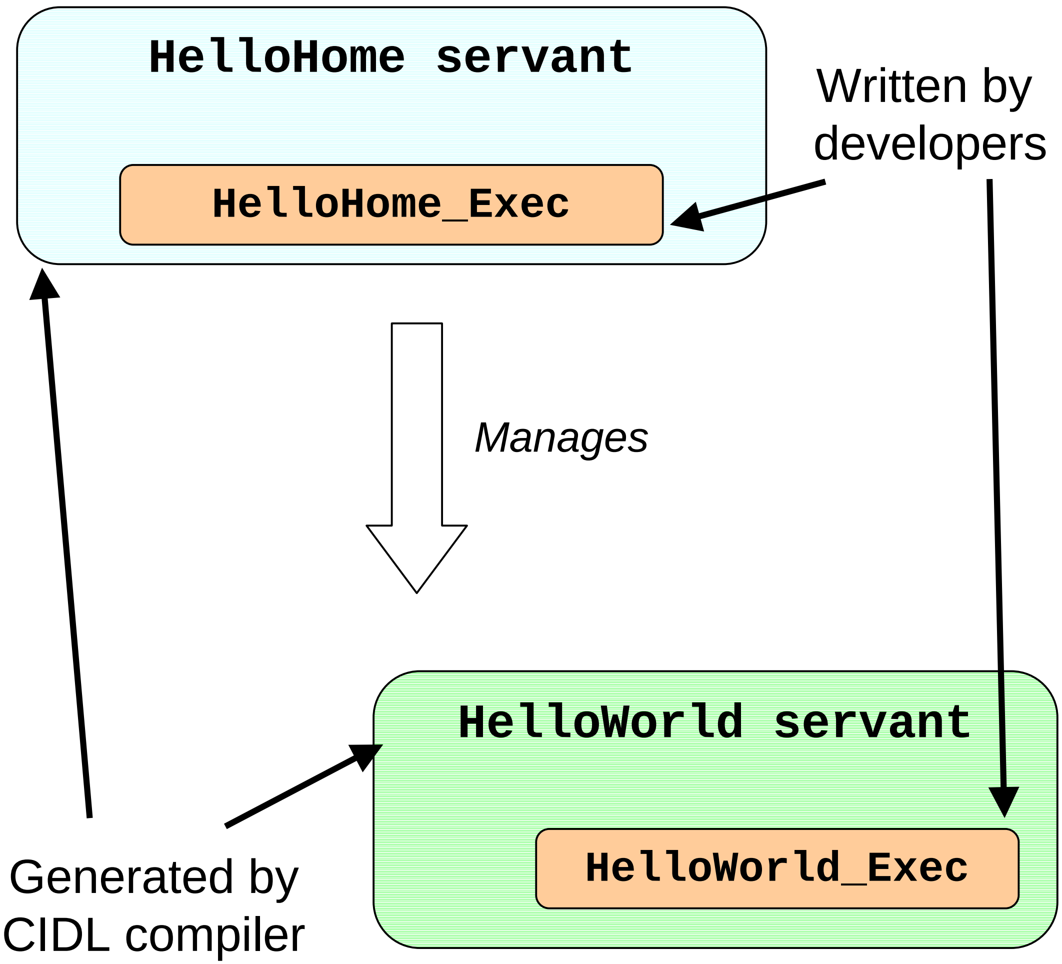
- *Service component*: has only a transient lifetime, and may exist only for the duration of a single operation.
- *Session component*: have only transient lifetime and no persistent state, their lifetime typically correspond to the duration of a client interaction.
- *Process component:* has both a persistent lifetime and persistent state, and is used to model business processes.
- *Entity component:* is used to model persistent entities; key difference with other component types is that it has a primary key.

Component category	**CORBA Usage Model**	**Object Reference**	**Container API Type**	**Primary key**	**EJB Type**
Service	Stateless	Transient	session	-	-
Session	Conversational	Transient	session	-	session
Process	Durable	Persistent	entity	-	-
Entity	Durable	Persistent	entity	yes	entity

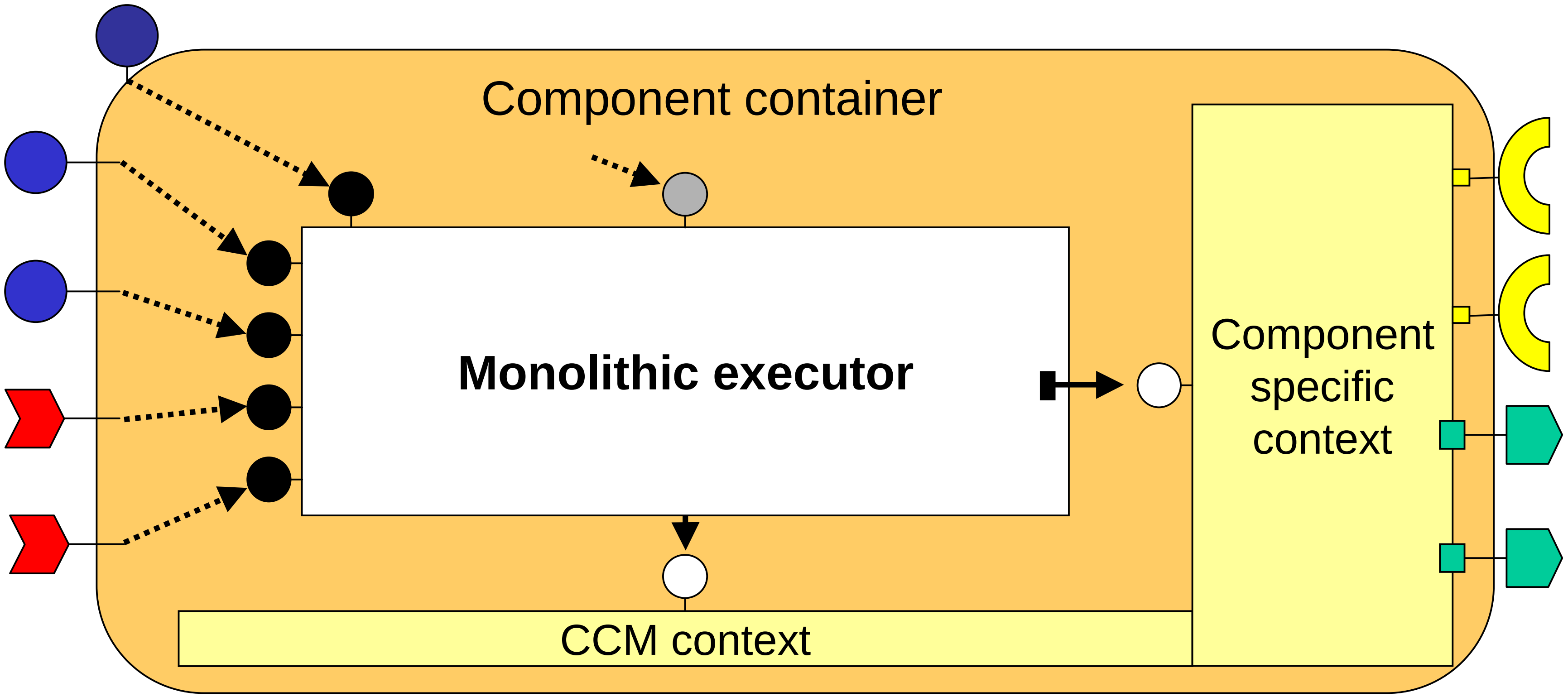
Component Executors & Home Executors

- Server-side programming artifacts that implement components & homes
  - Local CORBA objects with interfaces defined by a local server-side OMG IDL mapping

- Component executors can be
  - *Monolithic*, where all component ports implemented by one class, or
  - *Segmented*, where component ports split into several classes
- Home executors are always monolithic



A Monolithic Component Executor



Main component executor interface

Facet or event sink executor interface

SessionComponent or **EntityComponent**

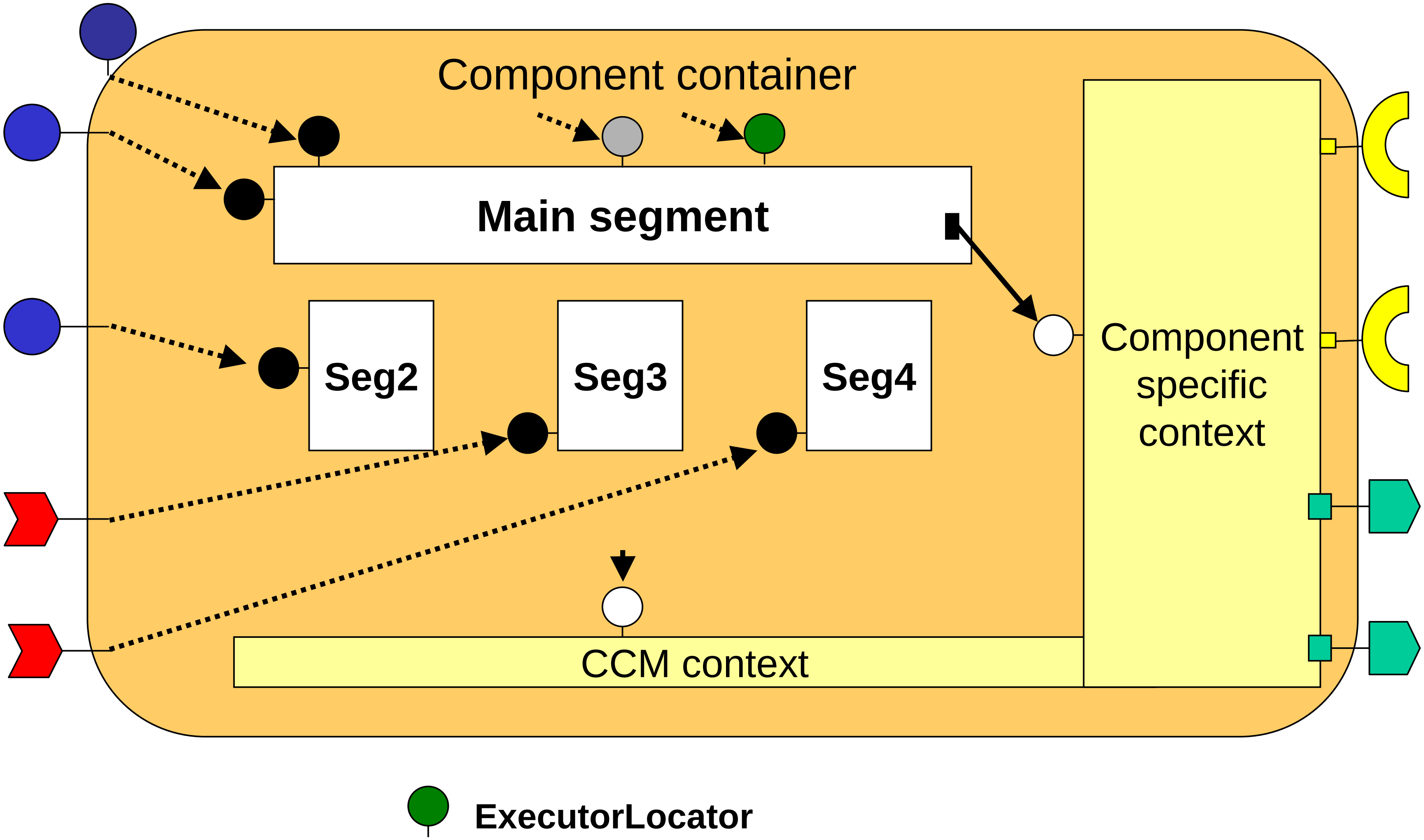
Component-oriented context interface

Container-oriented context interface

Context use

Container interposition

A Segmented Component Executor



Segmented executors are deprecated in favor of assembly-based components

Composition

-Top-level construct used to describe a component.

÷Defines the component category and the names of the component home and container *executors* in the target programming language.

÷An executor is equivalent to the implementation in target programming language. In Java, for instance, the executor for home and container correspond to Java classes.

Composition structure

```
composition <category> <composition_name> {
        home executor <home_executor_name> {
                implements <home_type> ;
                manages <executor_name>;
        };
};
```

-Example:

```
component Broker {
  attribute string broker_name;
  emits StockLimit limitAlert;
};
home BrokerHome manages Broker {}
composition process BrokerImpl {
  home executor BrokerHomeImpl {
    implements BrokerHome;
    manages BrokerProcessImpl;
  };
```

```
composition <category> <composition_name> {
    home executor <home_executor_name> {
        implements <home_type> ;
        manages <executor_name>;
    };
};
```

- The code generator generates *BrokerHomeImpl* and *BrokerProcessImpl* as abstract classes. Developers must subclass them, in order to implement the business logic.

Example

```
// USER-SPECIFIED IDL
//
module LooneyToons {
        interface Bird {
          void fly (in long how_long);
        };
        interface Cat {
            void eat (in Bird lunch);
        };
        component Toon {
            provides Bird tweety;
            provides Cat sylvester;
        };
        home ToonHome manages Toon {};
};
```

```
// USER-SPECIFIED CIDL
//
import ::LooneyToons;
module MerryMelodies {
// this is the composition:
        composition session ToonImpl {
          home executor ToonHomeImpl {
            implements LooneyToons::ToonHome;
            manages ToonSessionImpl;
                };
        };
};
```

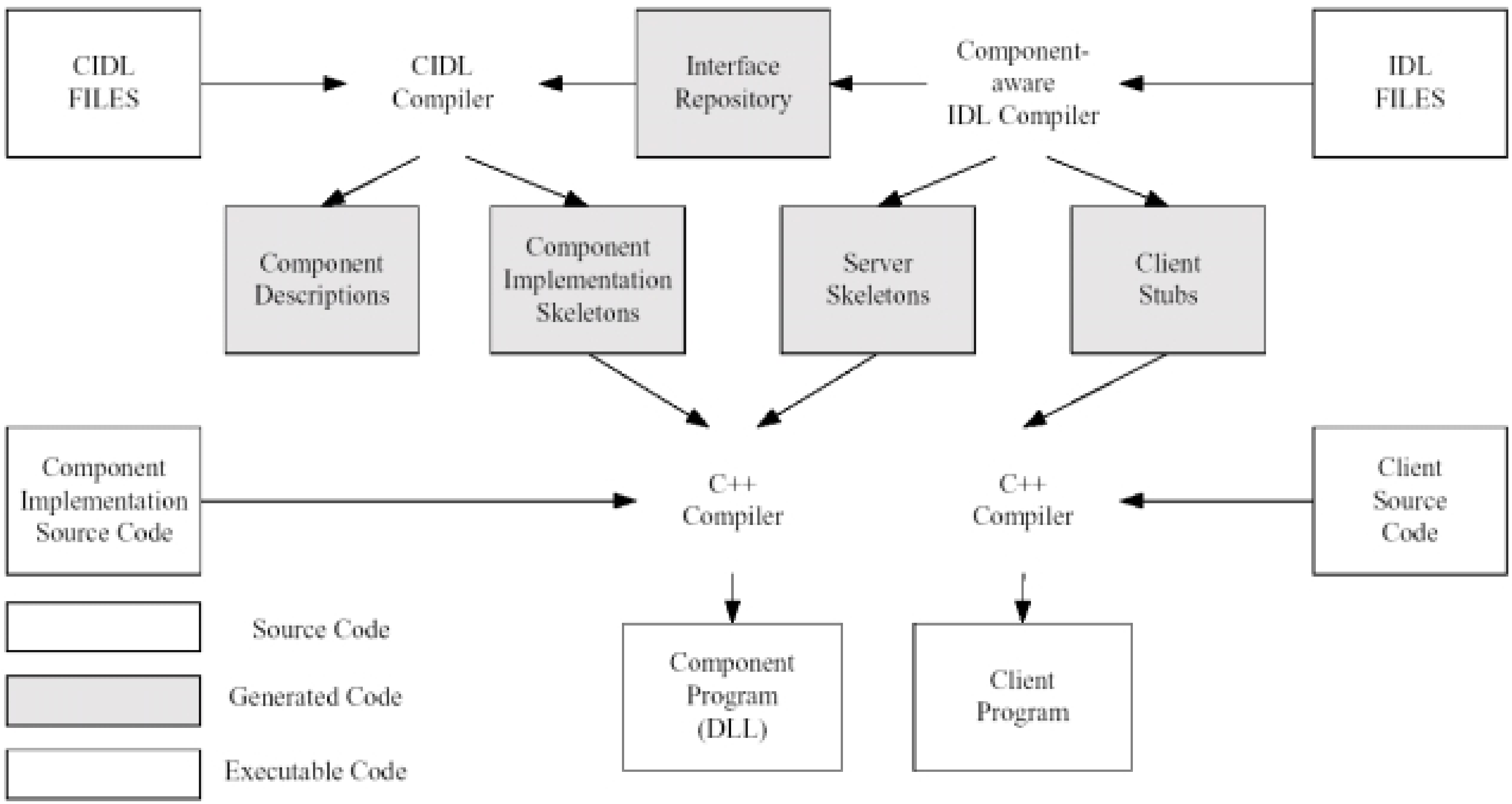
5. Implementing a CCM Component

CCM Component Creation and Deployment: Process

-The development of a typical CCM component is carried according to the following steps:

1. Specification
2. Design/Interface Definition
3. Implementation
4. Packaging
5. Assembling with other components
6. Deployment of components and assemblies

Implementing Components: Generated Files



A Basic Example

Writing the IDL

-We consider a calculator service that provides mathematical functions:

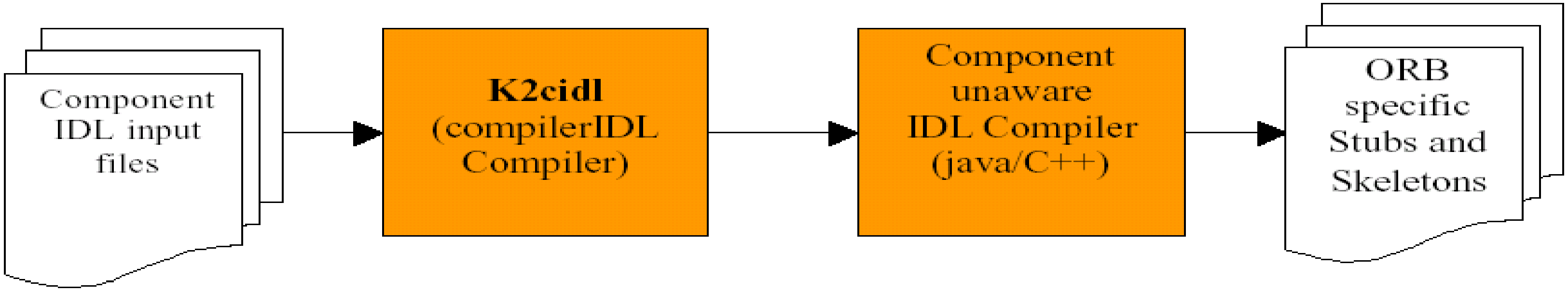
```
//Calculator.idl
    #include "Components.idl"
    module CalculatorModule {
          interface Functions {
                  long  factorial (in long number);
                };
         component CalculatorComp  {provides Functions function;};
         home CalculatorCompHome manages CalculatorComp {};
       };
```

Compiling the IDL

-Use the K2 CIDL compiler to convert component IDL code (CORBA 3.0) to standard CORBA IDL (CORBA 2.3).

K2cidl --extended-components Calculator.idl

÷The generated file (Calculator.idl2) can be compiled using IDL compilers provided by vendors



-Compilation of idl generates following files:

Calculator.cxx	C++ Stub code
Calculator.hxx	
Calculator.idl2	CORBA 2.0 idl generated from .idl file, used to generate stub files for other languages
Calculator_skel.cxx	Skeleton code
Calculator_skel.hxx	
Calculator_skel_tie.cxx	Skeleton code for tie approach
Calculator_skel_tie.hxx	
CalculatorC.i	Orb related files
CalculatorS.i	
CalculatorS_T.i	

Writing the CIDL

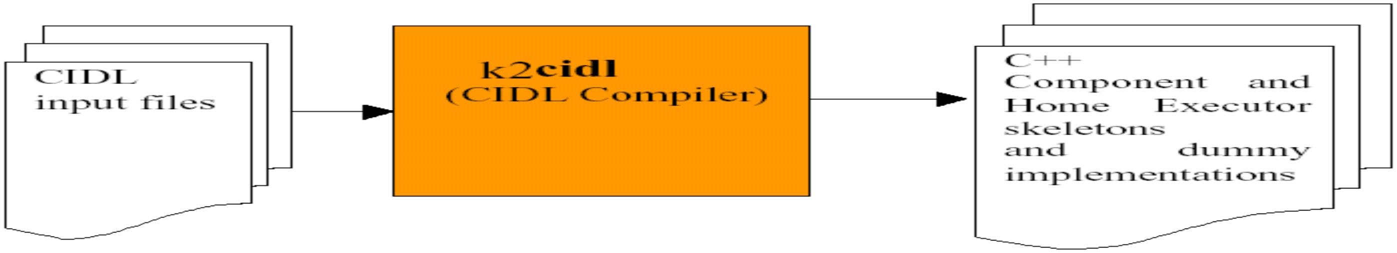
-The CIDL definition supports the automatic generation of the Component Implementation Framework (CIF) required for deploying the component within a container.

```
//Calculator.cidl
    #include "Calculator.idl"
     module CalculatorCIDL {
        composition service CalculatorCompImpl {
            home executor CalculatorCompHomeImpl {
               implements CalculatorModule::CalculatorCompHome;
               manages CalculatorCompServiceImpl;
             };
           };
        };
```

Compiling the CIDL

-The K2 CIDL compiler generates skeleton code, default implementations and XML descriptors for the CIDL definition.

K2cidl --impl -all --gen-desc Calculator.cidl



-The following files are generated from CIDL compilation:

Calculator_cimpl.cpp	Component implementation files (template)
Calculator_cimpl.h	
Calculator_cskel.cpp	Skeleton code
Calculator_cskel.h	
CalculatorModule_CalculatorComp.ccd	CORBA Component Descriptor
CalculatorModule_CalculatorComp.cpf	Component Property File
CalculatorModule_CalculatorComp.csd	Component Softpack Descriptor
tmpk2d.k2d	Used by K2 server

-Component implementation file generated after cidl compilation:

```
// ***********************************************************************
// Generated by the  CIDL to .cpp Translator
// Copyright (c) 2000 2001
// Internet Component Management Group
// All Rights Reserved
// ***********************************************************************

#include "Calculator_cimpl.h"
#include <k2/Tools.h>

/**
  *IDL:CalculatorModule/Functions/factorial:1.0
  */
CORBA::Long CalculatorCIDL::CalculatorCompServiceImpl _cimpl::
                    factorial(CORBA::Long) throw(CORBA::SystemException)
{
                    //TODO Implementation
                    CORBA::Long tmp = 0;
                    return tmp;
}
```

Implementing the Component

-Write the business logic by implementing the *Functions* interface:
modify corresponding methods prototypes (in *Calculator_cimpl.cpp*)
and provide the implementation:

```
CORBA::Long CalculatorCIDL::CalculatorCompServiceImpl _cimpl::
                    factorial(CORBA::Long number) throw(CORBA::SystemException)
  {
                    CORBA::Float tmp = 0;
                    if (number > 1)  tmp = (number*factorial(number-1));
                  else tmp= 1;
                    return tmp;
   }
```

-Compile the component implementation code using *make* utility, which
generates a shared object (*libCaculatorComp.so/CalculatorComp.dll*)
that can be loaded by the container.

Packaging the Component

-The component implementation has to be compiled to obtain the dynamic link library (dll) and then archive it together with component descriptors. This gives us the component package.

÷Use nmake utility by providing makefile.mak as the input:

```
nmake /f Makefile.mak
```

÷Makefile.mak defines all the procedures to create the dll for the component, groups the dll and description files, and puts them into a zip file. The following file will be generated:

Calculator.zip

Deploying the Component

-A component is deployed under the form of a component package in XML format, which represents the minimal deployment unit.

(see Tools Instructions Manual for details about specific platform).

Writing the Client

-The client accesses the deployed component using the component home specified in the component IDL definition.

```
#include <k2/CompatiblePlatform.h>
#include <k2/CompatibleCorba.h>

//Include the stub generated after IDL compilation of the idl2 file
#include GEN_CLIENT_INCLUDE(Calculator)
//Tools.h provides a client side framework for accessing ORB and K2 services
#include <k2/Tools.h>
using namespace CalculatorModule;
int main(int argc, char* argv[]) {
  CORBA::ORB_var orb;
  try {
 //Initialize the ORB and K2 related services; this returns a handle to access ORB
 // and K2 Trading service
        K2Utils::Tools* pK2tools= K2Utils::Tools::init(argc,argv);

//Returns a reference to Trader service; the location of Trader must be specified
//in a property file indicating HTTP host and port where K2Daemon is running
        K2Trading::Lookup_var lookup = pK2tools->getK2Trader();
        assert(!CORBA::is_nil(lookup));
```

```
//Locate a Home reference by querying the K2 Trader using the component home
// name. The Trader returns a load balanced reference to a component home, which
// is casted to obtain the Component Home reference.
        K2Trading::Offer_var offer = lookup->queryBest("CalculatorCompHome","");
         CalculatorCompHome_var home = CalculatorCompHome::_narrow(offer->reference);
         assert(!CORBA::is_nil(home));

//Invoke the create method on the Home to obtain a Component instance reference.
         CalculatorComp_ptr calculator_comp = home->create();

//Use the component instance; in this example, method factorial is invoked using
// component reference.
         long n=100;
         cout << "!" << n << " = " <<  calculator_comp->factorial(n);

          pK2tools->cleanup();
      }
   catch (const CORBA::Exception& ex) {
          cerr << "ERROR: " << argv[0] << ": " << endl;
          return 1;
      }
  } //end of main
```

Testing the Component

1. Use the Management console to:
   - ÷Install the package Calculator.zip
   - ÷Start a CCM server and load ***Calculator*** component into the CCM server instance.

2. Execute the client:

   client –K2PropFile=client.cfg

   - ÷The ***client.cfg*** file indicates where the K2 daemon is currently executing (can be obtained from the ***k2daemon.cfg*** file):

     ## HTTP Daemon properties
     k2.HTTPSERVER.NAME = <host name>
     k2.HTTPSERVER.PORT = <port-no>

6. Extending the Basic Example

-We consider a new component named *Generator* that uses the calculator component to generate some id.

The IDL

```
//Calculator.idl
  #inlude "Components.idl"
   module CalculatorModule {
        interface Functions {
                long  factorial (in long number);
               };
        interface IdGenerator {
            long generate ();
         }

        component CalculatorComp  {provides Functions function;};
        home CalculatorCompHome manages CalculatorComp {};
        component GeneratorComp {
                                        provides IdGenerator;
                                        uses Functions;
                                       };
        home GeneratorCompHome manages GeneratorComp {};
      };
```

The CIDL

```
//Generator.cidl
   #include Calculator.idl
     module GeneratorCIDL {
       composition session GeneratorCompImpl {
          home executor GeneratorCompHomeImpl {
             implements Calculator::GeneratorCompHome;
             manages GeneratorCompSessionImpl;
           };
        };
       };
```

Compiling the CIDL

```
K2cidl  --gen-desc --impl-all Calculator.cidl
K2cidl  --gen-desc --impl-all Generator.cidl
```

Writing the Components Implementations

-The Generator component uses a reference to the calculator component, which may be resolved in the constructor and stored as private variable, in ***GeneratorCompSessionImpl_cimpl***.

//add the private reference variables to
// ***GeneratorCompSessionImpl_cimpl*** *class*

```
private:
  //ORB Reference
  CORBA::ORB_var orb;

  //Trader reference
  K2Trading::Lookup_var trader;

  //Reference to the calculator component
  CalculatorComp_ptr comp_calculator;
```

//add the following code to the constructor of //GeneratorCompSessionImpl_cimpl

```
K2Utils::Tools* pK2tools= K2Utils::Tools::init(argc,argv);
K2Trading::Lookup_var lookup = pK2tools->getK2Trader();
 assert(!CORBA::is_nil(lookup));
K2Trading::Offer_var offer = lookup->queryBest("CalculatorCompHome","");
 CalculatorCompHome_var home = CalculatorCompHome::_narrow(offer->reference);
 assert(!CORBA::is_nil(home));

  //Initialize the calculator component reference
  comp_calculator = home->create();
```

-Add the following implementations for the methods:

```
long generate () {
   try {
         long r = rand();
         return comp_calculator->factorial(r);
        }
   catch (const CORBA::Exception& ex) {
         cerr <<_LINE_<< " -> ERROR: " << ": " << ex  << endl;
        }
    }
```

-Include the additional header files in ***Generator_cimpl.h*** file

#include GEN_SERVER_INCLUDE(Calculator)

#include <k2/Tools.h>

Testing the Application

-To test the application:

÷Use make to compile and package the components

÷Deploy the components in the following sequence: *calculator*, and then *generator*.

÷Execute the client (The client can be written as seen previously):

client –K2PropFile=client.cfg