Tutorial on the Lightweight CORBA Component Model (CCM)

http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/CIAO/OMG-CCM-Tutorial.pptx

William R. Otte
Research Scientist
Vanderbilt University
Nashville, Tennessee

Johnny Willemsen
Remedy IT
Nijkerk, Netherlands

Douglas C. Schmidt
Professor of EECS
Vanderbilt University
Nashville, Tennessee

Remedy IT

Other contributors include Tao Lu, Gan Deng, Jai Balasubramanian, Kitty Balasubramanian, Bala Natarajan, William R. Otte, Jeff Parsons, Frank Pilhofer, Craig Rodrigues, Nanbor Wang, & Johnny Willemsen

Tutorial on the Lightweight CORBA Component Model (CCM)

Industrializing the Development Distributed Real-time & Embedded Systems

Dr. Douglas C. Schmidt
d.schmidt@vanderbilt.edu
http://www.dre.vanderbilt.edu/~schmidt/

Professor of EECS Vanderbilt University Nashville, Tennessee

Other contributors include Tao Lu, Gan Deng, Jai Balasubramanian, Kitty Balasubramanian, Bala Natarajan, William R. Otte, Jeff Parsons, Frank Pilhofer, Craig Rodrigues, Nanbor Wang, & Johnny Willemsen

Tutorial on the Lightweight CORBA Component Model (CCM)

Industrializing the Development of Distributed Real-time & Embedded Systems

Douglas C. Schmidt

Vanderbilt University
Nashville, Tennessee

Frank Pilhofer

Mercury Computer Systems
Boston, Massachusetts

Other contributors include Jai Balasubramanian, Kitty Balasubramanian, Gan Deng, Tao Lu, Bala Natarajan, Jeff Parsons, Craig Rodrigues, Nanbor Wang, & Johnny Willemsen

Tutorial Overview

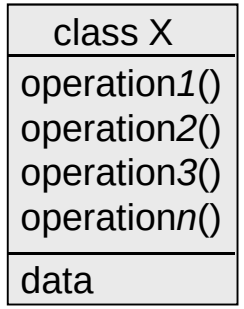
- The purpose of this tutorial is to
  - Motivate the need for the CORBA Component Model (CCM) & contrast it with the CORBA 2.x distributed object computing (DOC) model
  - Introduce CCM features most relevant to distributed real-time & embedded (DRE) applications
    - e.g., Lightweight CCM & the new OMG Deployment & Configuration spec
  - Show how to implement DRE applications using CCM & C++
  - Illustrate status of CCM & Lightweight CCM support in existing platforms
- but not to
  - Enumerate all the CCM C++ or Java mapping rules & features
  - Provide detailed references of all CORBA & CCM interfaces
  - Make you capable of implementing CORBA & CCM middleware

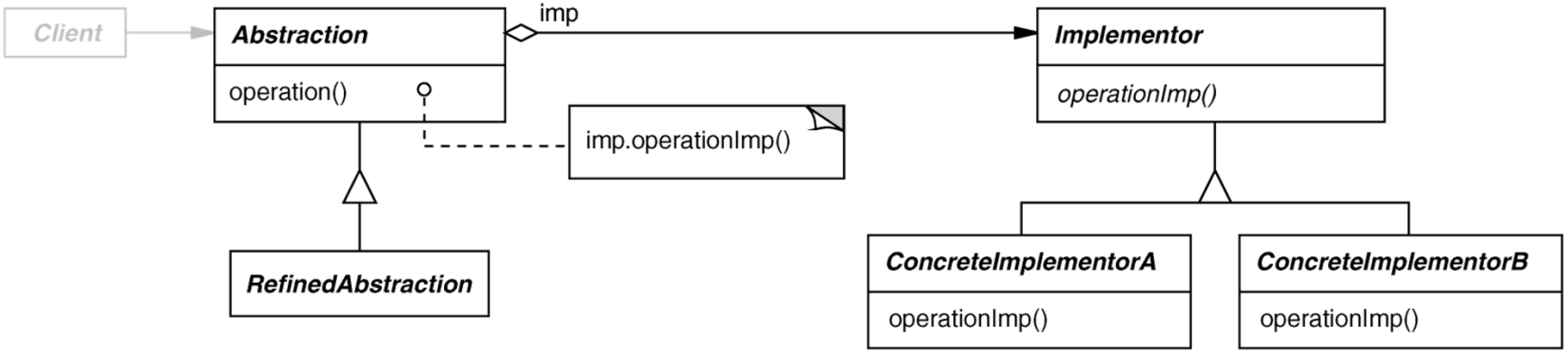
Motivation & Overview of Component Middleware

www.cs.wustl.edu/~schmidt/cuj-16.doc

Where We Started: Object-Oriented Programming

- Object-oriented (OO) programming simplified software development through higher level abstractions & patterns, e.g.,
  - Associating related data & operations
  - Decoupling interfaces & implementations

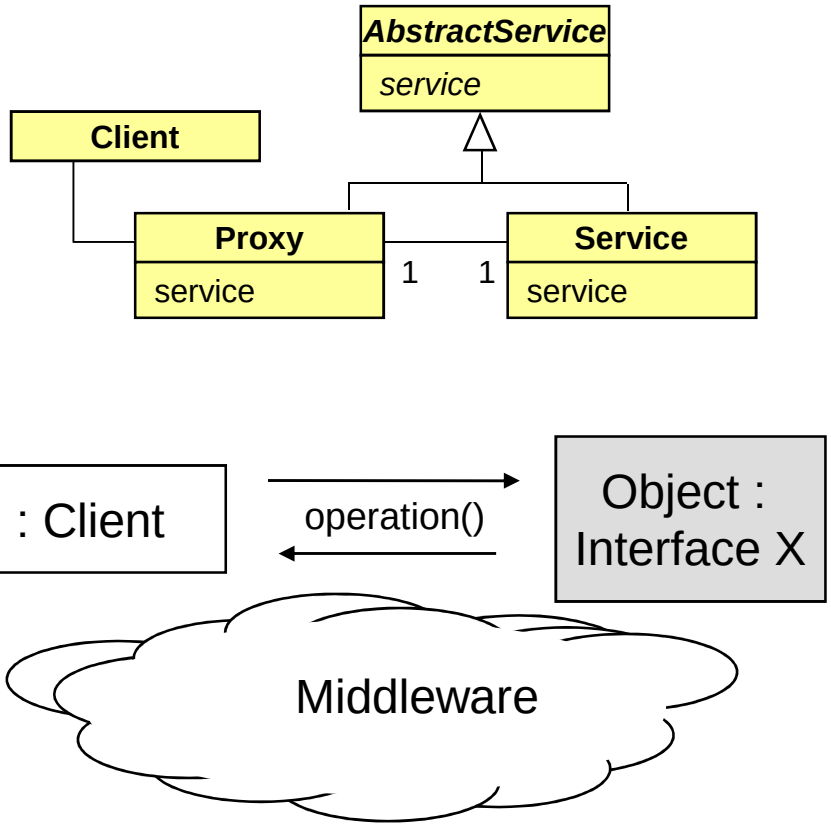




Well-written OO programs exhibit recurring structures that promote abstraction, flexibility, modularity, & elegance

Next Step: Distributed Object Computing (DOC)

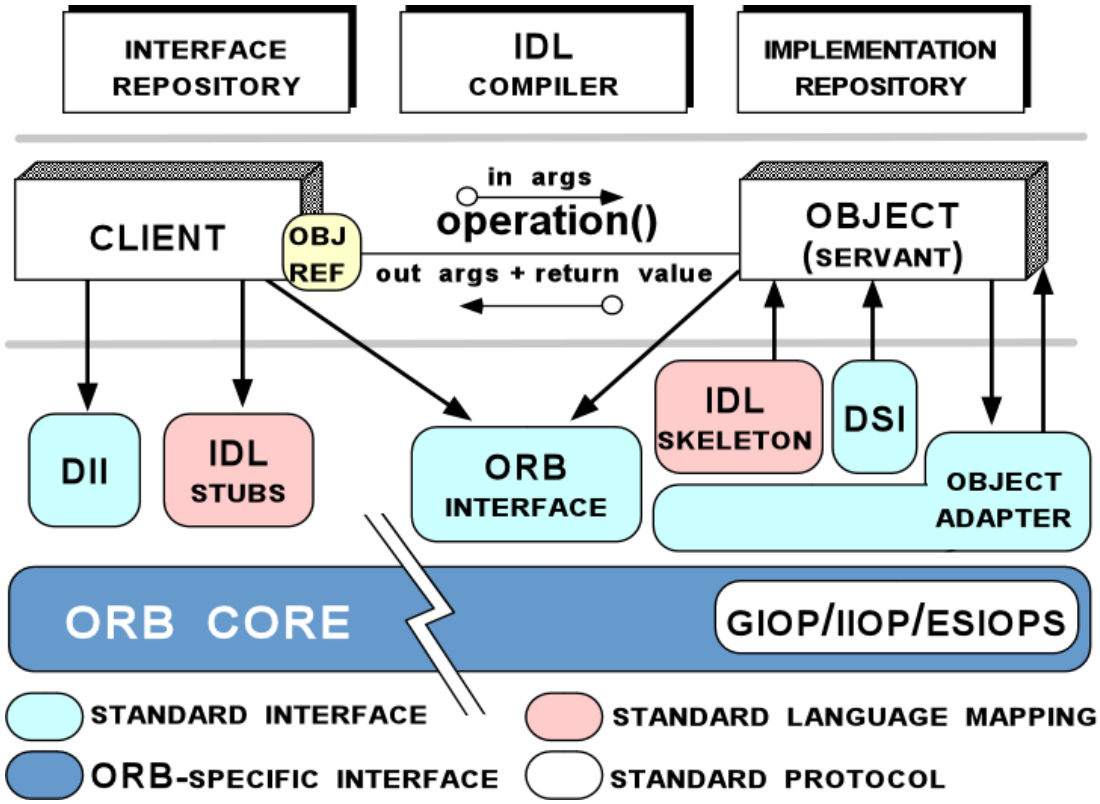
- Applies the Broker pattern to abstract away lower-level OS & protocol-specific details for network programming
- Creates distributed systems that are easier to model & build using OO techniques
- Result: robust distributed systems built with *distributed object computing (DOC) middleware*
  - e.g., CORBA, Java RMI, etc.



We now have more robust software & more powerful distributed systems

Overview of CORBA 2.x Standard

- CORBA 2.x is DOC middleware that shields applications from *dependencies* on heterogeneous platforms
  - *e.g.,* languages, operating systems, networking protocols, hardware



- CORBA 2.x automates
  - Object location
  - Connection & memory mgmt.
  - Parameter (de)marshaling
  - Event & request demultiplexing
  - Error handling & fault tolerance
  - Object/server activation
  - Concurrency & synchronization
  - Security

CORBA 2.x defines interfaces & policies, but *not* implementations

Example: Applying OO to Network Programming

- CORBA 2.x IDL specifies *interfaces* with operations
  - Interfaces map to objects in OO programming languages
    - e.g., C++, Java, Ada95, Ruby, etc.

```
interface Foo
{
  void bar (in long arg);
};
```

IDL

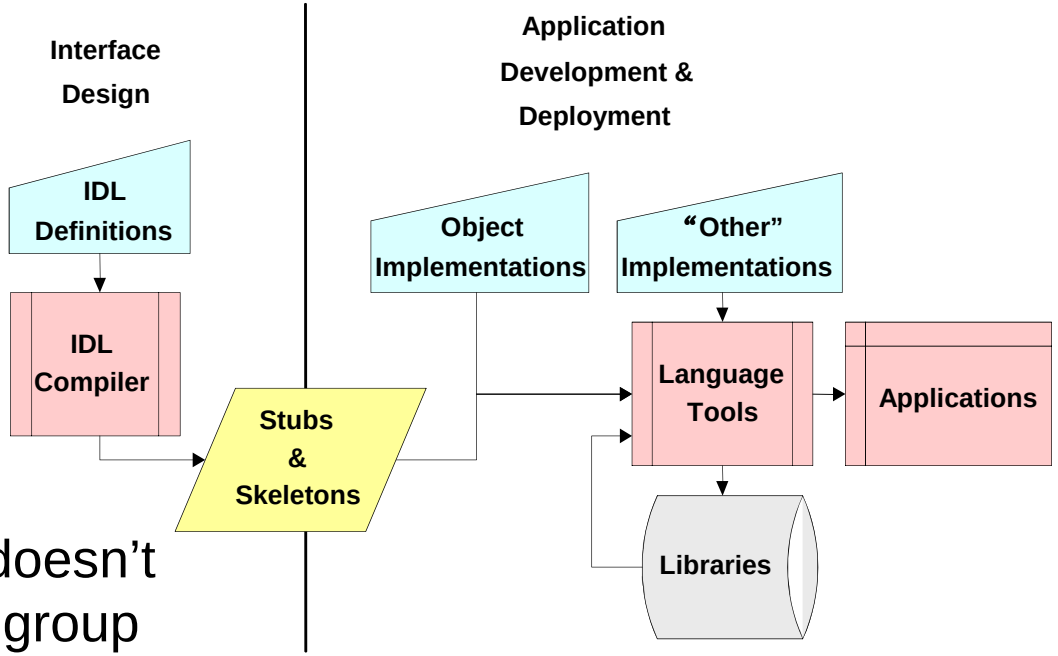
C++

```
class Foo : public
virtual ::CORBA::Object
{
  virtual void bar (CORBA::Long arg);
};
```

  - Operations defined in interfaces can be invoked on local or remote objects

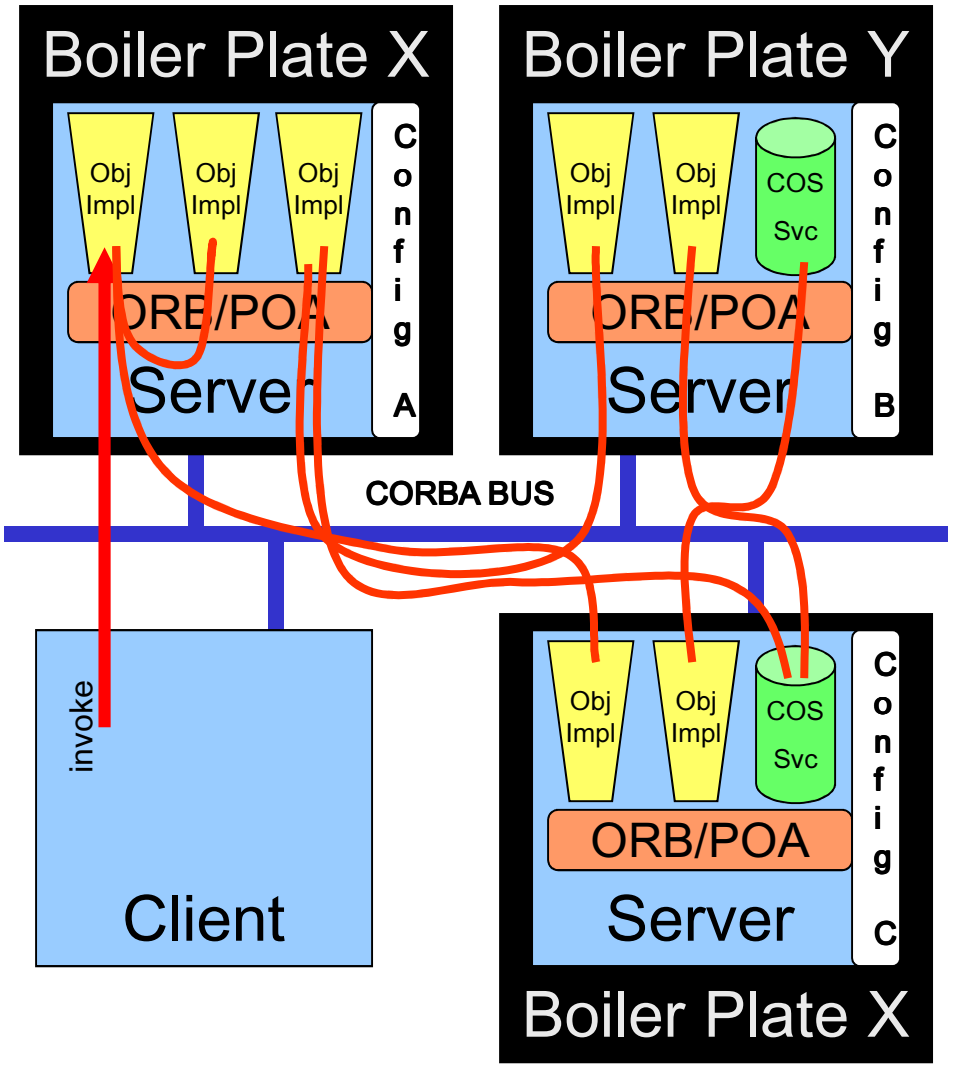
Drawbacks of DOC-based CORBA 2.x Middleware

CORBA 2.x application development is unnecessarily tedious & error-prone



- CORBA 2.x IDL doesn't provide a way to group together related interfaces to offer a service family
  - Such "bundling" must be done by developers via CORBA idioms & patterns
- CORBA 2.x doesn't specify how configuration & deployment of objects should be done to create complete applications
  - Proprietary infrastructure & scripts are written by developers to enable this

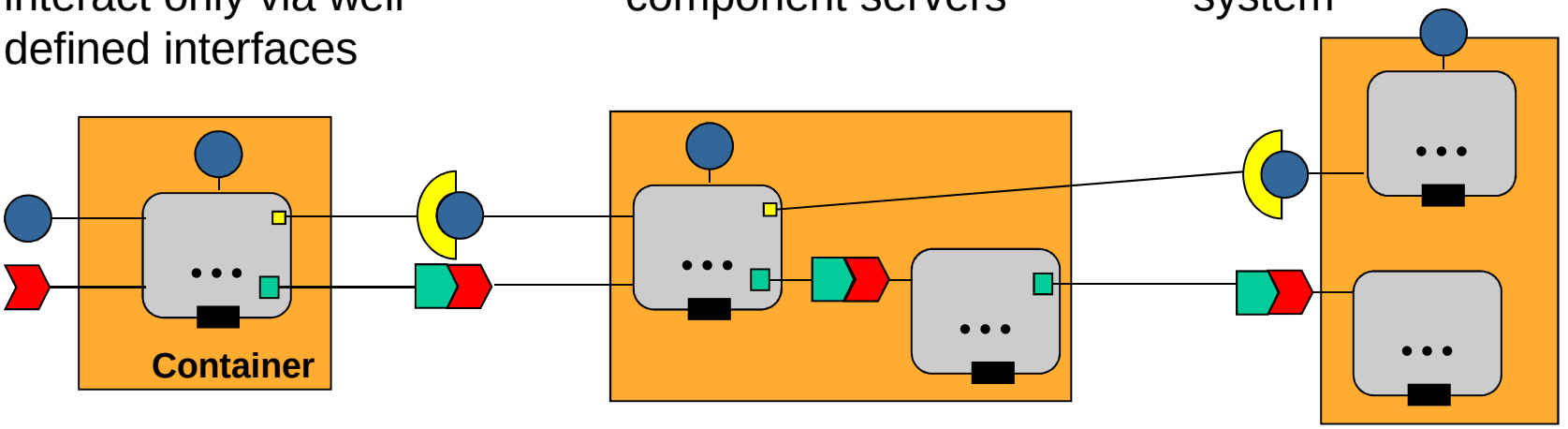
Example: Limitations of CORBA 2.x Specification



- Requirements of non-trivial DRE systems:
  - Collaboration of multiple objects & services
  - Deployment on diverse platforms
- CORBA 2.x limitations – lack of ***standards*** for
  - Server/node configuration
  - Object/service configuration
  - Application assembly
  - Object/service deployment
- Consequences:
  - Brittle, non-scalable implementation
  - Hard to adapt & maintain
  - Increased time-to-market

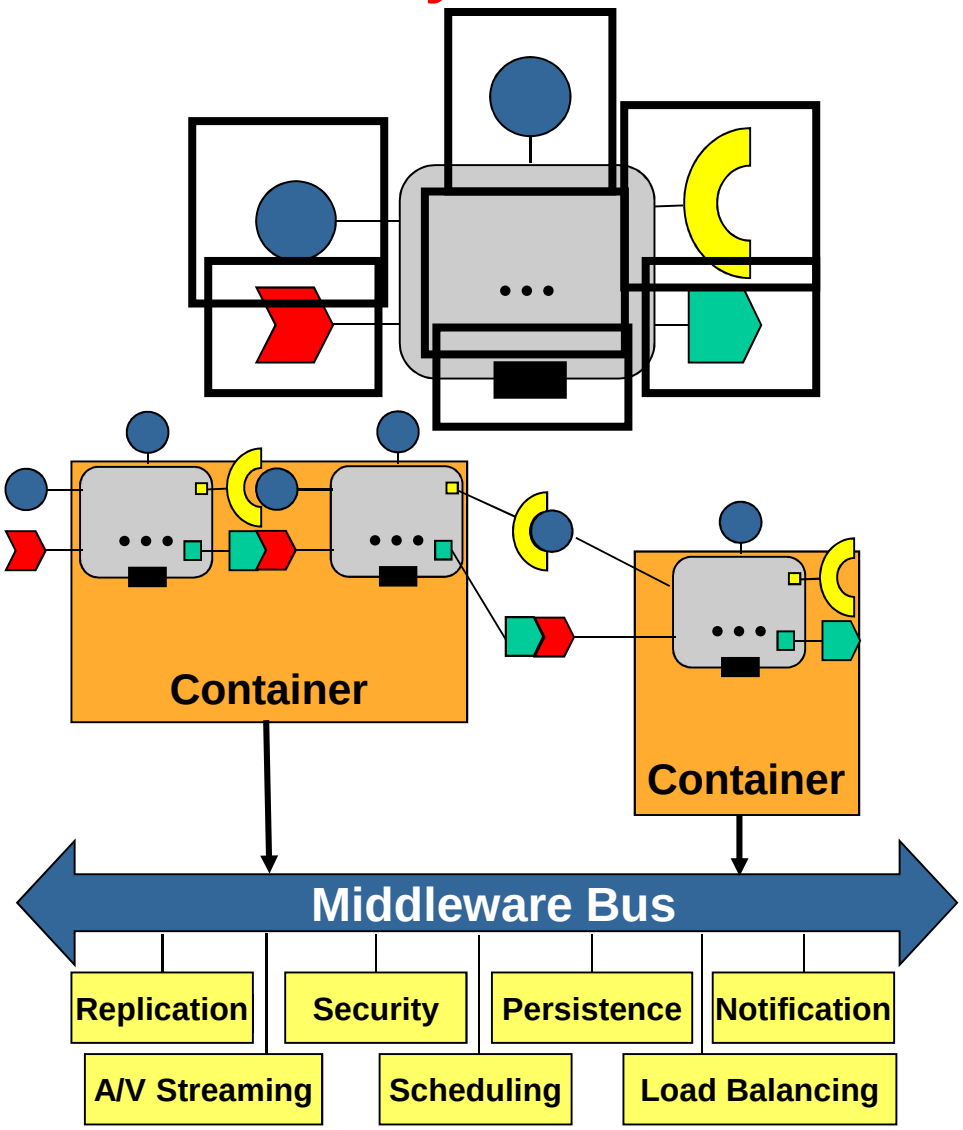
Solution: Component Middleware

- Creates a standard "virtual boundary" around application **component** implementations that interact only via well-defined interfaces
- Define standard **container** mechanisms needed to execute components in generic component servers
- Specify the infrastructure needed to **configure & deploy** components throughout a distributed system




```
<ComponentAssemblyDescription id="a_HUDDisplay"> ...
 <connection>
   <name>GPS-RateGen</name>
   <internalEndPoint><portName>Refresh</portName><instance>a_GPS</instance></internalEndPoint>
   <internalEndPoint><portName>Pulse</portName><instance>a_RateGen</instance></internalEndPoint>
  </connection>
  <connection>
    <name>NavDisplay-GPS</name>
    <internalEndPoint><portName>Refresh</portName><instance>a_NavDisplay</instance></internalEndPoint>
    <internalEndPoint><portName>Ready</portName><instance>a_GPS</instance></internalEndPoint>
  </connection> ...
</ComponentAssemblyDescription>
```

Birdseye View of Component Middleware

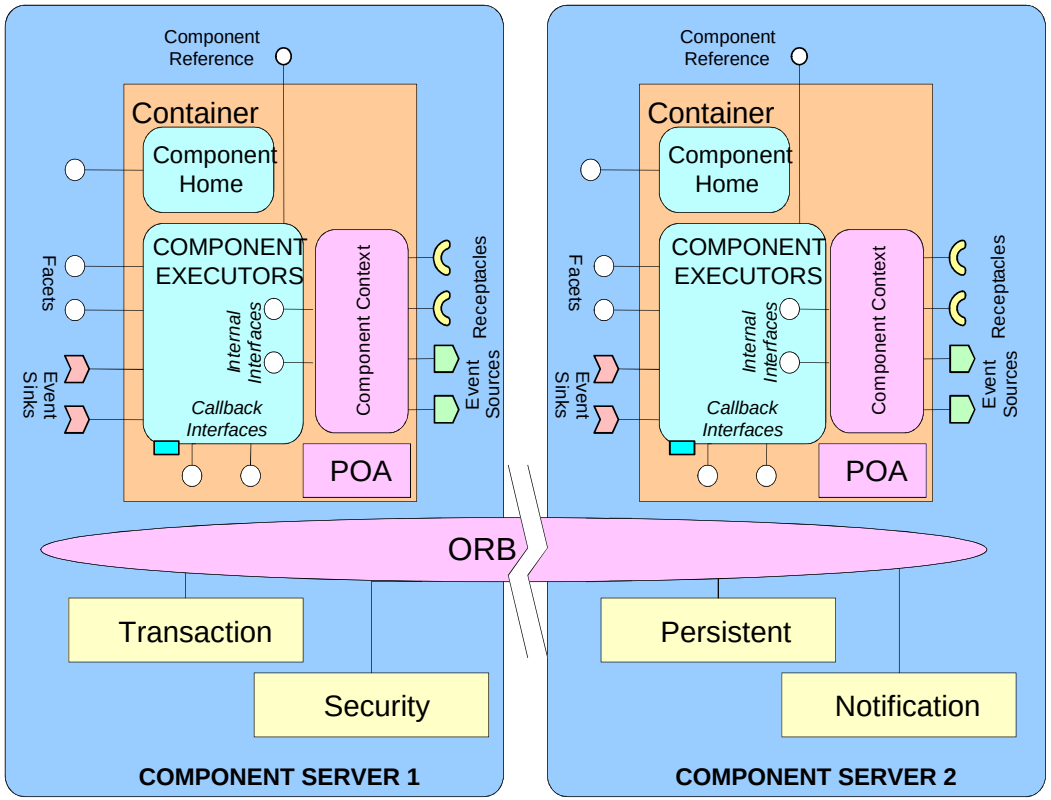


- *Components* encapsulate application "business" logic
- Components interact via *ports*
  - *Provided interfaces*, e.g.,facets
  - *Required connection points*, e.g., receptacles
  - *Event sinks & sources*
  - *Attributes*
- *Containers* provide execution environment for components with common operating requirements
- Components/containers can also
  - Communicate via a *middleware bus* &
  - Reuse *common middleware services*

Component middleware defines interfaces, policies, & *some* implementations

Overview of the CORBA Component Model (CCM)

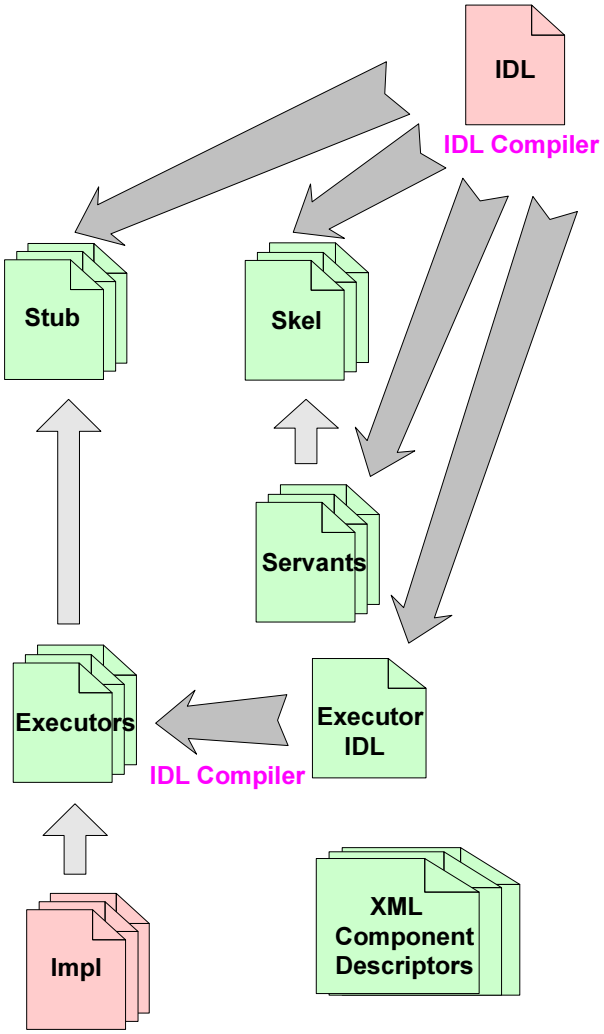
Capabilities of CORBA Component Model (CCM)



- Containers define operations that enable component executors to access common middleware services & runtime policies

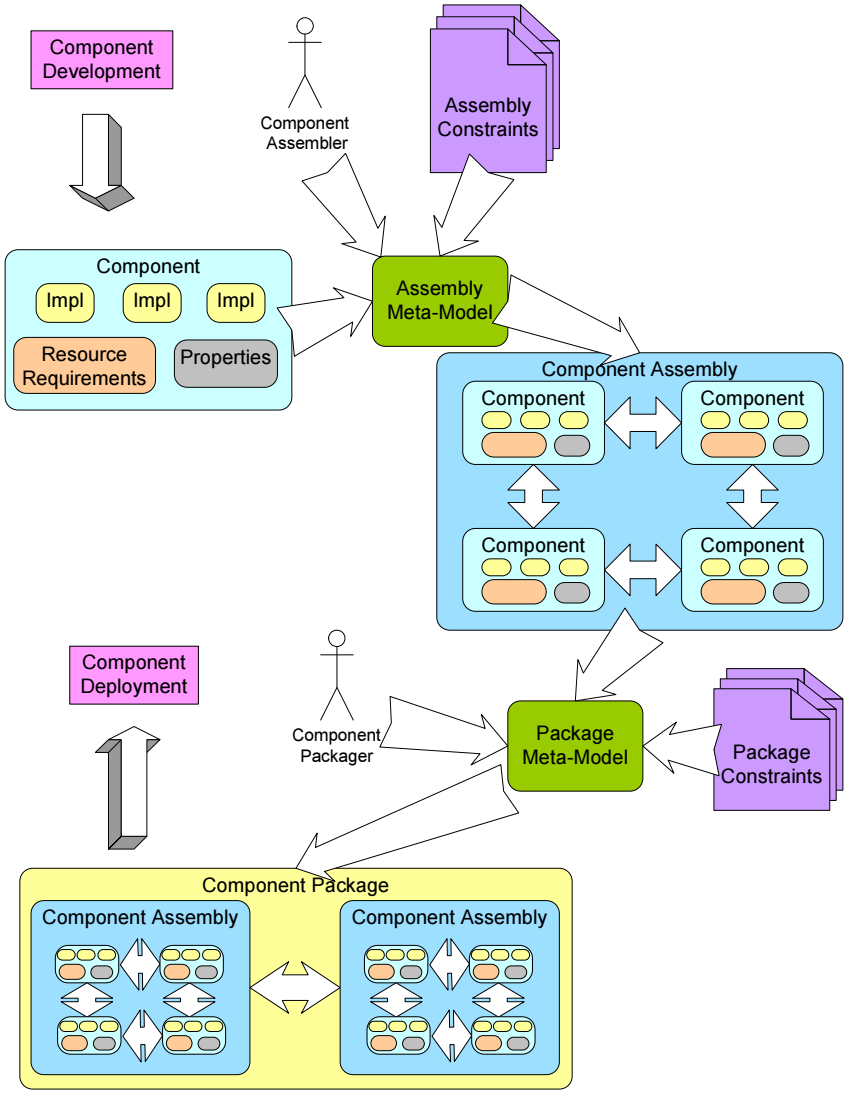
- **Component Server**
  - A generic server process for hosting containers & component/home executors
- **Component Implementation Framework (CIF)**
  - Automates the implementation of many component features
- **Component packaging tools**
  - Compose implementation & configuration information into deployable assemblies
- **Component deployment tools**
  - Automate the deployment of component assemblies to component servers

Capabilities of CORBA Component Model (CCM)



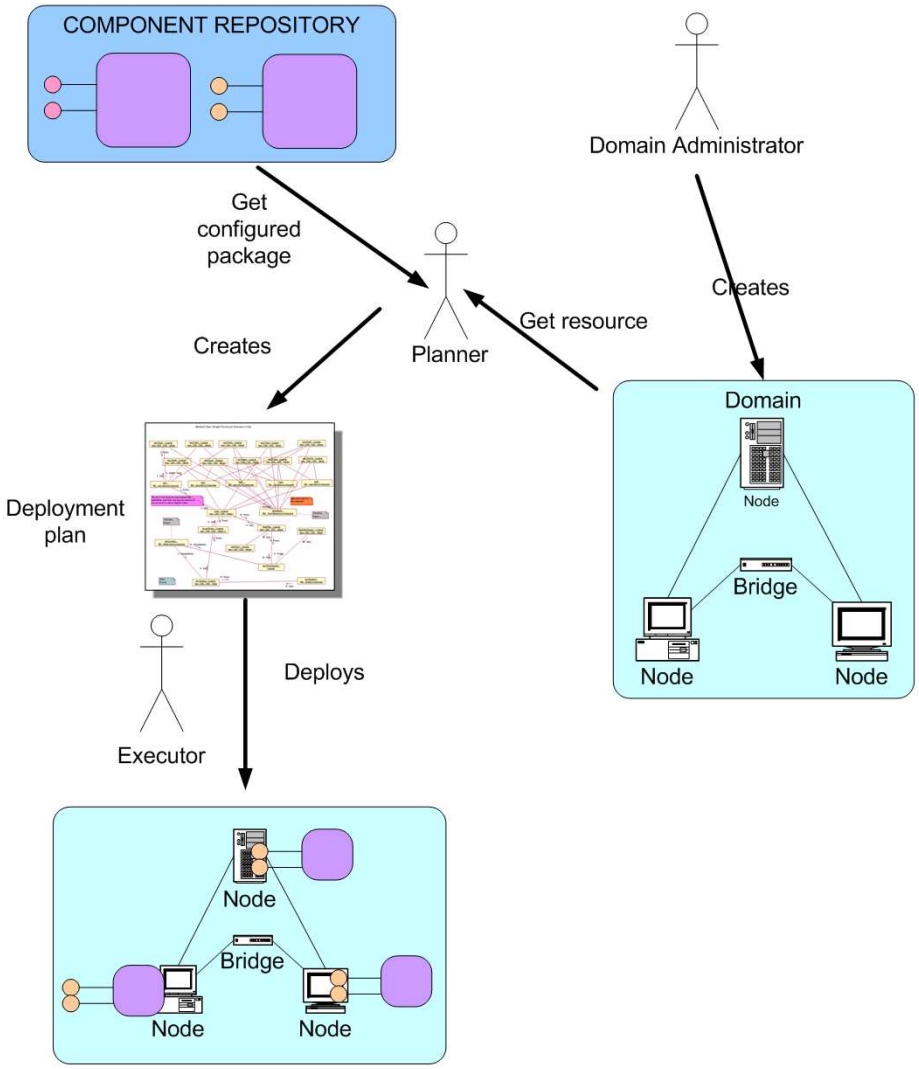
- **Component Server**
  - A generic server process for hosting containers & component/home executors
- **Component Implementation Framework (CIF)**
  - Automates the implementation of many component features
- **Component packaging tools**
  - Compose implementation & configuration information into deployable assemblies
- **Component deployment tools**
  - Automate the deployment of component assemblies to component servers

Capabilities of CORBA Component Model (CCM)



- **Component Server**
  - A generic server process for hosting containers & component/home executors
- **Component Implementation Framework (CIF)**
  - Automates the implementation of many component features
- **Component packaging tools**
  - Compose implementation & configuration information into deployable assemblies
- **Component deployment tools**
  - Automate the deployment of component assemblies to component servers

Capabilities of CORBA Component Model (CCM)



- **Component Server**
  - A generic server process for hosting containers & component/home executors
- **Component Implementation Framework (CIF)**
  - Automates the implementation of many component features
- **Component packaging tools**
  - Compose implementation & configuration information into deployable assemblies
- **Component deployment tools**
  - Automate the deployment of component assemblies to component servers

Available CCM Implementations

Name	Provider	Open Source	Language	URL
Component Integrated ACE ORB (CIAO)	Vanderbilt University & Washington University	Yes	C++	www.dre.vanderbilt.edu/CIAO/
Enterprise Java CORBA Component Model (EJCCM)	Computational Physics, Inc.	Yes	Java	www.cpi.com/ejccm/
K2	iCMG	No	C++	www.icmgworld.com/products.asp
MicoCCM	FPX	Yes	C++	www.fpx.de/MicoCCM/
OpenCCM	ObjectWeb	Yes	Java	openccm.ow2.org/
QoS Enabled Distributed Object (Qedo)	Fokus	Yes	C++	www.qedo.org
StarCCM	Source Forge	Yes	C++	sourceforge.net/projects/starccm/

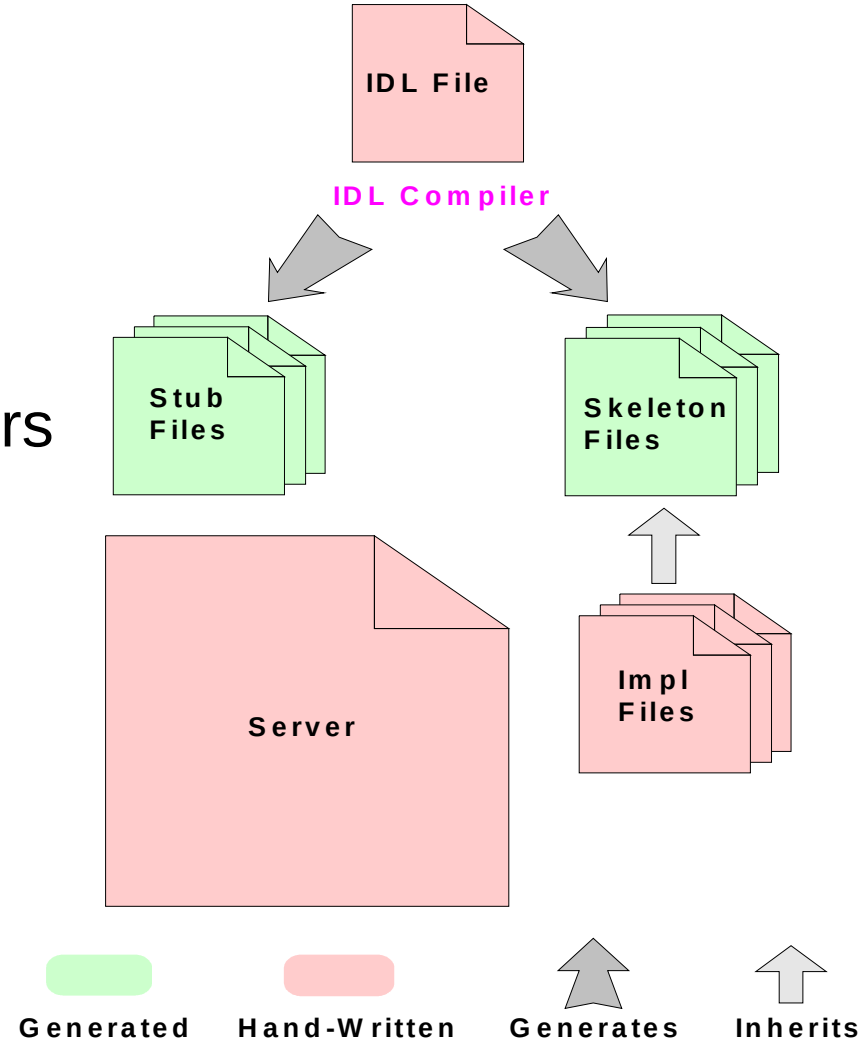
CCM Compared to EJB, COM, & .NET

- Like Sun Microsystems' Enterprise Java Beans (EJB)
  - CORBA components created & managed by homes
  - Run in containers that manage system services transparently
  - Hosted by generic application component servers
  - But can be written in more languages than Java
- Like Microsoft's Component Object Model (COM)
  - Have several input & output interfaces per component
  - Both point-to-point sync/async operations & publish/subscribe events
  - Component navigation & introspection capabilities
  - But has more effective support for distribution & QoS properties
- Like Microsoft's .NET Framework
  - Could be written in different programming languages
  - Could be packaged to be distributed
  - But runs on more platforms than just Microsoft Windows

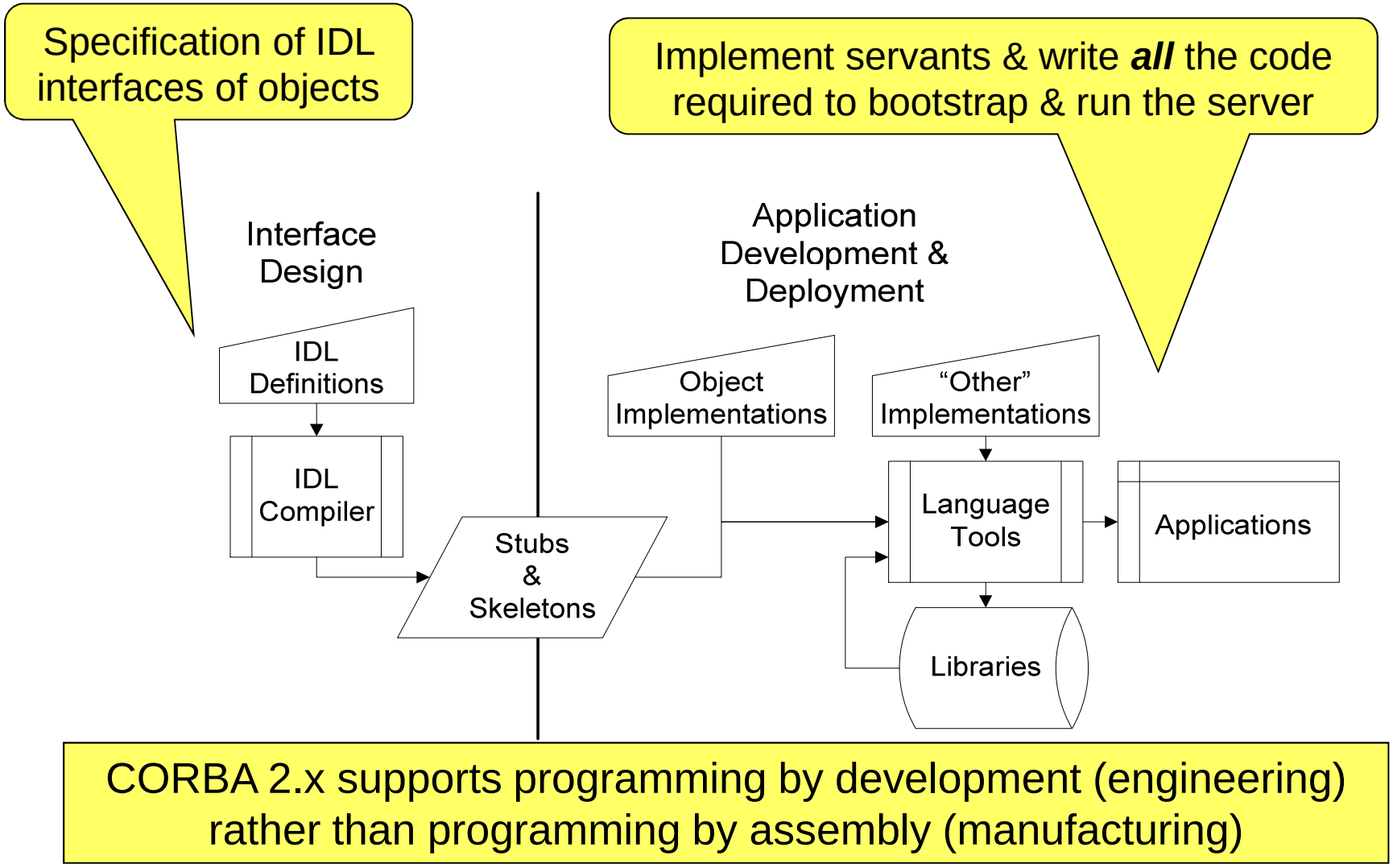
Comparing Application Development with CORBA 2.x vs. CCM

CORBA 2.x User Roles

- Object interface designers
- Server developers
- Client application developers



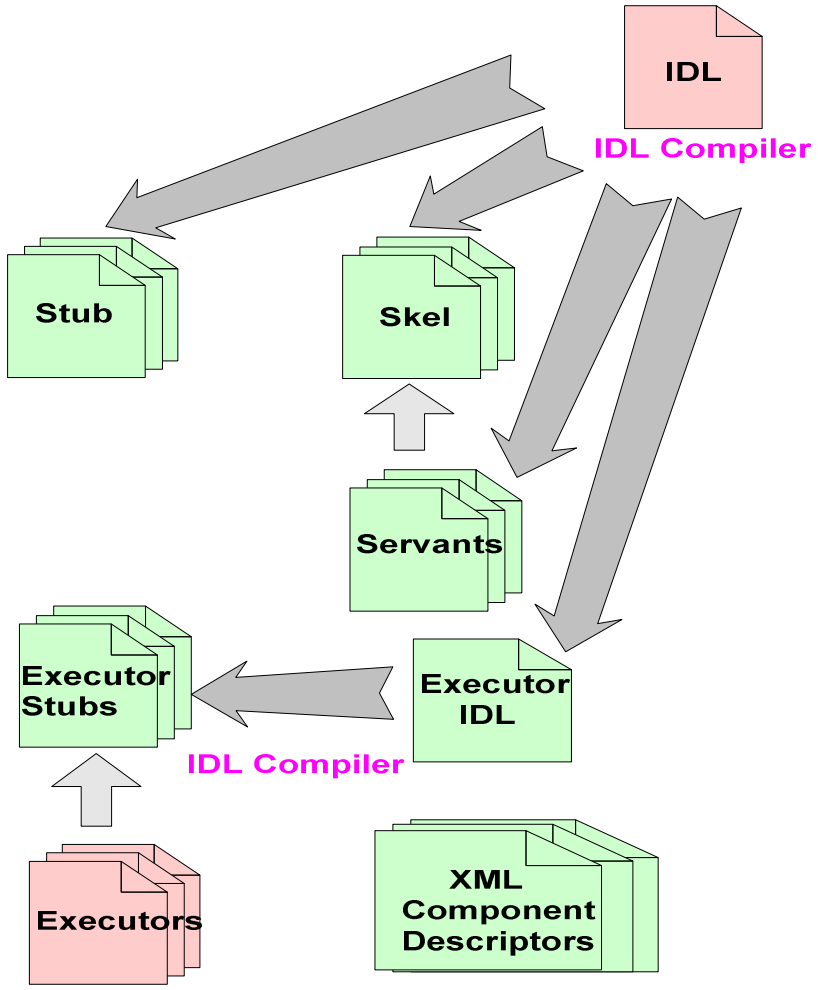
CORBA 2.x Application Development Lifecycle



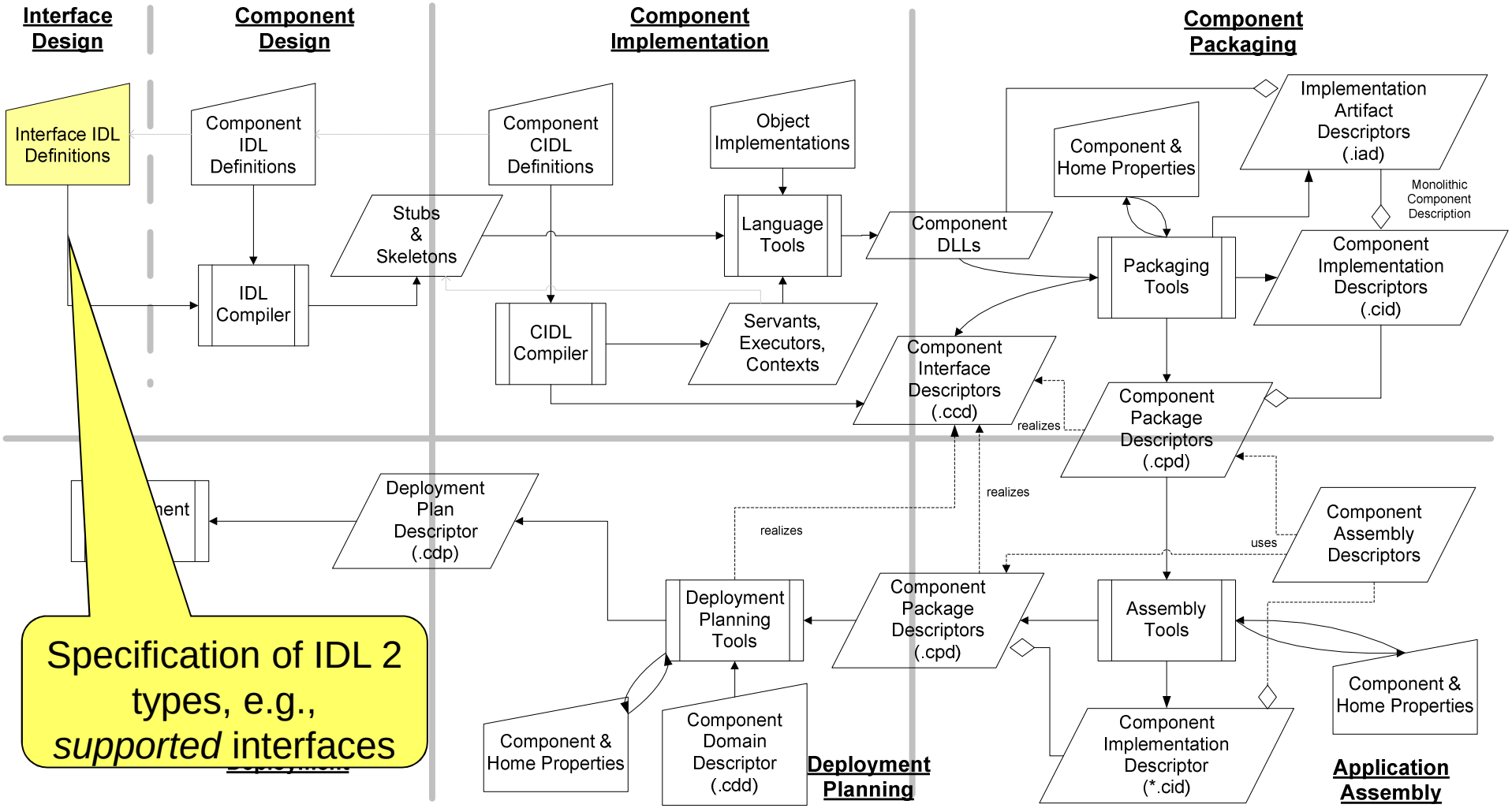
CORBA 2.x supports programming by development (engineering) rather than programming by assembly (manufacturing)

CCM User Roles

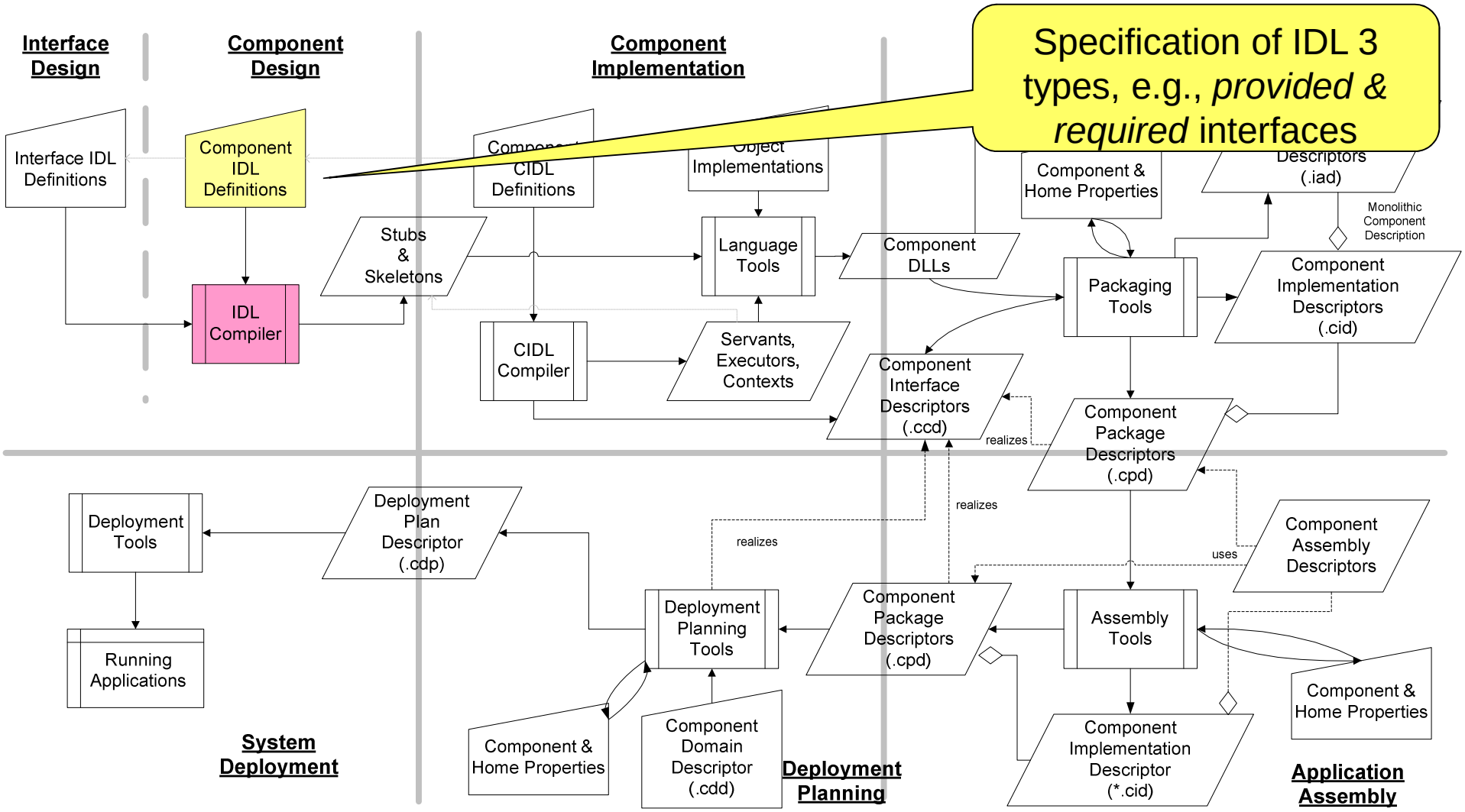
- Component designers
- Component clients
- Composition designers
- Component implementers
- Component packagers
- Component deployers
- Component end-users



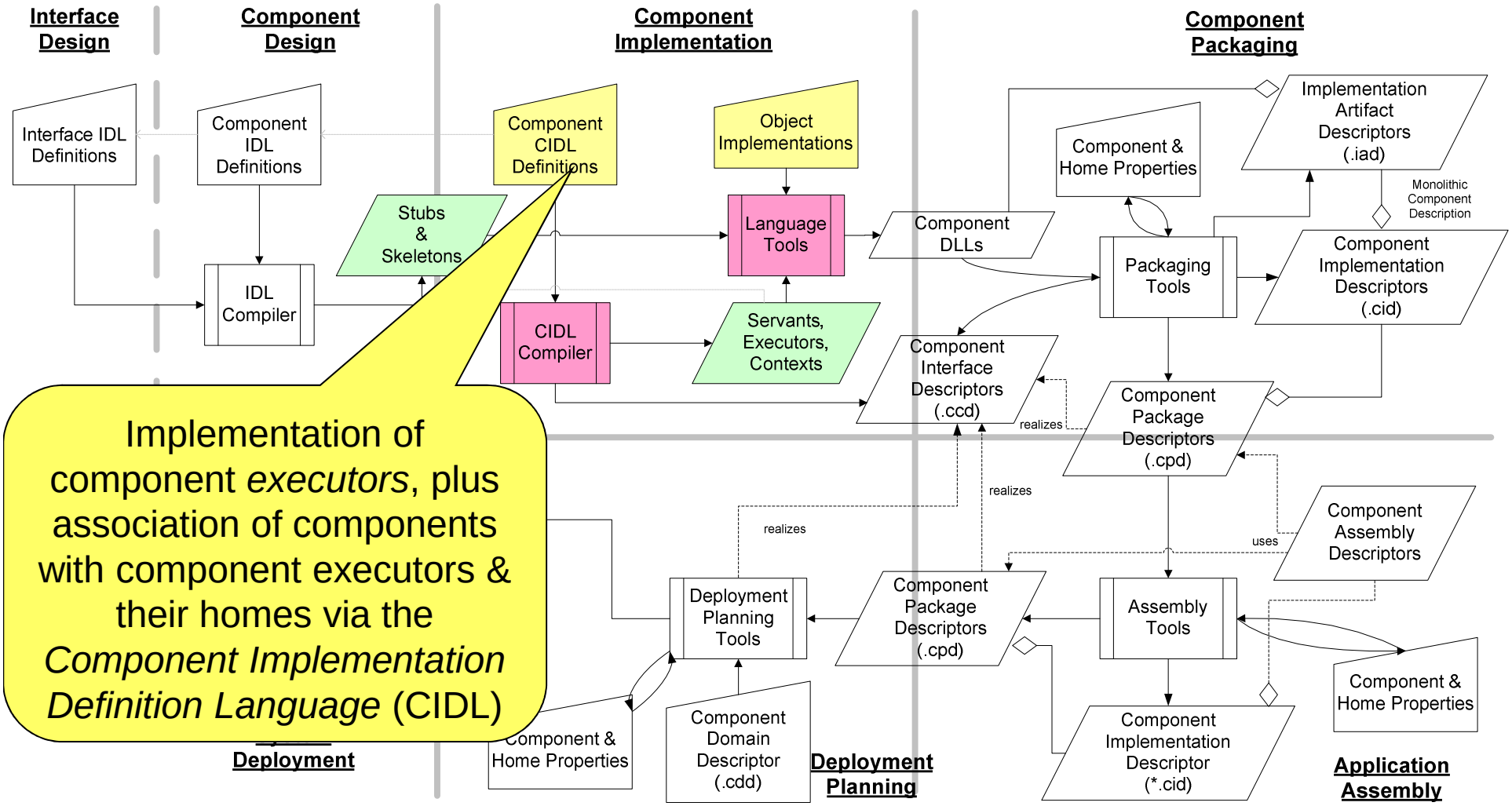
CCM Application Development Lifecycle



CCM Application Development Lifecycle

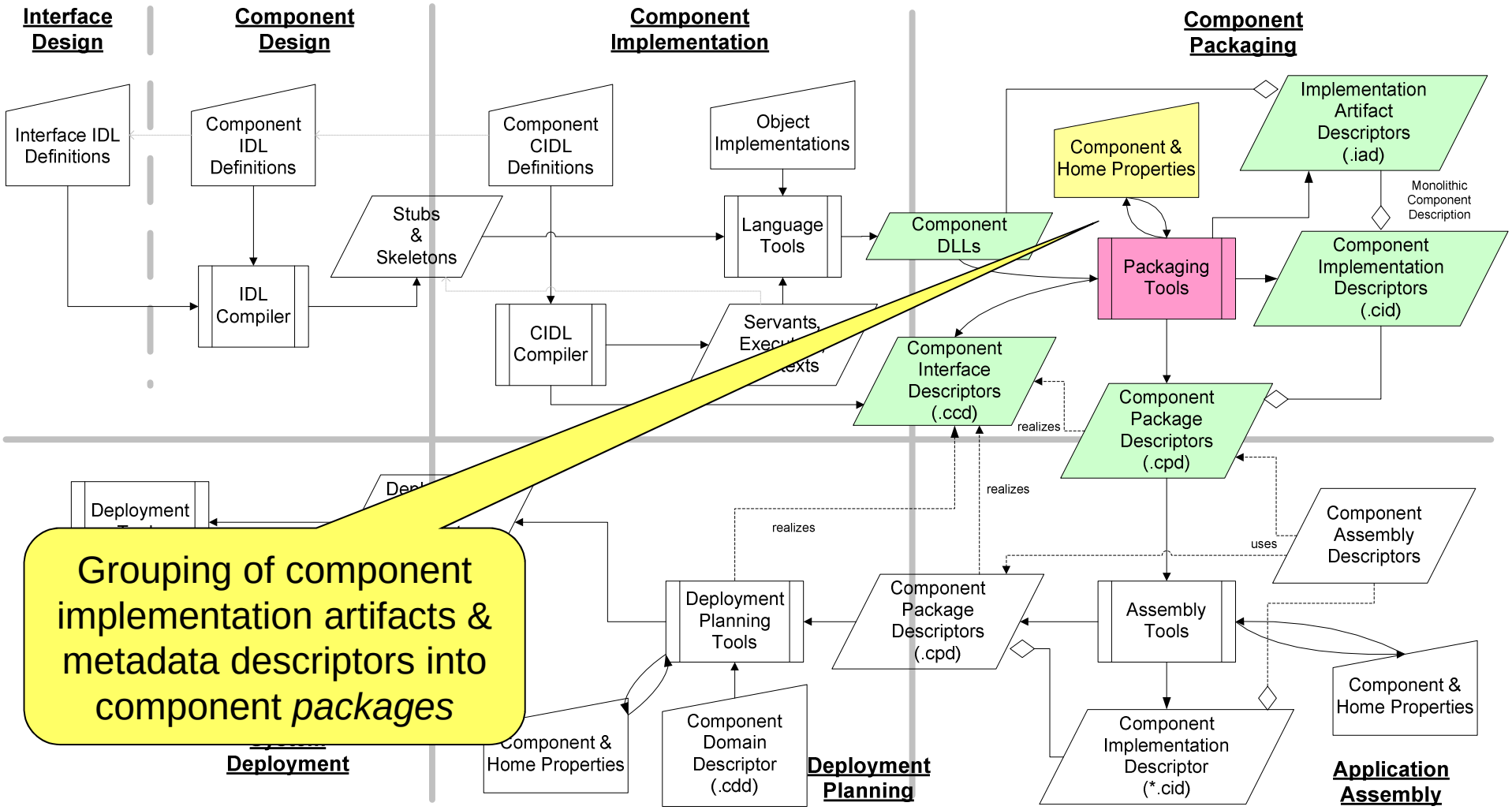


CCM Application Development Lifecycle

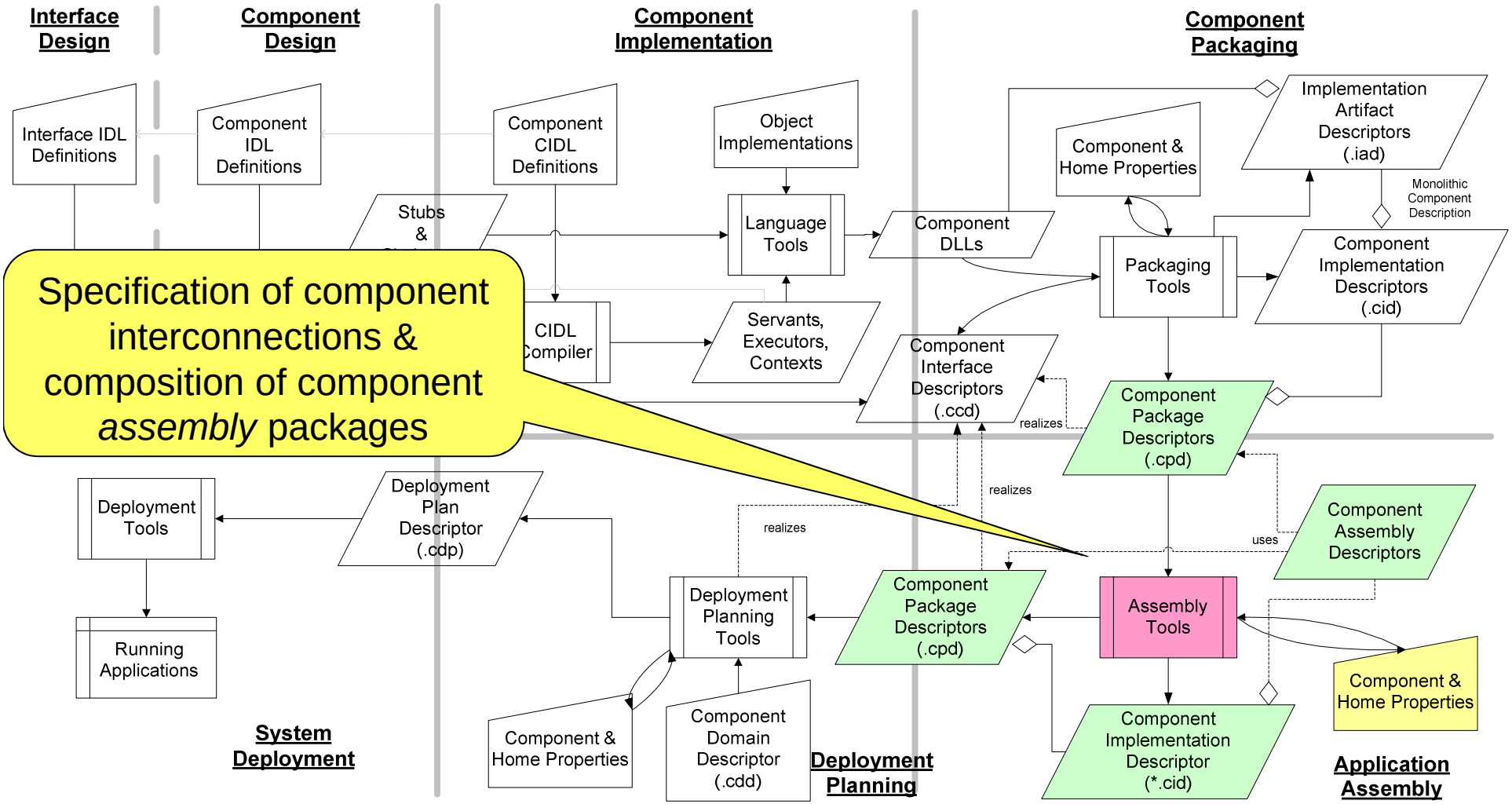


Implementation of component *executors*, plus association of components with component executors & their homes via the *Component Implementation Definition Language* (CIDL)

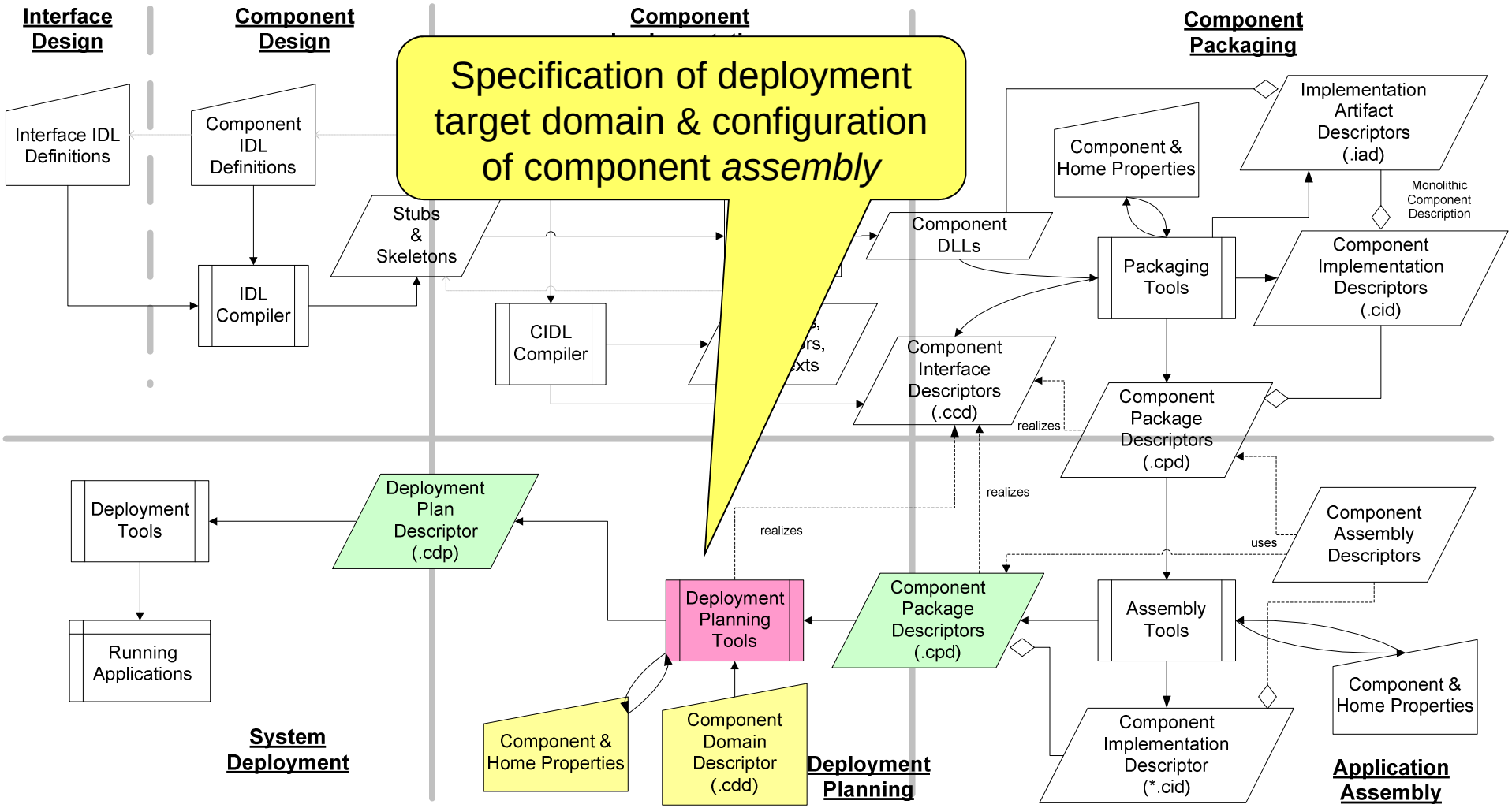
CCM Application Development Lifecycle



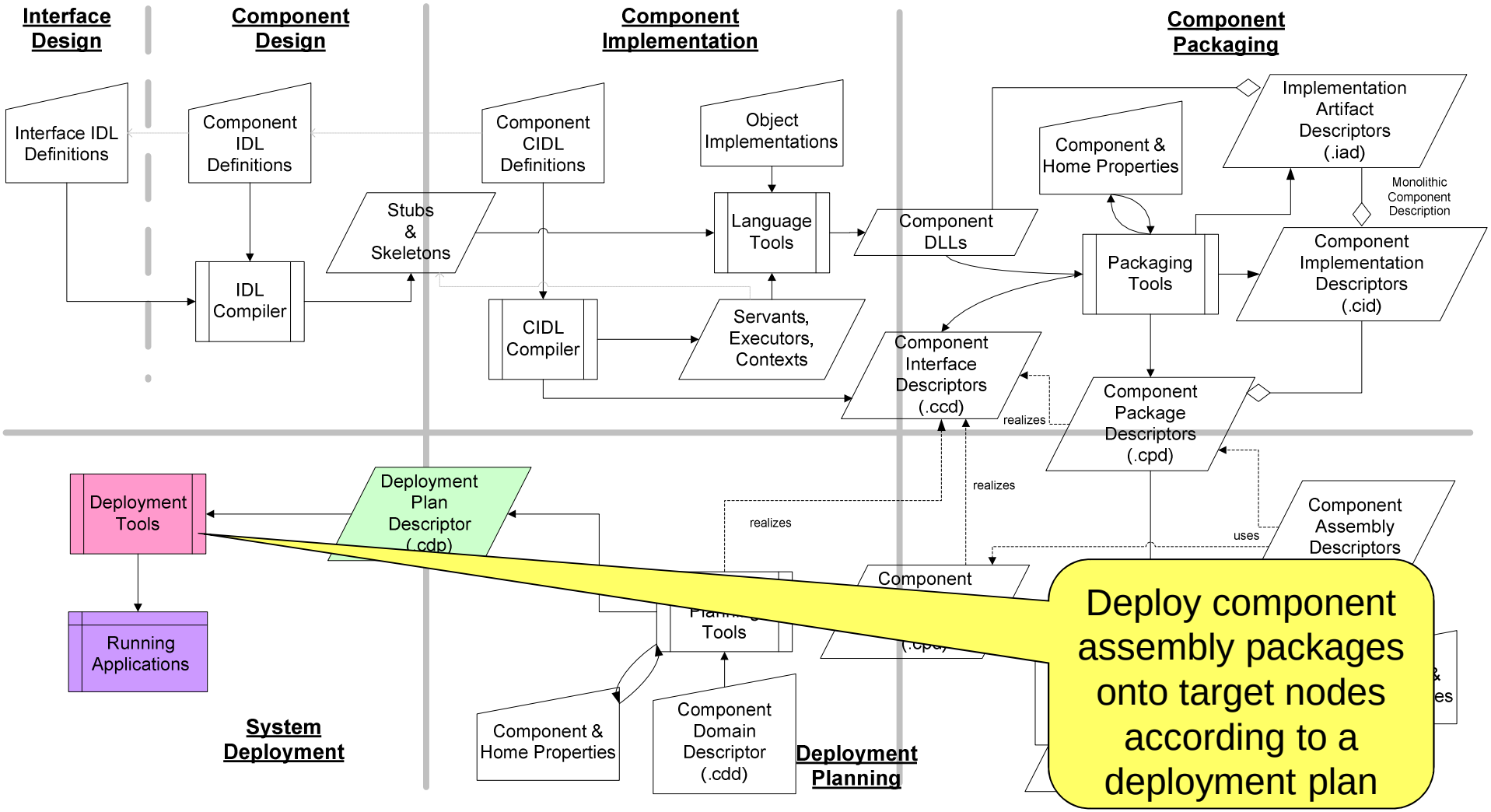
CCM Application Development Lifecycle



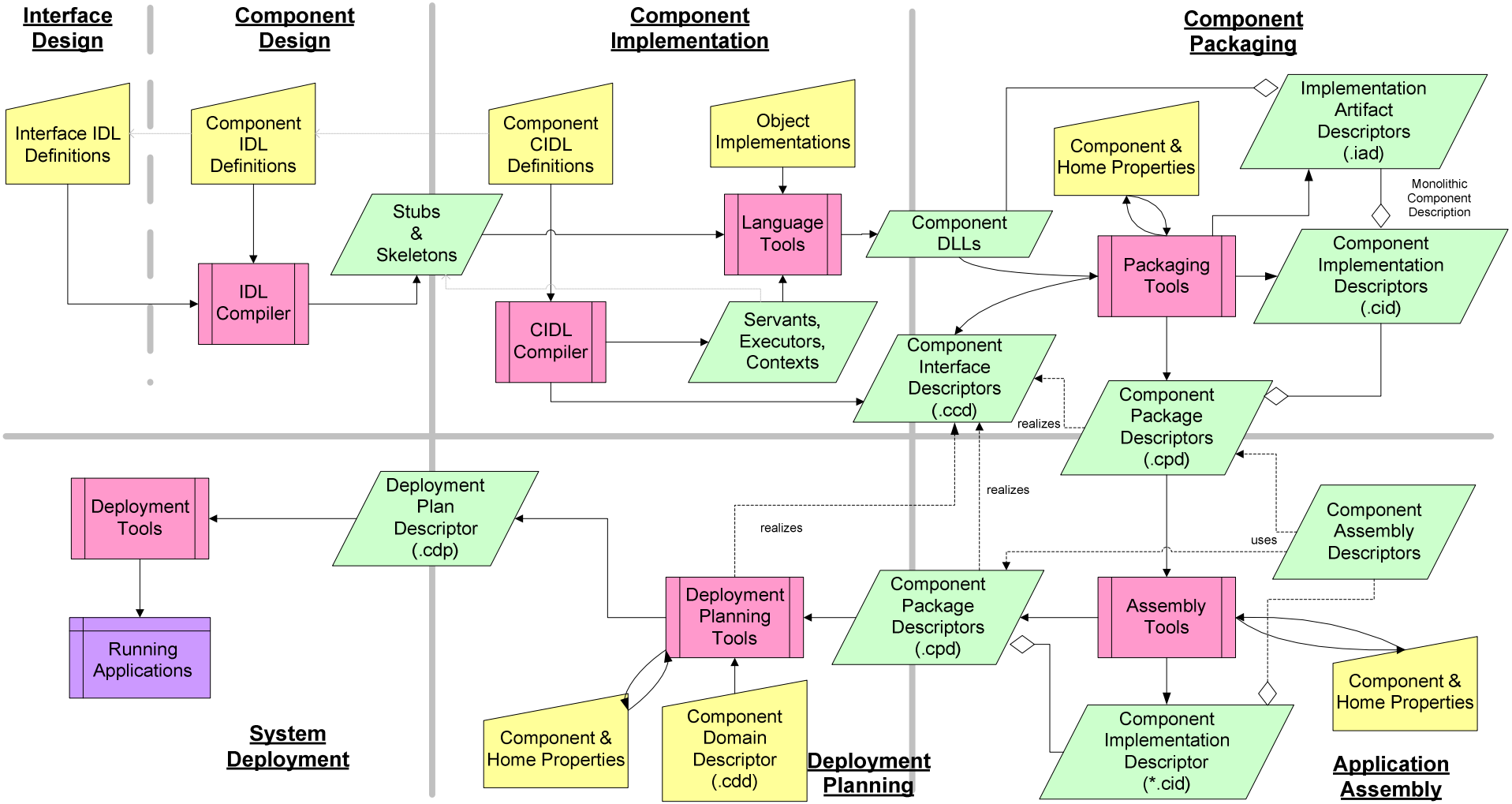
CCM Application Development Lifecycle



CCM Application Development Lifecycle



CCM Application Development Lifecycle



CCM makes *explicit* steps performed *implicitly* in CORBA 2.x

CORBA Component Model (CCM) Features

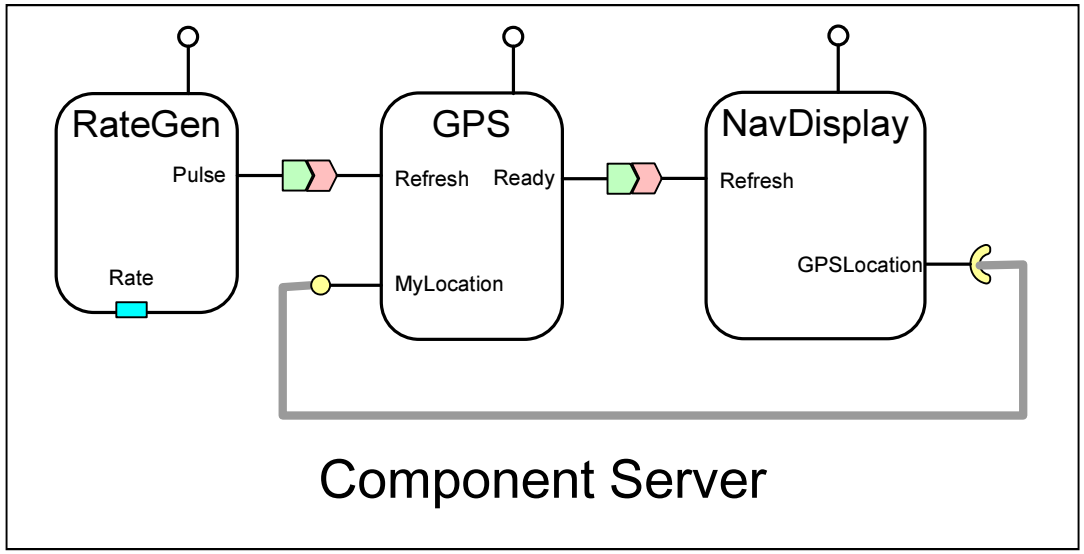
Example CCM DRE Application

Avionics example used throughout tutorial as typical DRE application

Rate Generator

Positioning Sensor

Display Device

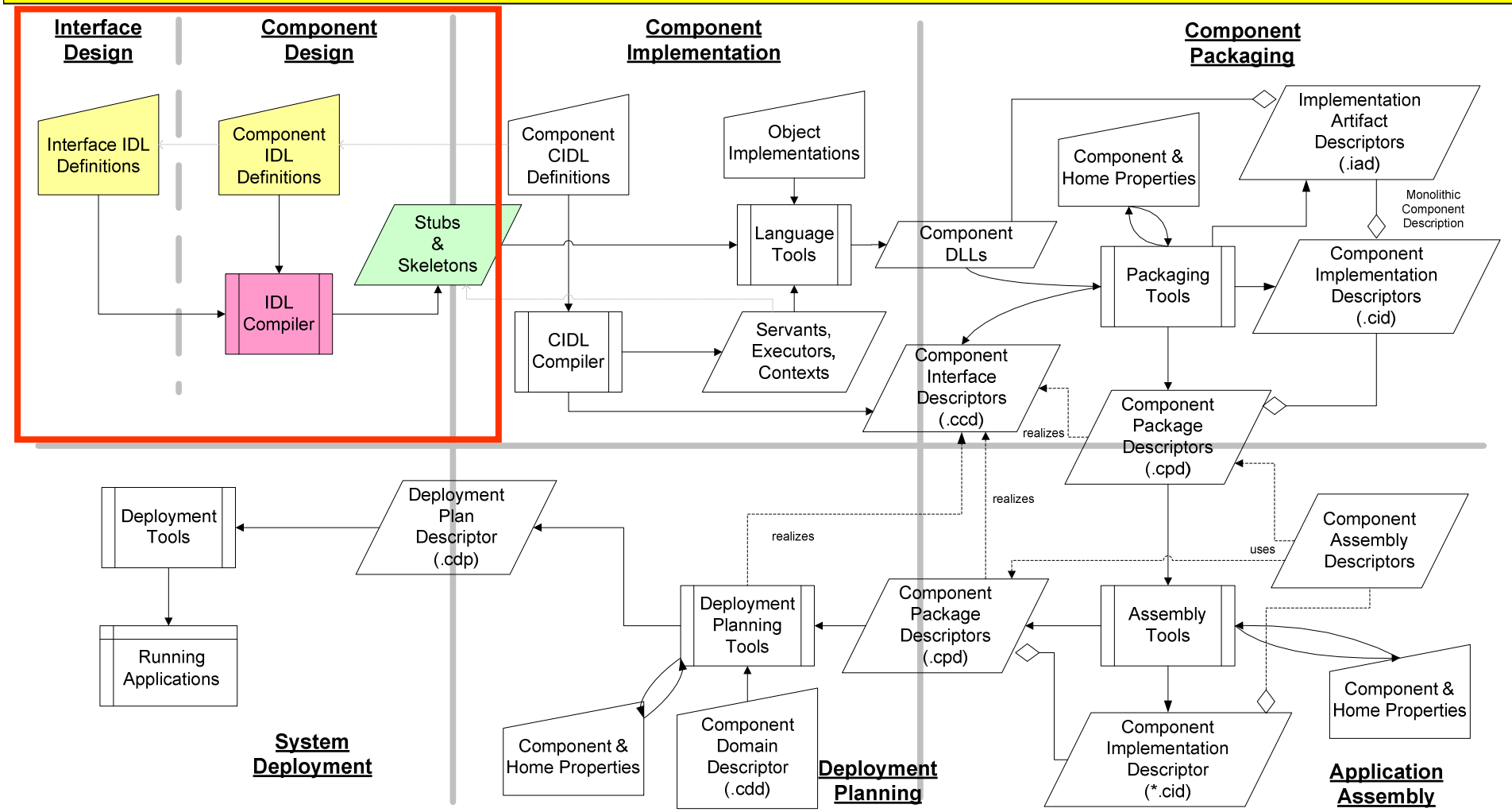


`$CIAO_ROOT/examples/Display/`

- ***Rate Generator***
  - Sends periodic **`Pulse`** events to consumers
- ***Positioning Sensor***
  - Receives **`Refresh`** events from suppliers
  - Refreshes cached coordinates available thru **`MyLocation`** facet
  - Notifies subscribers via **`Ready`** events
- ***Display Device***
  - Receives **`Refresh`** events from suppliers
  - Reads current coordinates via its **`GPSLocation`** receptacle
  - Updates display

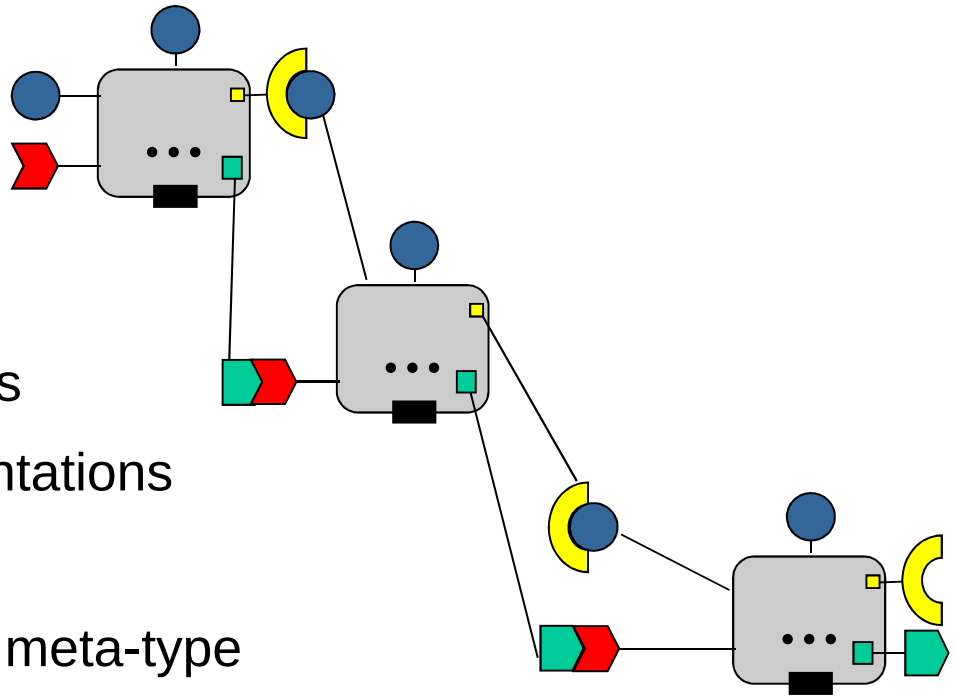
Interface & Component Design Stage

Goal: Specify *supported*, *provided*, & *required* interfaces & event *sinks/sources*



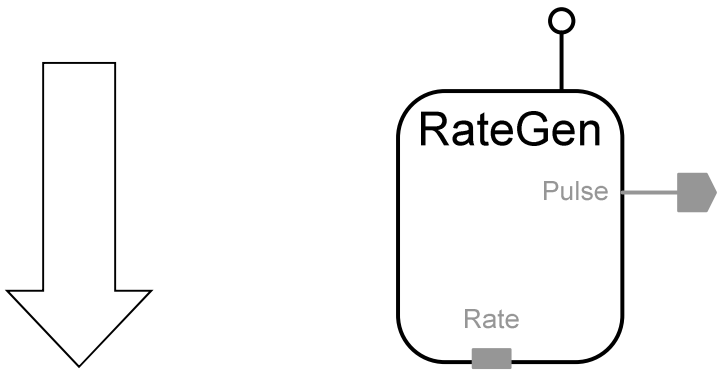
Unit of Business Logic & Composition in CCM

- Context
  - Development via *composition*
- Problems
  - CORBA 2.x object limitations
    - Objects just identify interfaces
    - No direct relation w/implementations
- CCM Solution
  - Define CORBA 3.0 **`component`** meta-type
    - Extension of CORBA 2.x **`Object`** interface
    - Has interface & object reference
    - Essentially a stylized use of CORBA interfaces/objects
      - i.e., CORBA 3.x IDL maps onto equivalent CORBA 2.x IDL



Simple CCM Component Example

```
// IDL 3
interface rate_control
{
  void start ();
  void stop ();
};
component RateGen
  supports rate_control {};
```




```
// Equivalent IDL 2
interface RateGen :
  ::Components::CCMObject,
  rate_control {};
```

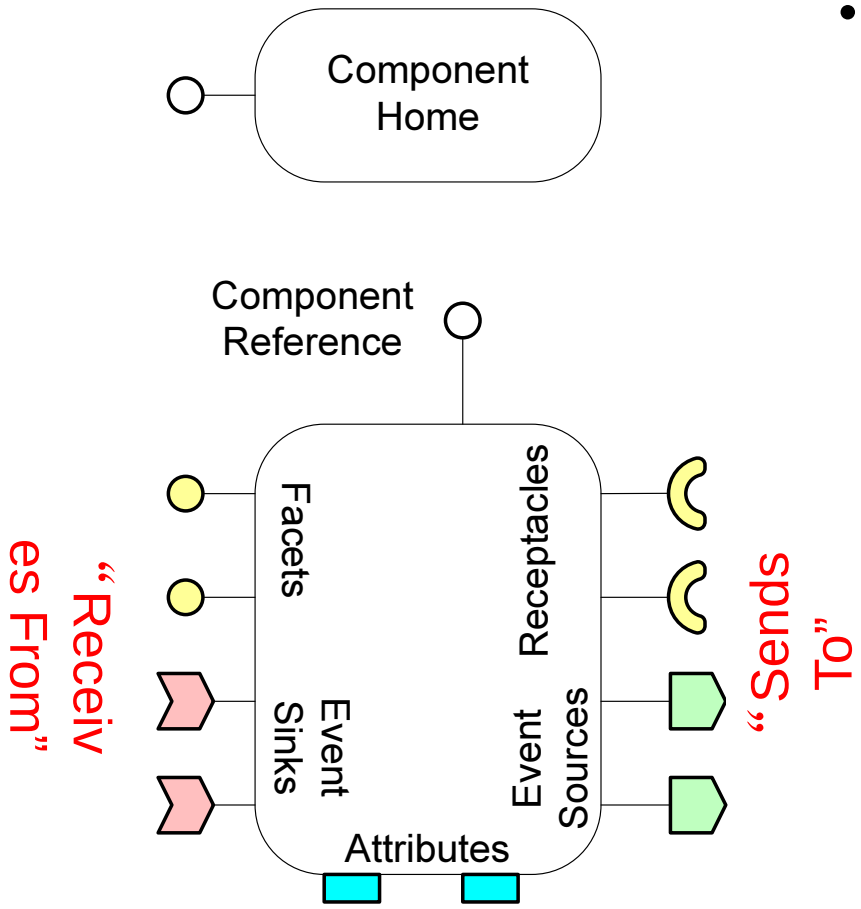
- Roles played by CCM **`component`**
  - Define a unit of composition, reuse, & implementation
  - Encapsulate an interaction & configuration model
- A CORBA **`component`** has several derivation options, i.e.,
  - It can *inherit* from a single component type

```
component E : D {};
```

  - It can *support* multiple IDL interfaces

```
interface A {};
interface B {};
component D supports A, B {};
```

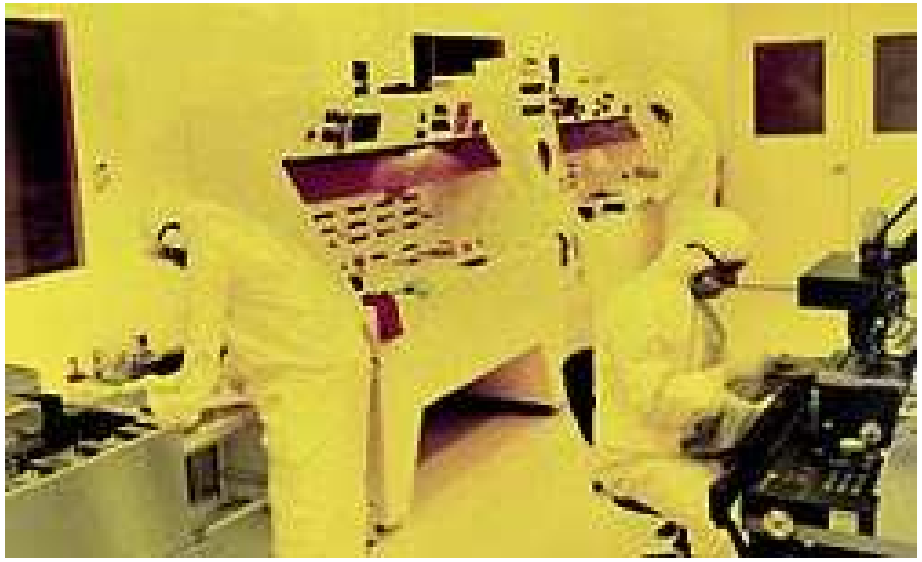
CORBA Component Ports



- A CORBA component can contain *ports*:
  - *Facets* (**`provides`**)
    - Offers operation interfaces
  - *Receptacles* (**`uses`**)
    - Required operation interfaces
  - *Event sources* (**`publishes`** & **`emits`**)
    - Produced events
  - *Event sinks* (**`consumes`**)
    - Consumed events
  - *Attributes* (**`attribute`**)
    - Configurable properties
- Each component instance is created & managed by a unique component **home**

Managing Component Lifecycle

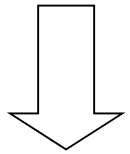
- Context
  - Components need to be created by the CCM run-time
- Problems with CORBA 2.x
  - No standard way to manage component's lifecycle
  - Need standard mechanisms to strategize lifecycle management

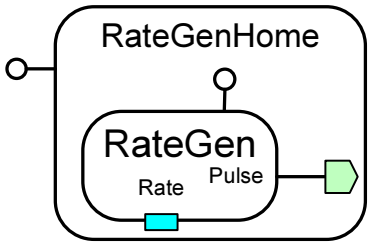


- CCM Solution
  - Integrate lifecycle service into component definitions
  - Use different component *home's* to provide different lifecycle managing strategies
    - Based on Factory & Finder patterns

A CORBA Component Home

```
// IDL 3
home RateGenHome manages RateGen
{
  factory create_pulser
    (in rateHz r);
};
```






```
// Equivalent IDL 2
interface RateGenHomeExplicit
: Components::CCMHome {
 RateGen create_pulser
  (in rateHz r);
};
interface RateGenHomeImplicit
: Components::KeylessCCMHome {
  RateGen create ();
};
interface RateGenHome :
 RateGenHomeExplicit,
 RateGenHomeImplicit {};
```

- **home** is new CORBA meta-type
  - A **home** has an interface & object reference
- Manages one type of component
  - More than one home type can manage same component type
  - However, a component instance is managed by one home instance
- Standard *factory* & *finder* operations
  - e.g., **`create()`**
- **home** can have user-defined operations

A Quick CCM Client Example

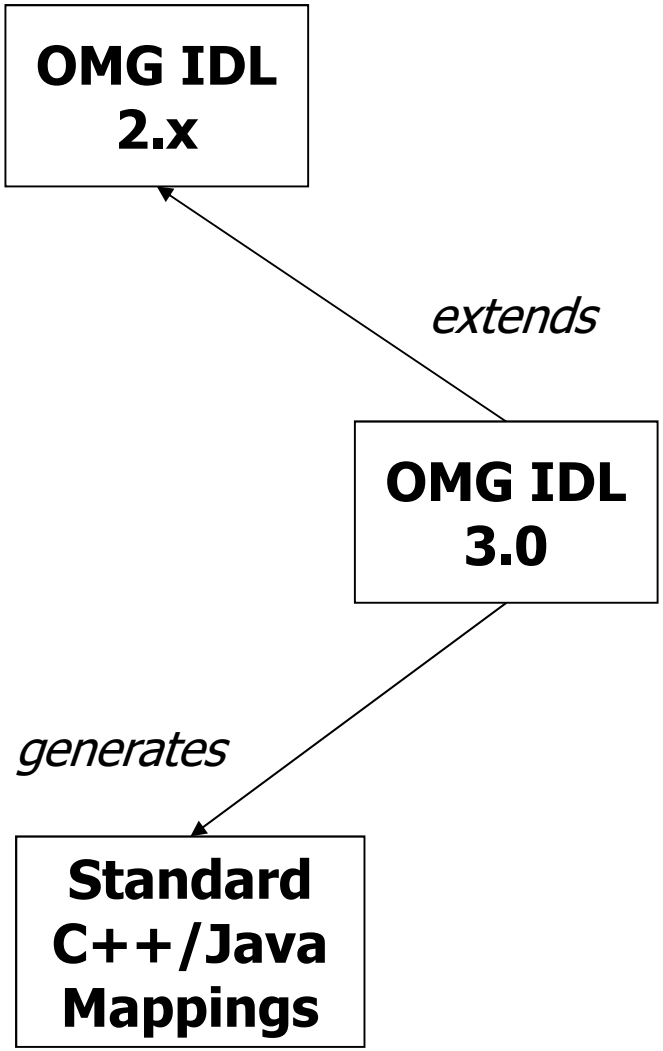
Component & Home for Simple **HelloWorld**

```
interface Hello {
  void sayHello (in string username);
};
interface Goodbye {
  void sayGoodbye (in string username);
};
component HelloWorld supports Hello {
      provides Goodbye Farewell;
};
home HelloHome manages HelloWorld {};
```

- IDL 3 definitions for
  - Component: **HelloWorld**
  - Managing home: **HelloHome**

The Client OMG IDL Mapping

- As we've seen, each OMG IDL 3.0 construction has an equivalent in terms of OMG IDL 2.x
- Component & home types are viewed by clients through the CCM client-side OMG IDL mapping
- This mapping requires no change in CORBA's client programming language mapping
  - i.e., clients still use their favorite IDL-oriented tools, such as CORBA stub generators, etc.
- Clients need not be "component-aware"
  - i.e., they can just invoke interface operations



Simple Client for **HelloWorld** Component

```
 1 int
 2 main (int argc, char *argv[])
 3 {
 4   CORBA::ORB_var orb =
 5     CORBA::ORB_init (argc, argv);
 6   CORBA::Object_var o =
 7     orb->resolve_initial_references
 8             ("NameService");
 9   CosNaming::NamingContextExt_var nc =
10     CosNaming::NamingContextExt::_narrow (o);
11   o = nc->resolve_str ("myHelloHome");
12   HelloHome_var hh = HelloHome::_narrow (o);
13   HelloWorld_var hw = hh->create ();
14   hw->sayHello ("Dennis & Brian");
15   hw->remove ();
16   return 0;
17 }
```

```
$ ./hello-client # Triggers this on the server:

Hello World!  -- from Dennis & Brian.
```

- Lines 4-10: Perform standard ORB bootstrapping
- Lines 11-12: Obtain object reference to home via Naming Service
- Line 13: Use home to create component
- Line 14: Invoke remote operation
- Line 15: Remove component instance
  - Clients don't always need to manage component lifecycle directly

CCM Component Features in Depth

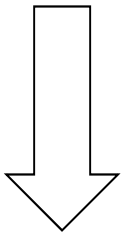
www.cs.wustl.edu/~schmidt/cuj-17.doc

Components Can Offer Different Views

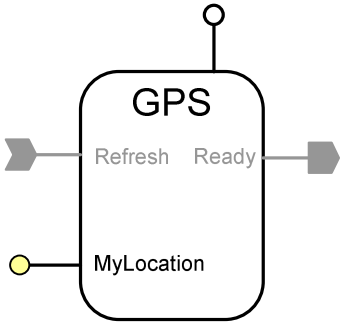
- Context
  - Components need to collaborate with other types of components
  - These collaborating components may understand different interfaces
- Problems with CORBA 2.x
  - Hard to extend interface without breaking/bloating it
  - No standard way to acquire new interfaces
- CCM Solution
  - Define facets, a.k.a. *provided* interfaces, that embody a view of the component & correspond to roles in which a client may act relatively to the component
    - Represents the “top of the Lego”

Component Facets

```
// IDL 3
interface position
{
  long get_pos ();
};
component GPS
{
  provides position MyLocation;
  …
};
```



```
// Equivalent IDL 2
interface GPS
  : Components::CCMObject
{
  position
    provide_MyLocation ();
  …
};
```



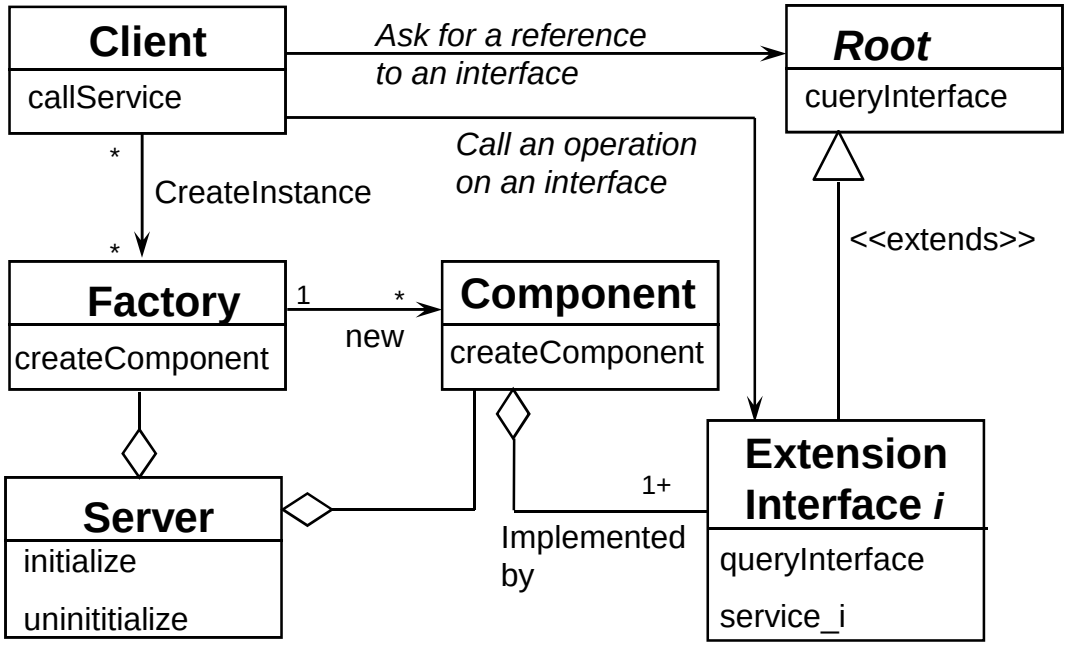
- Facet characteristics:
  - Define *provided* operation interfaces
    - Specified with **`provides`** keyword
      - *Logically* represents the component itself, not a separate entity contained by the component
        - However, facets have independent object references obtained from **`provide_*()`** factory operation
      - Can be used to implement *Extension Interface* pattern

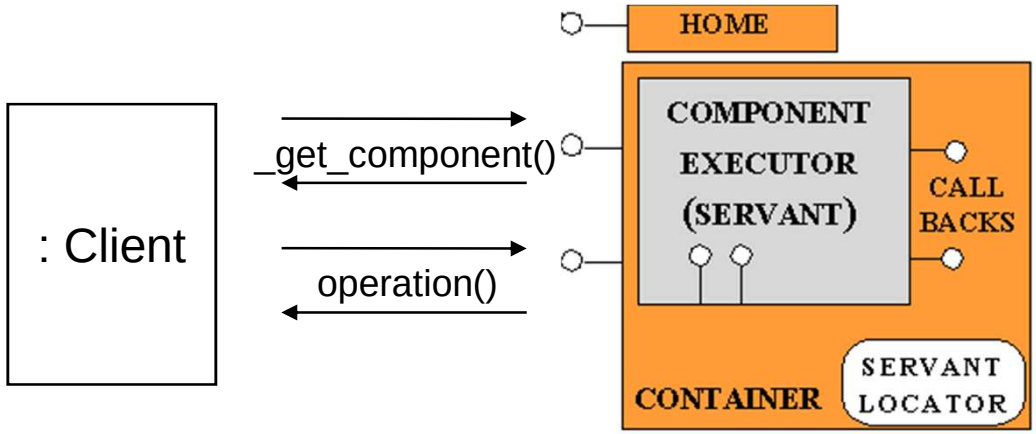
Extension Interface Pattern

The *Extension Interface* design pattern (POSA2) allows multiple interfaces to be exported by a component to prevent

- breaking of client code &
- bloating of interfaces

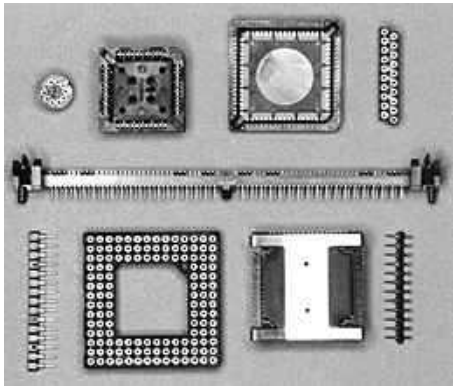
when developers extend or modify component functionality



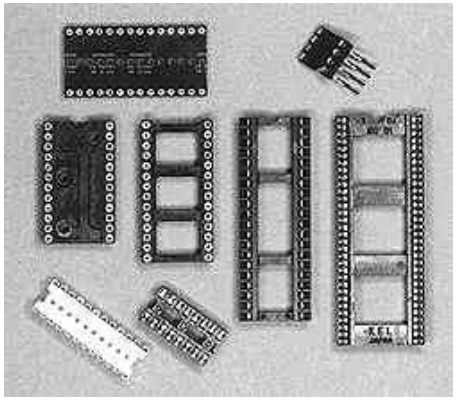


Using Other Components

- Context
  - Components need to collaborate with several different types of components/applications
  - These collaborating components/applications may provide different types of interfaces



- Problems with CORBA 2.x
  - No standard way to specify interface dependencies
  - No standard way to connect an interface to a component

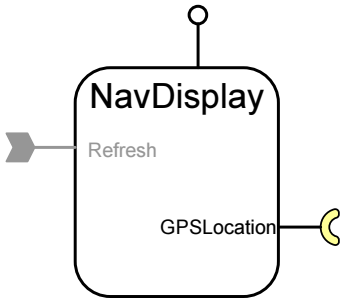


- CCM Solution
  - Define receptacles, a.k.a. *required* interfaces, which are distinct named connection points for potential connectivity
    - Represents the “bottom of the Lego”

Component Receptacles

```
// IDL 3
component NavDisplay
{
 …
 uses position GPSLocation;
 …
};
```

```
// Equivalent IDL 2
interface NavDisplay
  : Components::CCMObject
{
  …
  void connect_GPSLocation
         (in position c);
  position disconnect_GPSLocation();
  position get_connection_GPSLocation ();
  …
};
```

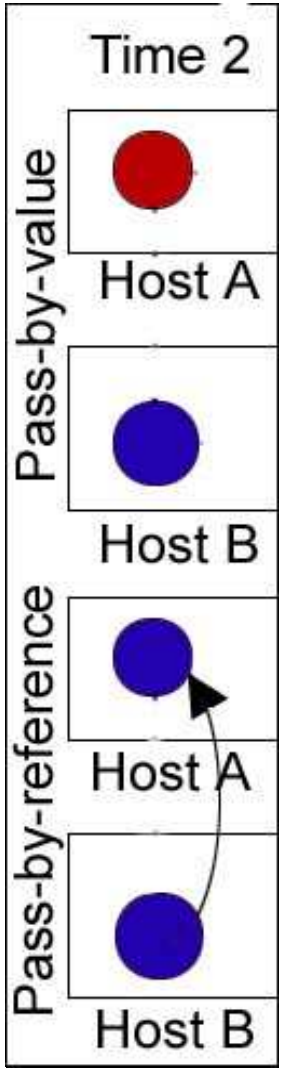


- Receptacle characteristics
  - Define a way to connect one or more *required* interfaces to this component
    - Specified with **`uses`** (**`multiple`**) keyword
    - Can be *simplex* or *multiplex*
      - Connections are established *statically* via tools during deployment phase
      - Connections are managed *dynamically* at run-time by containers to offer interactions with clients or other components via callbacks
      - CCM also enables connection establishment during run-time

Event Passing

- Context
  - Components often want to communicate using publisher/subscriber message passing mechanism
- Problems with CORBA 2.x
  - Standard CORBA Event Service is dynamically typed, i.e., there's no static type-checking connecting publishers/subscribe
  - Non-trivial to extend request/response interfaces to support event passing
  - No standard way to specify an object's capability to generate & process events
- CCM Solution
  - Standard **`eventtype`** & **`eventtype`** consumer interface (which are based on **`valuetypes`**)
  - Event sources & event sinks ("push mode" only)

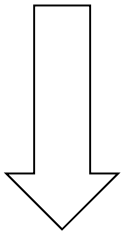
CORBA Valuetypes

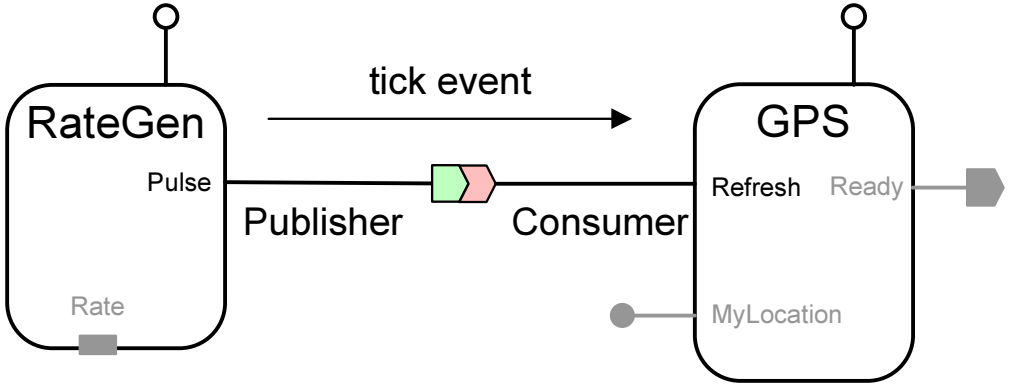


- Context
  - Parameters of IDL operations that are an **`interface`** type always have *pass-by-reference* semantics (even **`in`** parameters)
  - IDL interfaces hide implementations from clients
- Problems
  - Clients cannot instantiate CORBA objects
  - IDL **`structs`** are passed by value, but don't support operations or inheritance
- CORBA Solution
  - The IDL **`valuetype`**
    - Always passed by value
    - Can have both operations & state
    - Supports inheritance

Component Events

```
// IDL 3
eventtype tick
{
  public rateHz Rate;
};
```



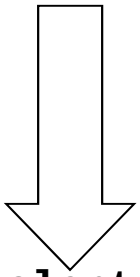


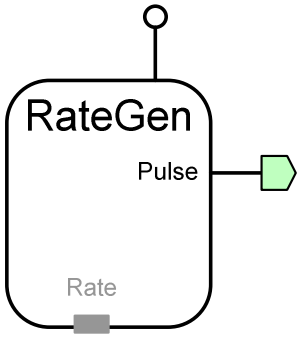

```
// Equivalent IDL 2
valuetype tick : Components::EventBase
{
  public rateHz Rate;
};
interface tickConsumer :
  Components::EventConsumerBase {
  void push_tick
    (in tick the_tick);
};
```

- Events are implemented as IDL **`valuetypes`**
- Defined with the new IDL 3 **`eventtype`** keyword
  - This keyword triggers generation of additional interfaces & glue code

Component Event Sources

```
// IDL 3
component RateGen
{
  publishes tick Pulse;
  emits tick Trigger;
  …
};
```






```
// Equivalent IDL 2
interface RateGen :
  Components::CCMObject {
  Components::Cookie
    subscribe_Pulse
    (in tickConsumer c);
  tickConsumer
    unsubscribe_Pulse
    (in Components::Cookie ck);
  …
};
```

- Event source characteristics
  - Named connection points for event production
  - Two kinds of event sources: *publisher & emitter*
    - **`publishes`** = may be multiple consumers
    - **`emits`** = only one consumer
  - Two ways to connect with event sinks
    1. Consumer connects directly
    2. CCM container mediates access to CosEvent channels

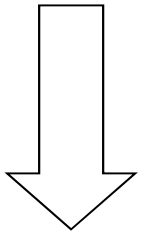
CCM Cookies

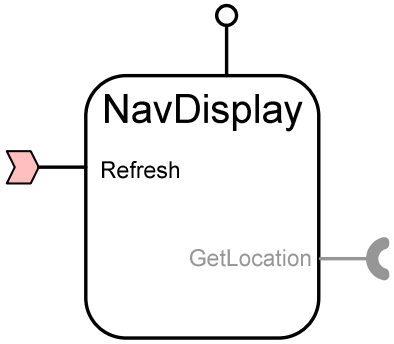
```
module Components
{
  valuetype Cookie
  {
    private CORBA::OctetSeq
            cookieValue;
  };

  interface Receptacles
  {
    Cookie connect (…);
    void disconnect (in Cookie ck);
  };

  interface Events
  {
    Cookie subscribe (…);
    void unsubscribe (in Cookie ck);
  };
};
```

- Context
  - Event sources & receptacles correlate **`connect()`** & **`disconnect()`** operations
- Problem
  - Object references cannot reliably be tested for equivalence
- CCM Solution
  - **`Cookie valuetype`**
    - Generated by receptacle or event source implementation
    - Retained by client until needed for **`disconnect()`**
    - Used as a unique id

Component Event Sinks

```
// IDL 3
component NavDisplay
{
 …
 consumes tick Refresh;
};
```

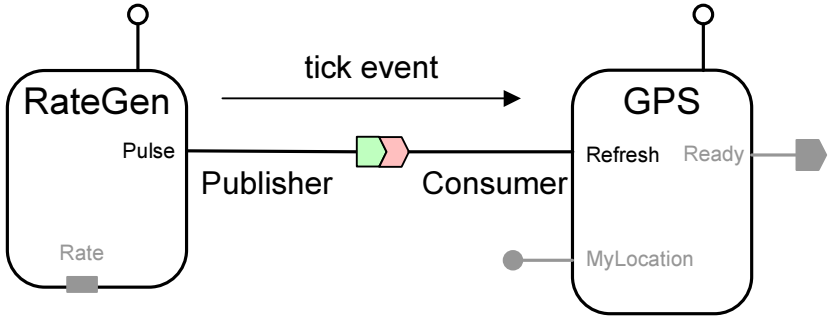





```
// Equivalent IDL 2
interface NavDisplay :
  Components::CCMObject
{
 …
 tickConsumer
   get_consumer_Refresh ();
 …
};
```

- Event sink characteristics
  - Named connection points into which events of a specific type may be pushed
  - Multiple event sinks of same type can subscribe to the same event sources
  - No distinction between emitter & publisher
  - Connected to event sources via object reference obtained from **`get_consumer_*()`** factory operation

CCM Events




```
// IDL 2
valuetype tick :
  Components::EventBase {…};

interface tickConsumer :
  Components::EventConsumerBase
   {…};
```

```
// C++ mapping
class tickConsumer : // ...
{
  virtual void push_event
   (Components::EventBase *evt);
  …
};
```

- Context
  - Generic event **`push()`** operation requires a generic event type
- Problem
  - User-defined **`eventtypes`** are not generic
- CCM Solution
  - **`EventBase abstract valuetype`**

```
module Components
{
  abstract valuetype EventBase {};

  interface EventConsumerBase {
    void push_event (in EventBase evt);
  };
};
```

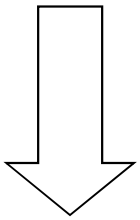
Enables both statically- & dynamically-typed event passing

The Need to Configure Components

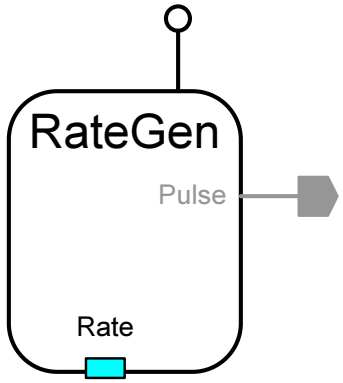
- Context
  - To make component implementations more adaptable, components properties should be (re)configurable, e.g., color, size, strategies, etc.
- Problems
  - Applications shouldn't commit to a configuration too early
  - No standard way to specify component's configurable parameters in CORBA 2.x
  - Need standard mechanisms to configure components
- CCM Solution
  - Configure components via *attributes* in assembly/deployment environment, by homes, and/or during component initialization

Component Attributes

```
// IDL 3
typedef unsigned long
        rateHz;
component RateGen
  supports rate_control
{
  attribute rateHz Rate;
};
```



```
// Equivalent IDL 2
interface RateGen :
  Components::CCMObject,
  rate_control
{
  attribute rateHz Rate;
};
```



- Attribute characteristics
  - Named configurable properties intended for component configuration
    - e.g., optional behaviors, modality, resource hints, etc.
  - Can raise user-defined exceptions (new CCM capability)
  - Exposed through accessors & mutators
  - Can be set by various configuration mechanisms
    - e.g., XML descriptor files generated by modeling tools

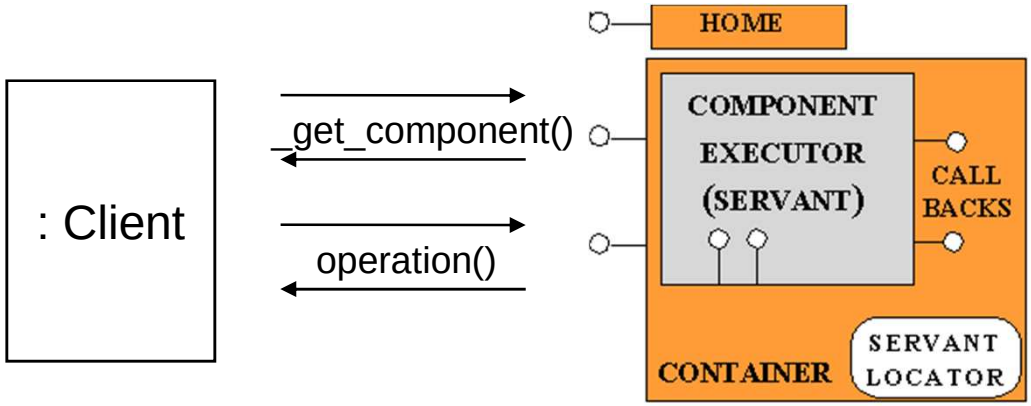
Connecting Components

- Context
  - Components need to be connected together to form complete applications
- Problems
  - Components can have multiple ports with different types & names
  - It's not scalable to write code manually to connect a set of components for a specific application

- CCM Solutions
  - Provide introspection interface to discover component capability
  - Provide generic port operations to connect components using external deployment & configuration tools
  - Represents snapping the lego bricks together

CCM Navigation & Introspection

- Navigation & introspection capabilities provided by **`CCMObject`**
  - i.e., via **`Navigation`** interface for facets, **`Receptacles`** interface for receptacles, & **`Events`** interface for event ports
- Navigation from component base reference to any facet(s) via generated facet-specific operations
  - e.g., **`Components::CCMObject::get_all_facets()`** & **`Components::CCMObject::provide()`**
- Navigation from any facet to component base reference with **`CORBA::Object::_get_component()`**
  - Returns nil if not a component facet, else component reference



All this navigation & introspection code is auto-generated by the CIDL compiler in the form of servant!

Using Navigation Interfaces of a Component

```
 1 int
 2 main (int argc, char *argv[])
 3 {
 4   CORBA::ORB_var orb =
 5     CORBA::ORB_init (argc, argv);
 6-10  // Get the NameService reference…
11   CORBA::Object_var o = ns->resolve_str (“myHelloHome");
12   HelloHome_var hh = HelloHome::_narrow (o.in ());
14   HelloWorld_var hw = hh->create ();
15   // Get all facets & receptacles
16   Components::FacetDescriptions_var fd = hw->get_all_facets ();
17   Components::ReceptacleDescriptions_var rd =
18     hw->get_all_receptacles ();
19   // Get a named facet with a name “Farewell”
20   CORBA::Object_var fobj = hw->provide (“Farewell”);
21   // Can invoke sayGoodbye() operation on Farewell after
22   // narrowing to the Goodbye interface.
23   ...
24   return 0;
25 }
```

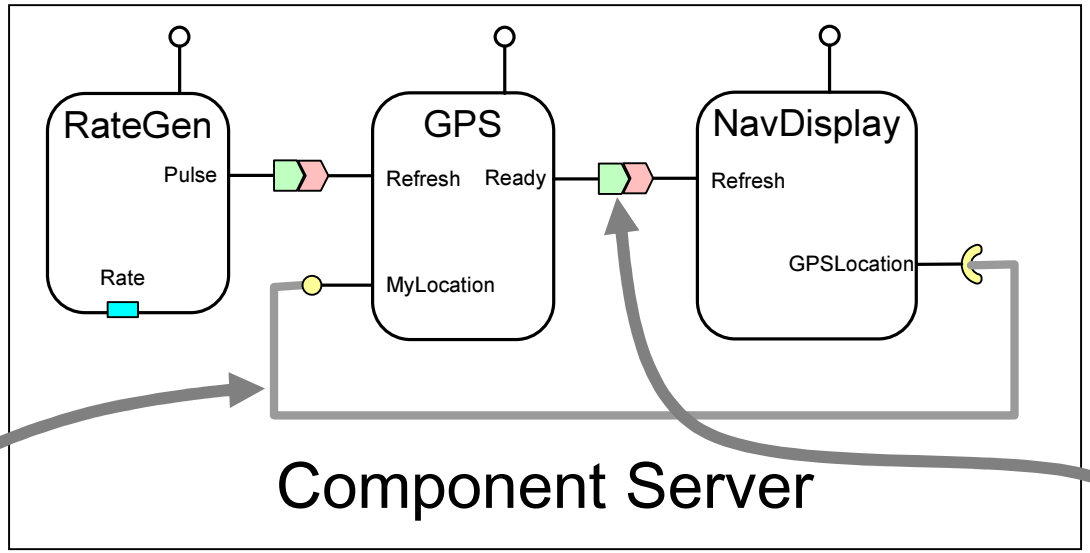
Generic Port Operations

Port	Equivalent IDL2 Operations	Generic Port Operations (**CCMObject**)
Facets	`provide_name ();`	`provide ("name");`
Receptacles	`connect_name (con);` `disconnect_name ();`	`connect ("name", con);` `disconnect ("name");`
Event sources (publishes only)	`subscribe_name (c);` `unsubscribe_name ();`	`subscribe ("name", c);` `unsubscribe ("name");`
Event sinks	`get_consumer_name();`	`get_consumer ("name");`

- Generic port operations for **provides**, **uses**, **subscribes**, **emits**, & **consumes** keywords are auto-generated by the IDL compiler
  - Apply the Extension Interface pattern
  - Used by CCM deployment & configuration tools
  - Lightweight CCM spec doesn't include equivalent IDL 2 operations

Example of Connecting Components

CCM components are connected via deployment tools during launch phase



- Facet → Receptacle

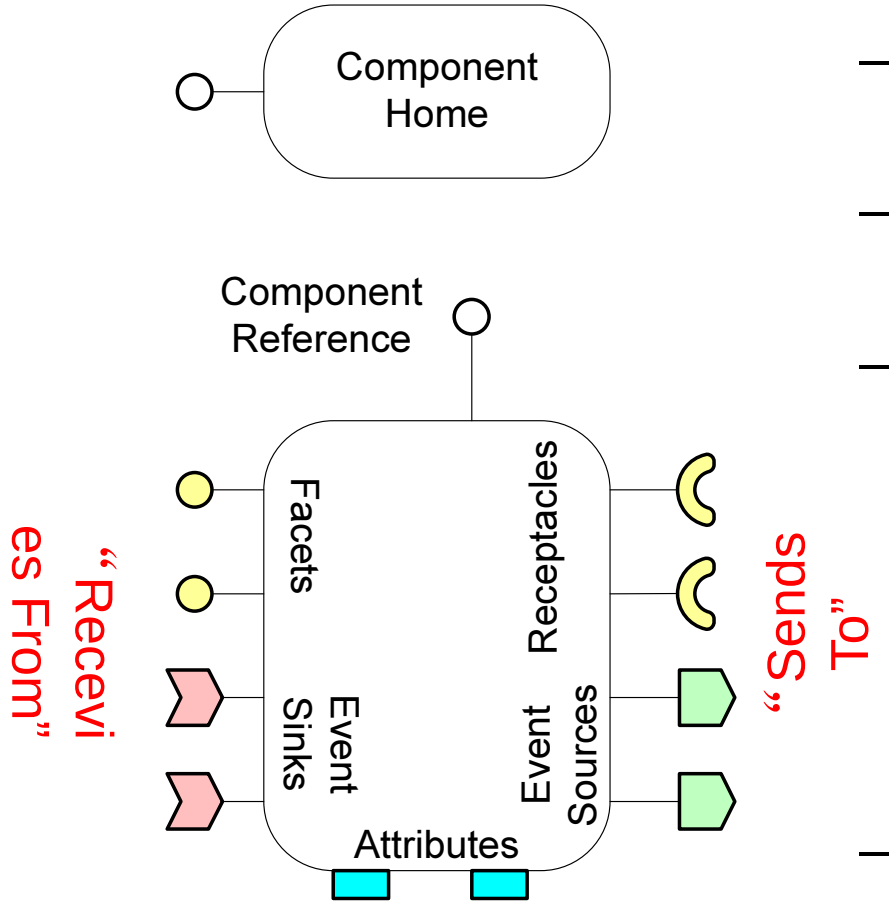
```
objref = GPS->provide
  ("MyLocation");
NavDisplay->connect
  ("GPSLocation", objref);
```

- Event Source → Event Sink

```
consumer = NavDisplay->
  get_consumer ("Refresh")
GPS->subscribe
  ("Ready", consumer);
```

Connected object references are managed by containers

Recap – CCM Component Features



- IDL 3 component from a *client* perspective
  - Define component life cycle operations (*i.e.*, **`home`**)
  - Define what a component **`provides`** to other components
  - Define what a component **`requires`** from other components

    Define what *collaboration modes* are used between components
    - Point-to-point via operation invocation
    - Publish/subscribe via event notification
  - Define which component **`attributes`** are configurable
- IDL 3 maps to "equivalent IDL 2 Interfaces"

Summary of Client OMG IDL Mapping Rules

- A *component type* is mapped to an interface inheriting from **`Components::CCMObject`**
- *Facets & event sinks* are mapped to a factory operation for obtaining the associated reference
- *Receptacles* are mapped to operations for connecting, disconnecting, & getting the associated reference(s)
- *Event sources* are mapped to operations for subscribing & unsubscribing for produced events
- An *event type* is mapped to
  - A value type that inherits from **`Components::EventBase`**
  - A consumer interface that inherits from **`Components::EventConsumerBase`**
- A *home type* is mapped to three interfaces
  - One for explicit user-defined operations that inherits from **`Components::CCMHome`**
  - One for generated implicit operations
  - One inheriting from both interfaces

We explored all of these mappings in detail in previous slides

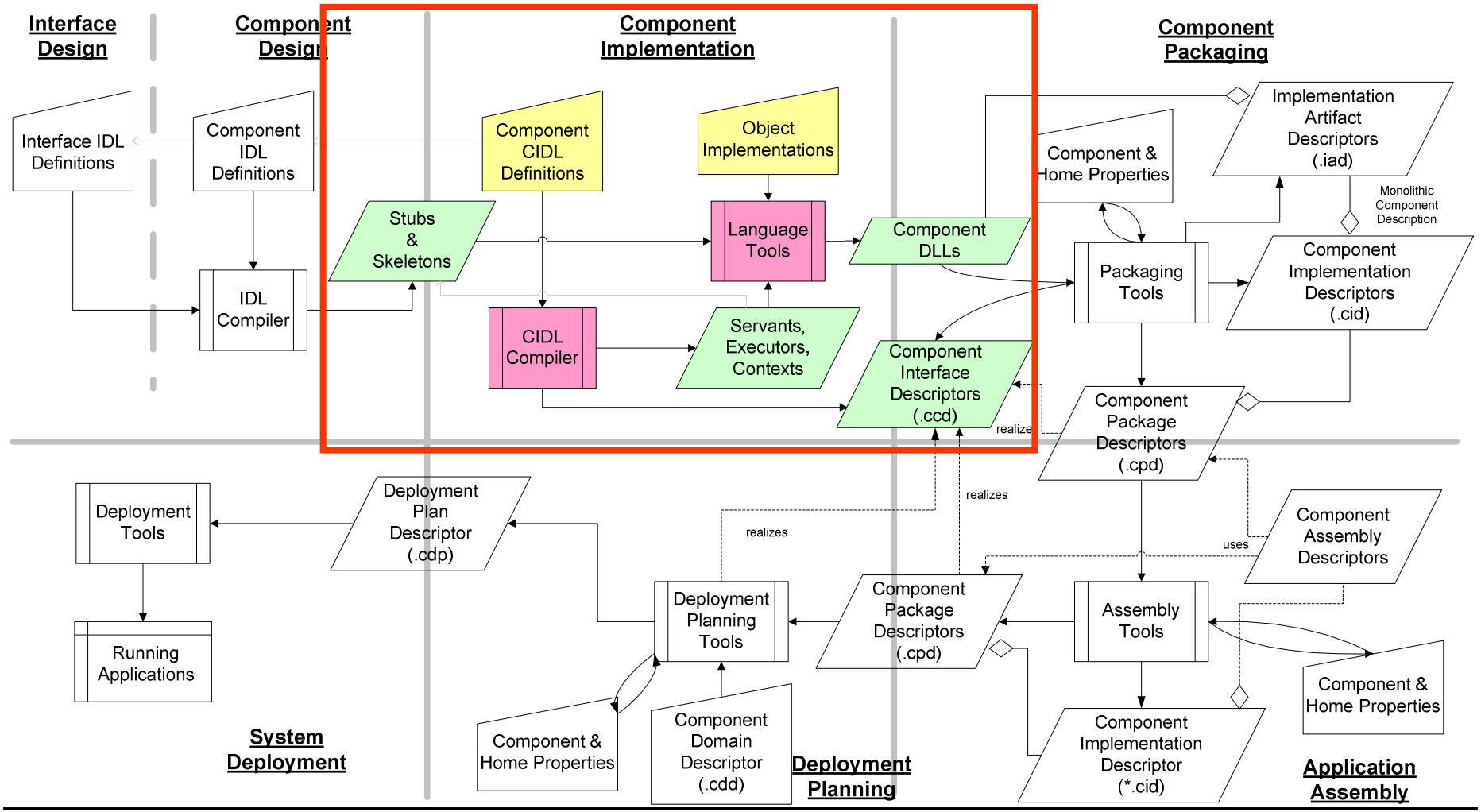
CCM Component Run-time Environment & Containers

www.cs.wustl.edu/~schmidt/cuj-18.doc

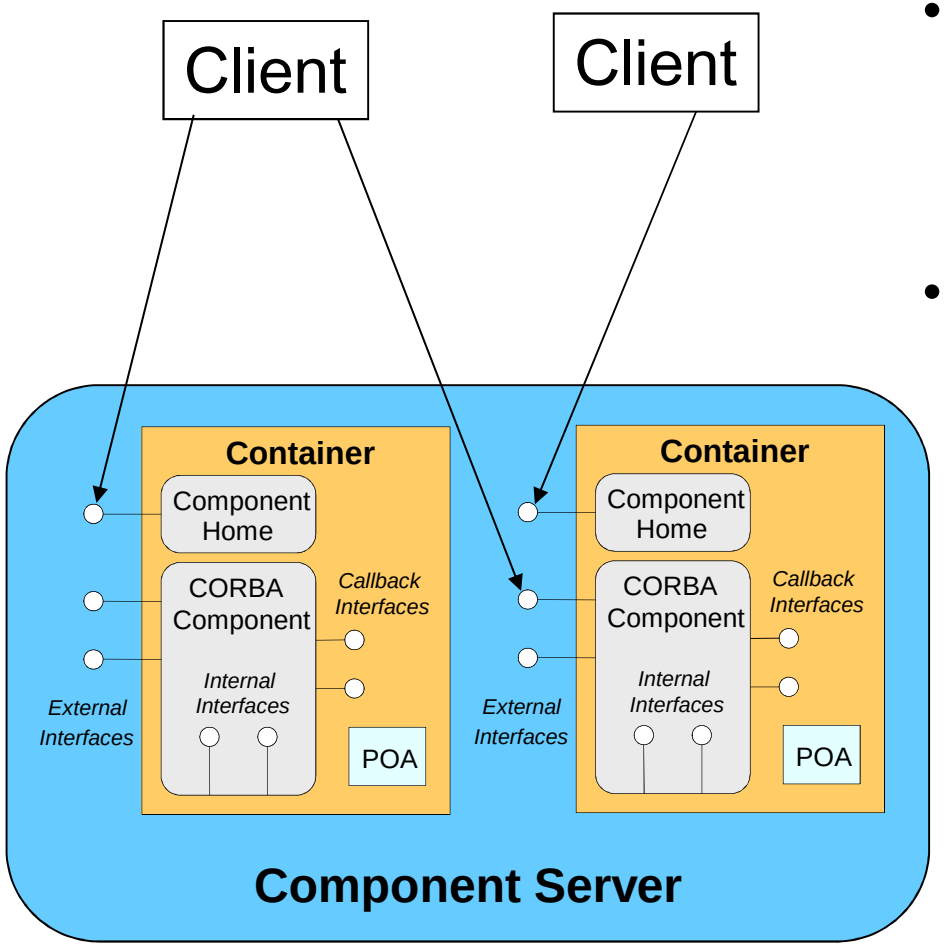
2012/4/11

Component Implementation Stage

Goal 1: Implement *components* in the context of *containers*

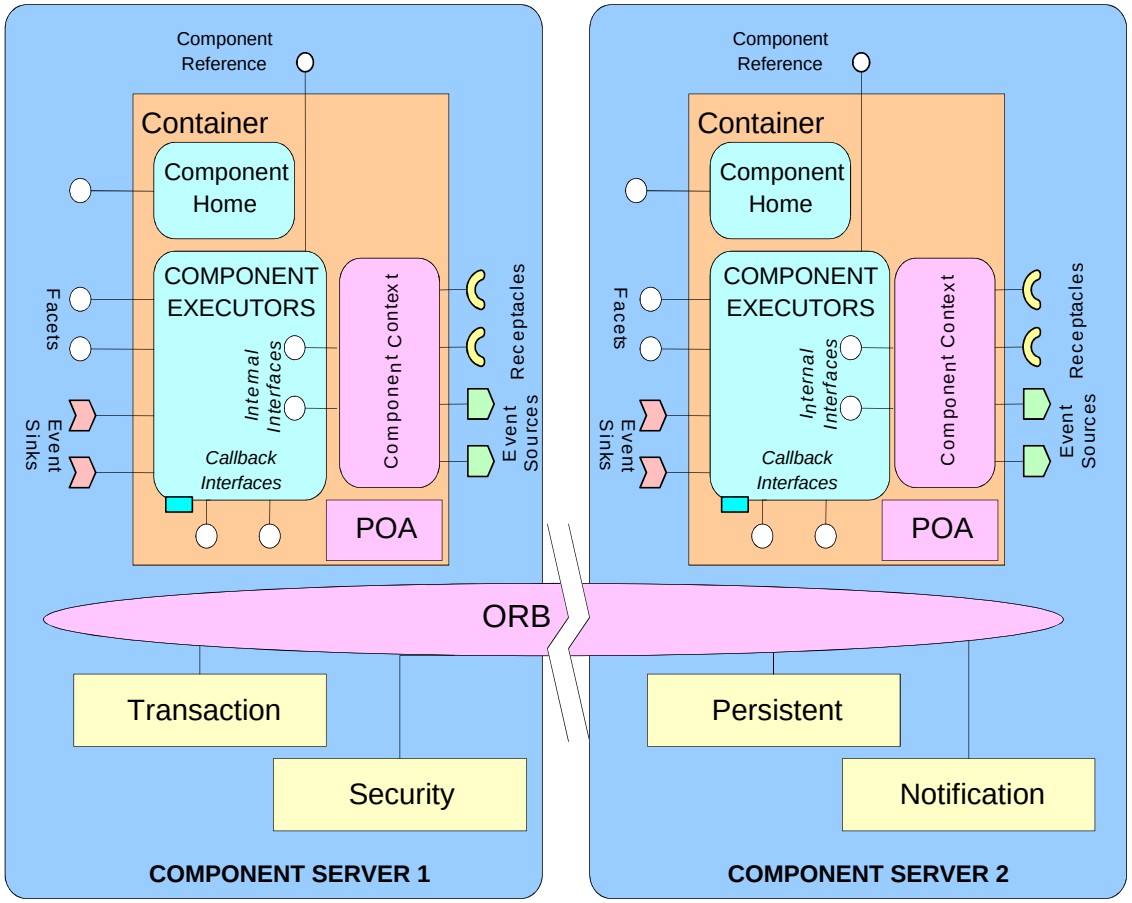


CCM Component Server Features



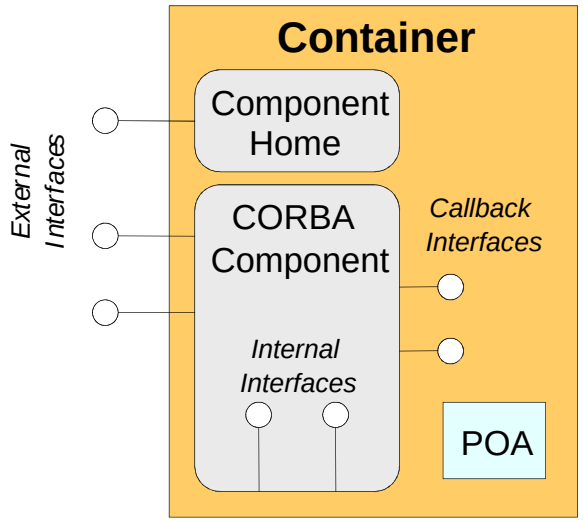
- CCM's primary enhancement to CORBA 2.x is its focus on *component servers & application deployment/configuration*
- CCM extends CORBA 2.x via
  - Higher-level abstractions of common servant usage models
  - Tool-based configuration & *meta-programming* techniques, e.g.:
    - Reusable run-time environment
    - Drop in & run
    - Transparent to clients
- The CCM *container framework* is central to this support

The CCM Container Framework



- A standard framework within CCM component servers
- Extends the Portable Object Adaptor (POA) with common patterns, e.g.,
  - Automatic activation & deactivation of components
  - Optimize resource usage
- Provides simplified access to CORBA Common Services
  - e.g., security, transactions, persistence, & events
- Uses *callbacks* to manage component instances
  - e.g., session states, activation, deactivation, etc.

External, Internal, & Container Interfaces



- **External APIs** are interfaces provided to clients
- **Container APIs** are *internal interfaces* & *callback interfaces* used by component developers to build applications

- *Internal interfaces* are used by components to access container facilities

```
local interface CCMContext {
  CCMHome get_CCM_home ();
};
local interface SessionContext :
  CCMContext {
    Object get_CCM_object ();
};
```

- *Callback interfaces* are used by containers to call into the component's *executor*

```
local interface EnterpriseComponent{};
local interface SessionComponent :
   EnterpriseComponent {
     void set_session_context
       (in SessionContext ctx)
     void ccm_activate ();
     void ccm_passivate ();
     void ccm_remove ();
 };
```

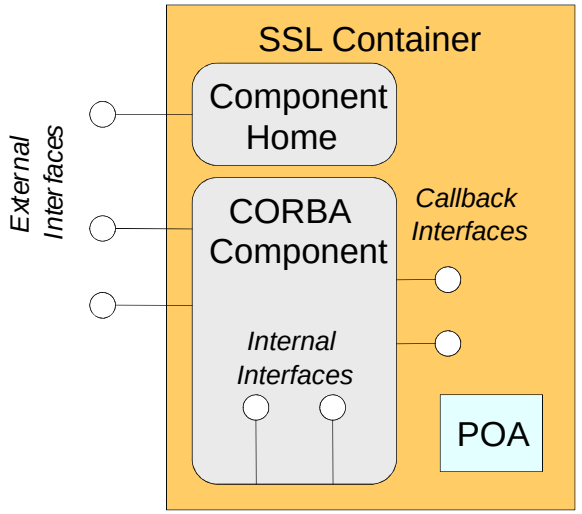
CCM Component/Container Categories

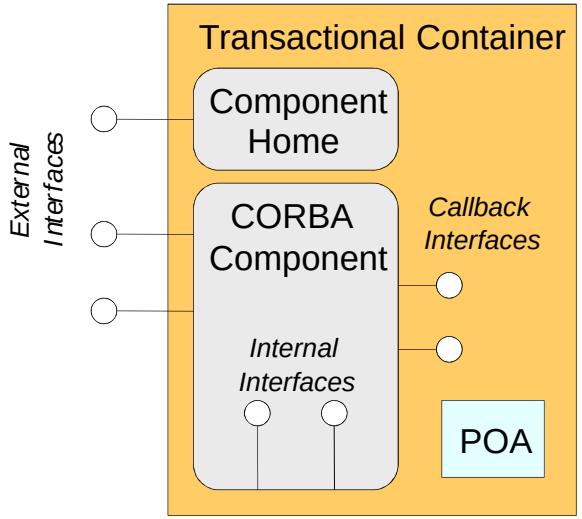
COMPONENT CATEGORY	CONTAINER IMPL TYPE	CONTAINER TYPE	EXTERNAL TYPE
Service	Stateless	Session	Keyless
Session	*Conversational*	*Session*	*Keyless*
Process	Durable	Entity	Keyless
Entity	Durable	Entity	Keyfull

In CCM these categories can be specified *declaratively* via a CIDL file, rather than programmed *imperatively*

Container-managed CORBA Policies

- Goal: decouple install-/run-time configuration policies from component implementation
- CORBA policy declarations defined for:
  - Servant lifetime
  - Transaction
  - Security
  - Events
  - Persistence
- Specified by component/composition developers using XML metadata and/or CIDL directives
- Implemented by the container, not the component
  - Uses Interceptor pattern (POSA2)





Component Implementation Framework (CIF)

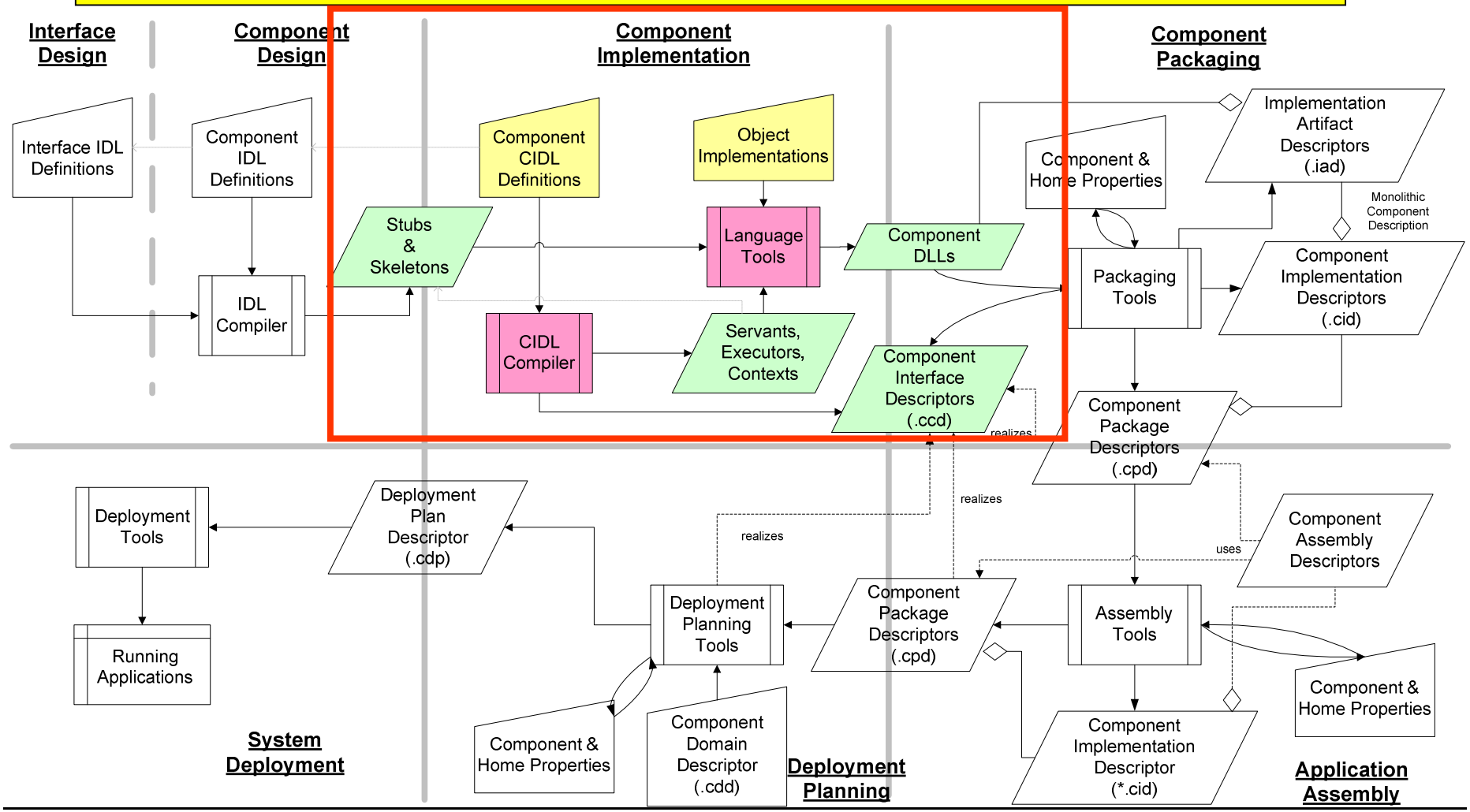
&

Component Implementation Definition Language (CIDL)

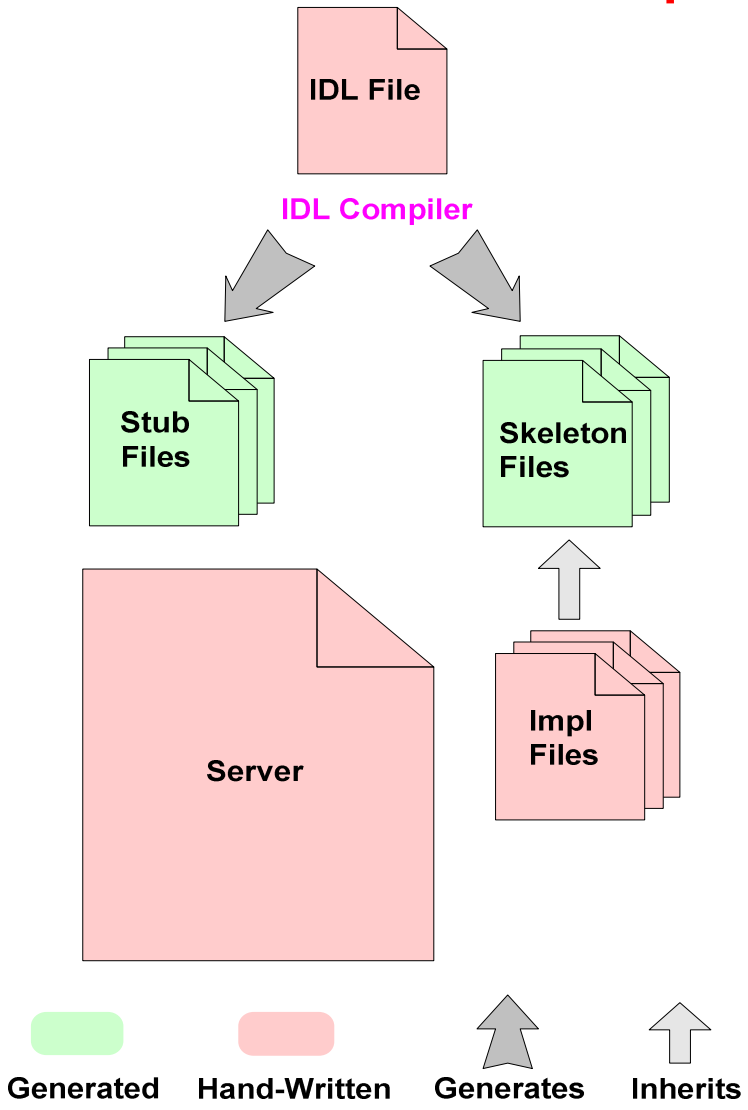
www.cs.wustl.edu/~schmidt/cuj-18.doc

Component Implementation Stage

Goal 2: Implement *components* & associate them with their *homes*



Difficulties with Implementing CORBA 2.x Objects



- **Problems**
  - Generic lifecycle & initialization server code must be handwritten, e.g.
    - Server initialization & event loop code
    - Support for introspection & navigation of object interfaces
  - Server application developers must
    - Keep track of dependencies their objects have on other objects
    - Manage the policies used to configure their POAs & manage object lifecycles
  - Consequences are *ad hoc* design, code bloat, limited reuse

Approach for Implementing Components

Requirements

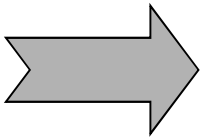
- Component implementations may need to support introspection, navigation, & manage connections
- Different component implementations may have different run-time requirements
- Different component run-time requirements may necessitate the use of different container policies

Approach: Generate as Much Code as Possible from Declarative Specs

IDL 3 & CIDL

Component & home definitions

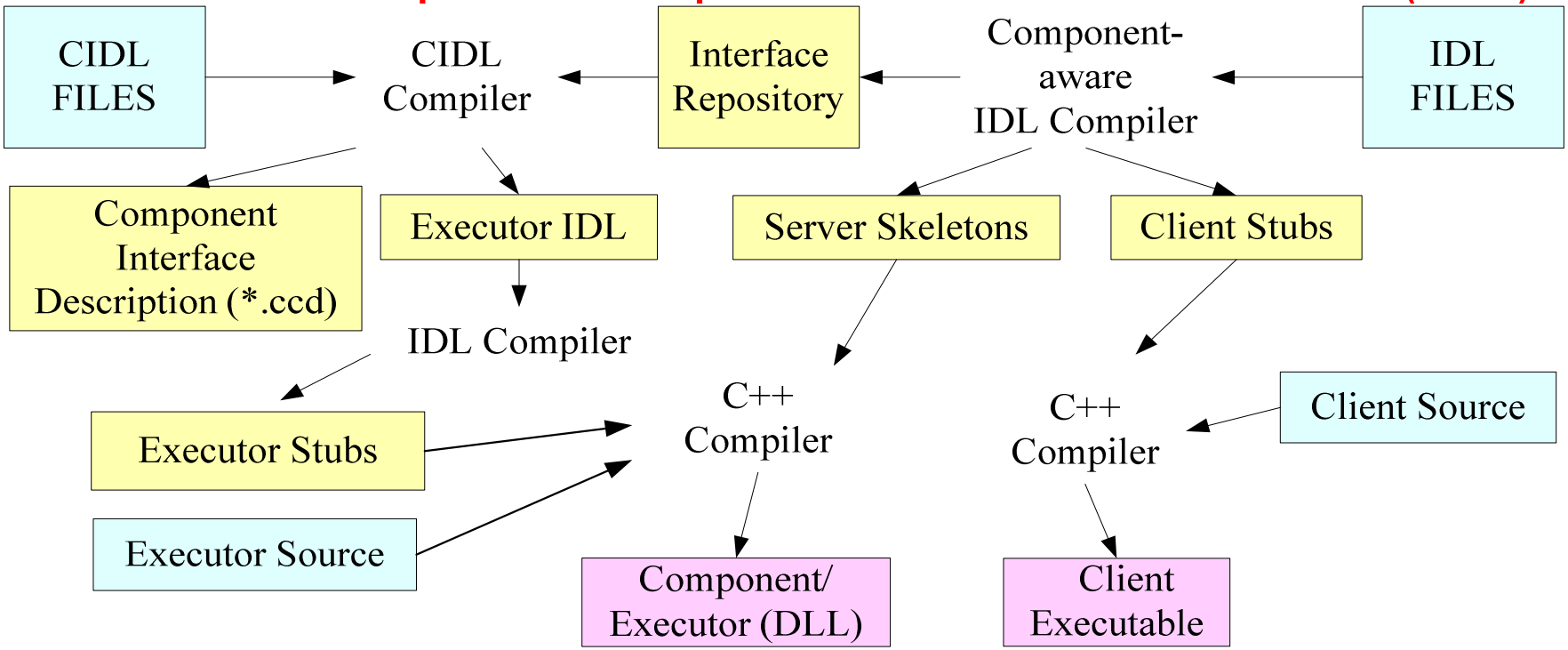
IDL 3 compiler



Generated Component & Home Servants

- Navigation interface operations
- Receptacle interface operations
- Event interface operations
- **`CCMObject`** interface operations
- **`CCMHome`** interface operations
- Implied equivalent IDL 2 port operations
- Application-related operations
  - i.e., facets, supported interfaces, event consumers

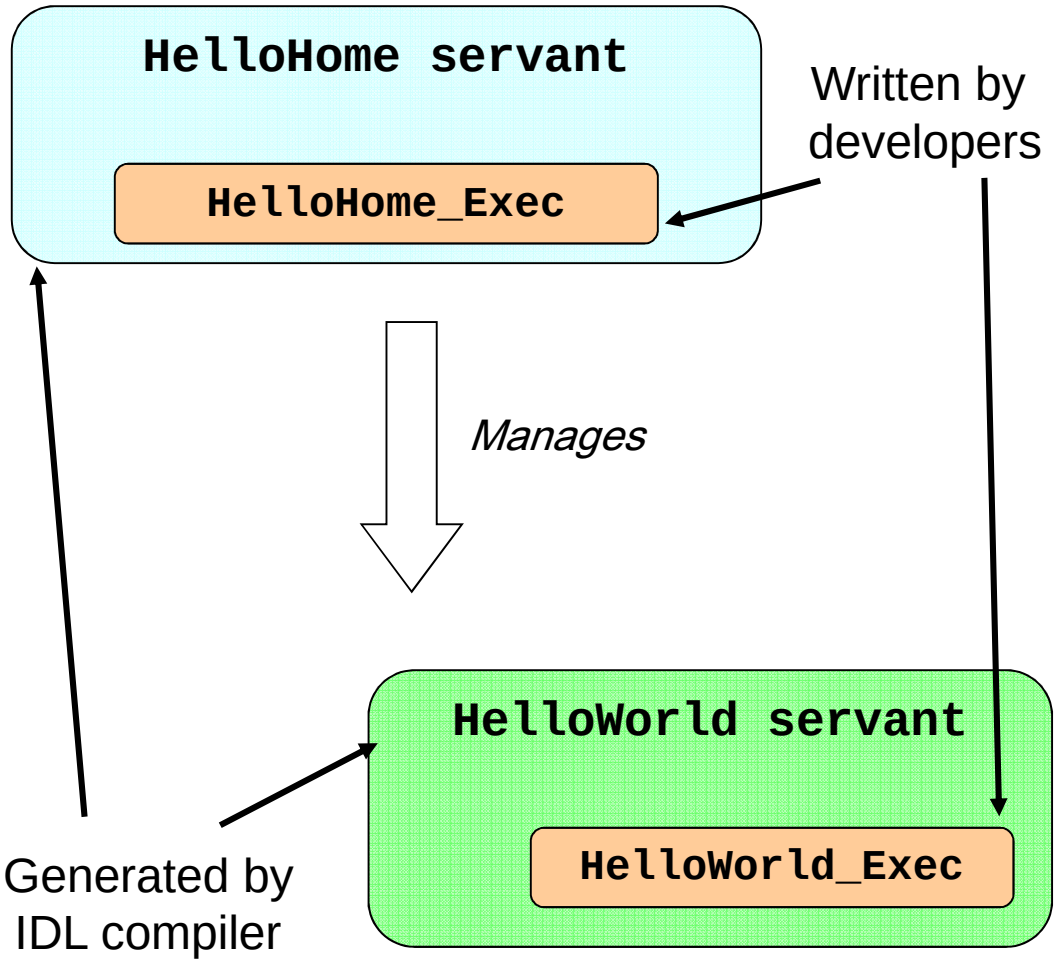
CCM Component Implementation Framework (CIF)



- Defines rules & tools for implementing components
  - i.e., specifies how to implement components via *executors*
- Simplifies component implementation
  - Developers only implement business logic, *not* activation, identification, port management, introspection, etc.
- Auto-generates much component "glue" code

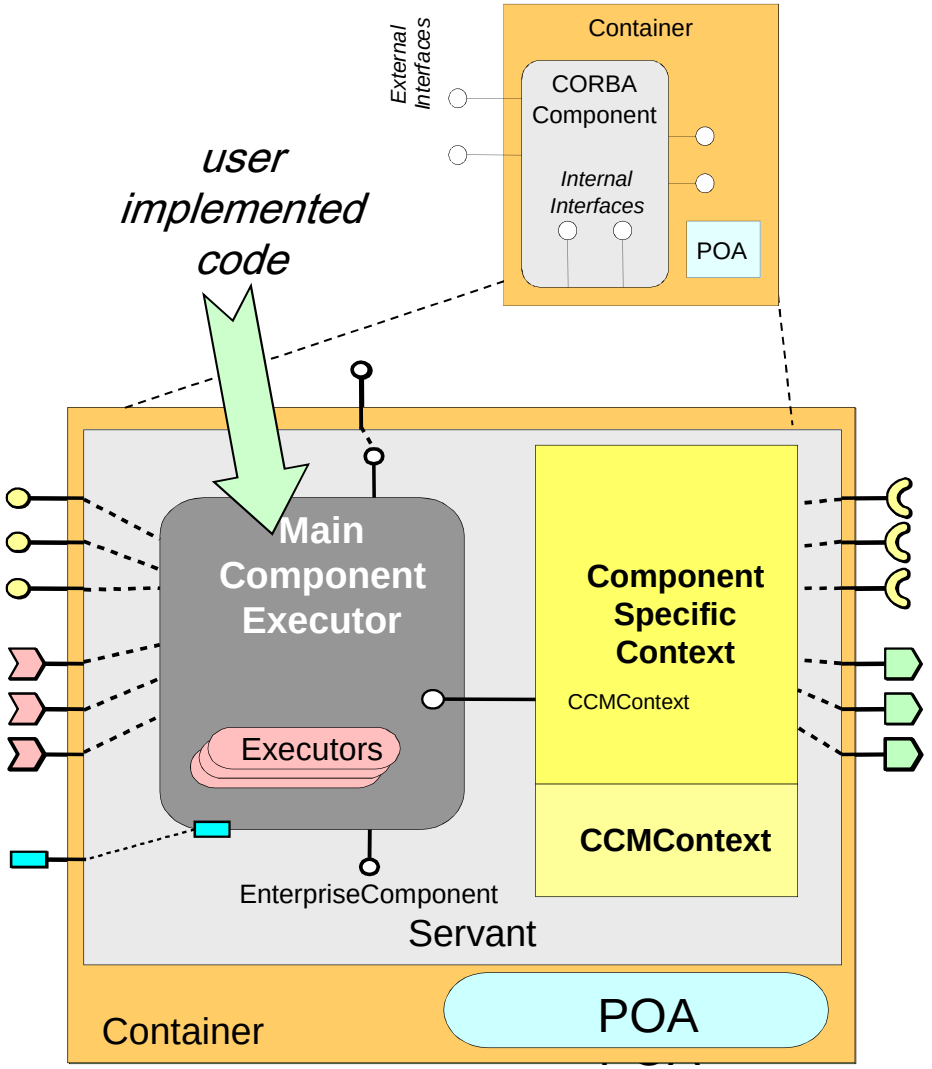
Component Executors & Home Executors

- 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

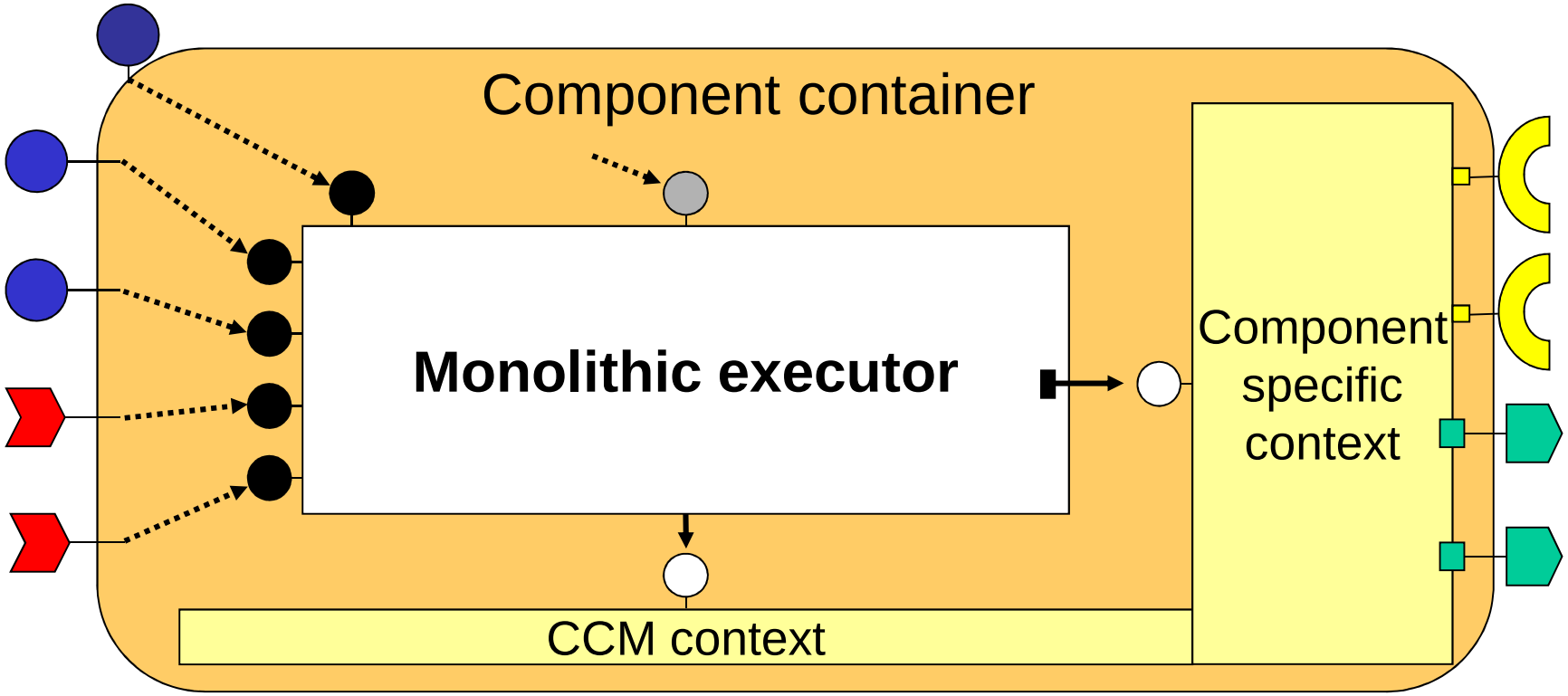


Executors (& Servants) Are Hosted by Containers

- Containers intercept invocations on executors & manage activation, security, transactions, persistency, etc.
- Component executors must implement a *local callback lifecycle interface* used by the container
  - **`SessionComponent`** for transient components
  - **`EntityComponent`** for persistent components
- Component executors can interact with their containers & connected components through a *context interface*

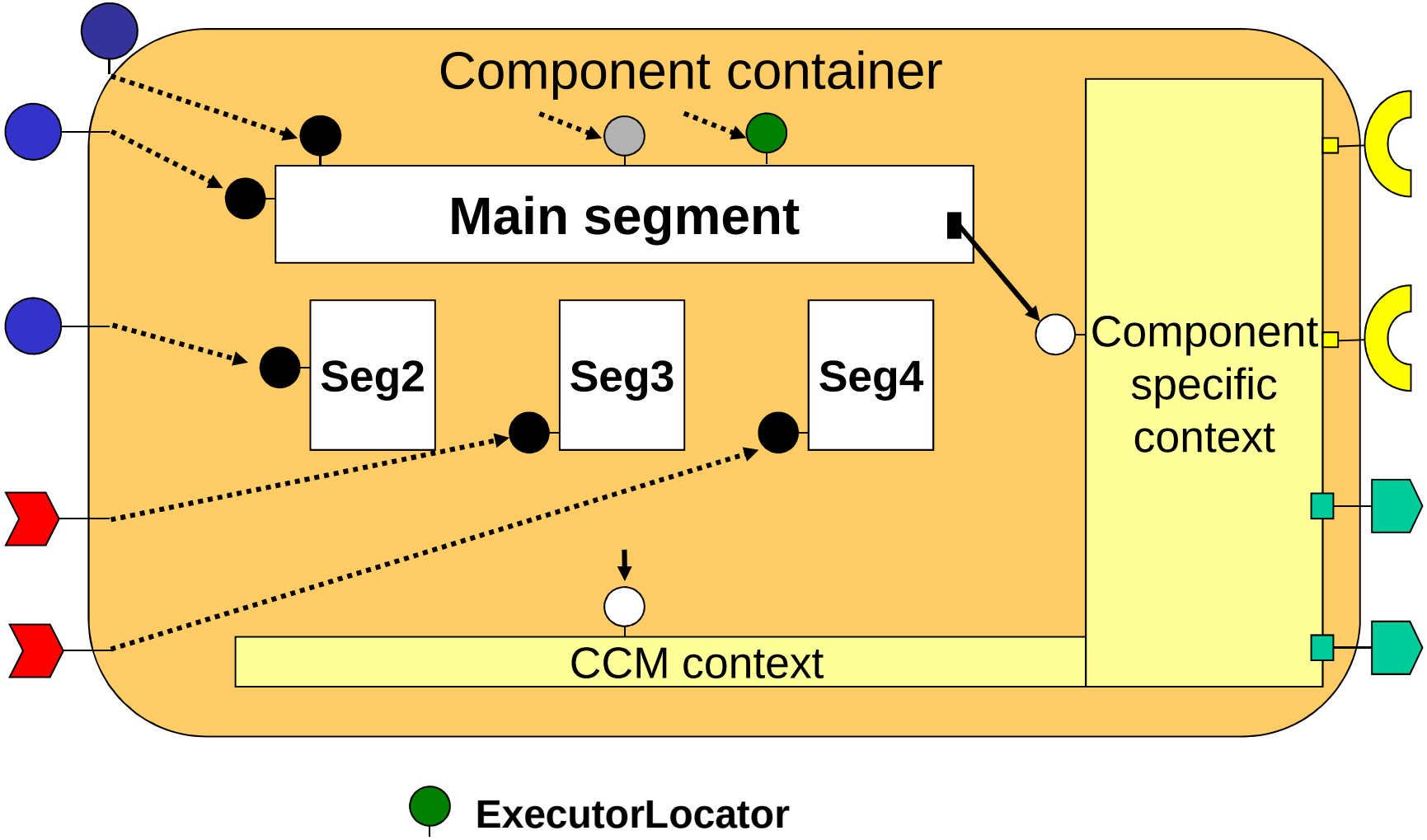


A Monolithic Component Executor



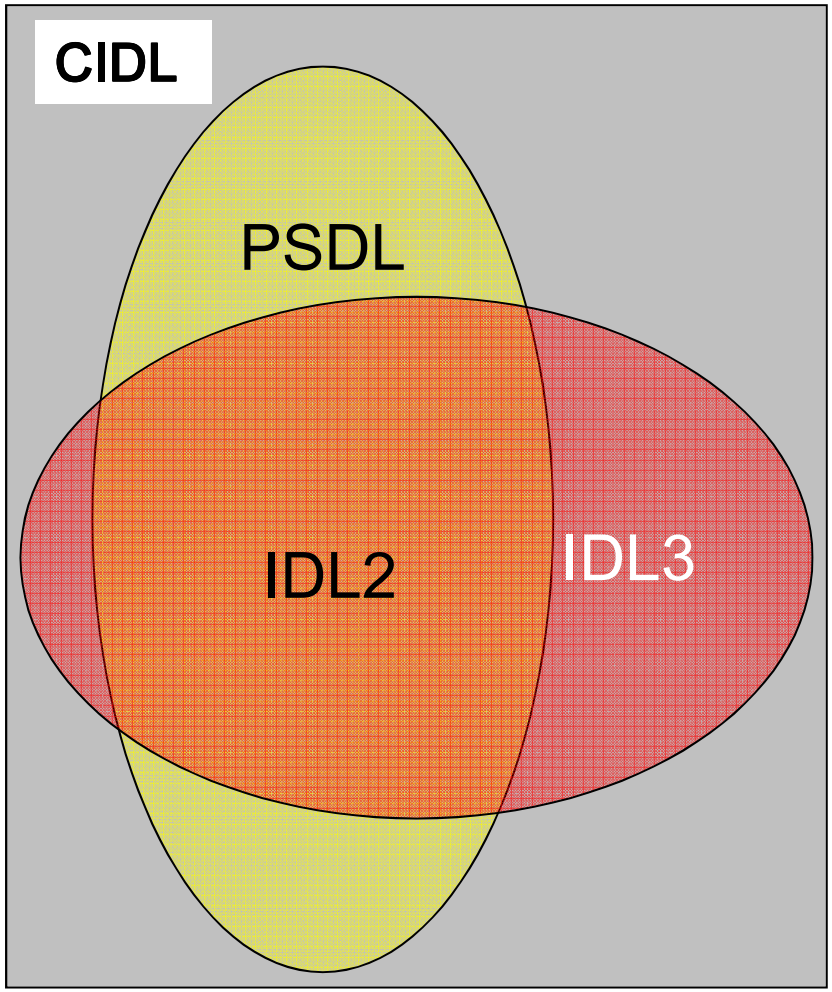
- Main component executor interface
- Facet or event sink executor interface
- **SessionComponent** or **EntityComponent**
- Component-oriented context interface
- Container-oriented context interface
- Context use
- Container interposition

A Segmented Component Executor



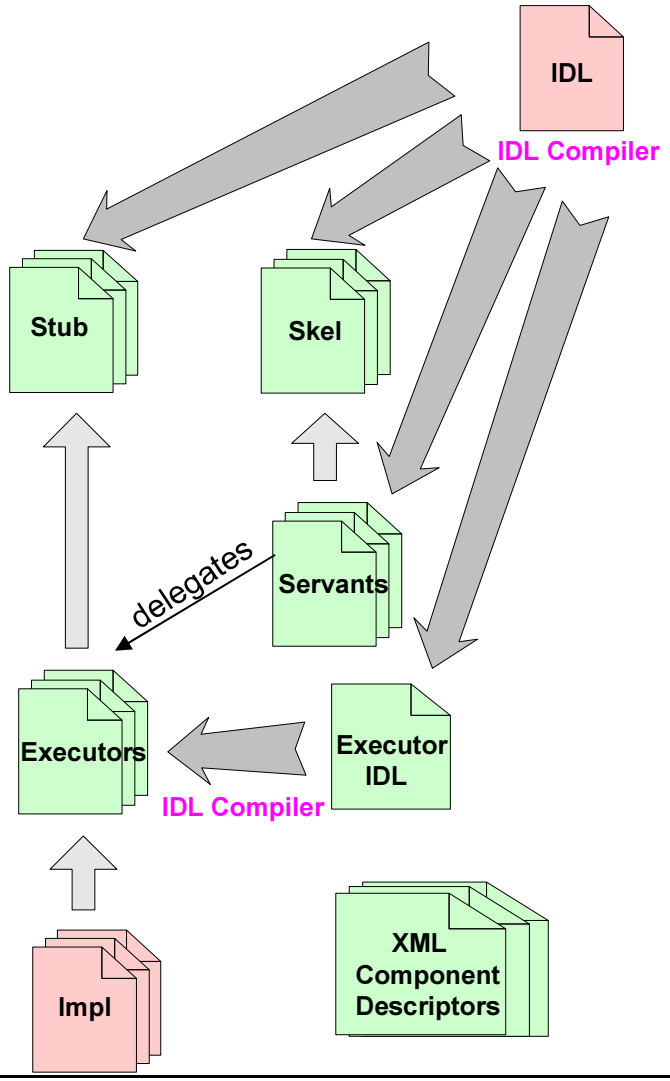
Segmented executors are deprecated in favor of assembly-based components

Overview of Component Implementation Definition Language (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)

Facilitating Component Implementation via CIDL

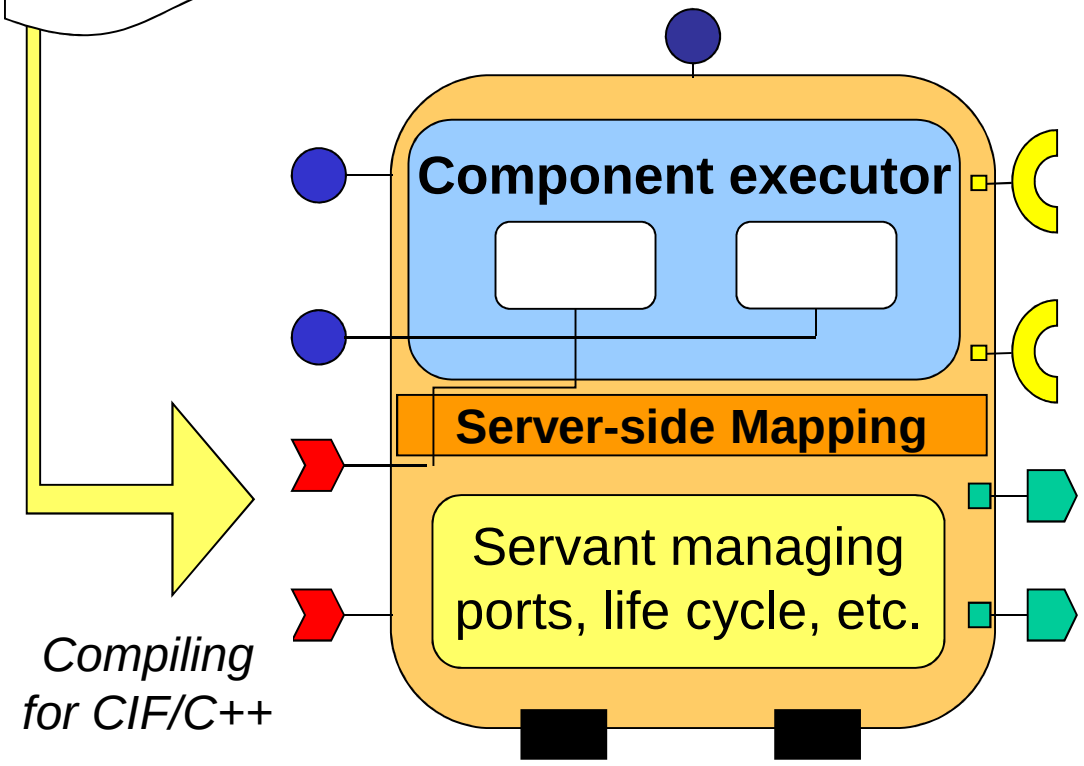


- CIDL is part of the CCM strategy for managing component-based applications
  - Enhances separation of concerns
  - Helps coordinate tools
  - Increases the ratio of generated to hand-written code
  - Server glue code is generated, installation & startup automated by other CCM tools

Connecting Components & Containers with CIDL

OMG 3.0 IDL file + CIDL

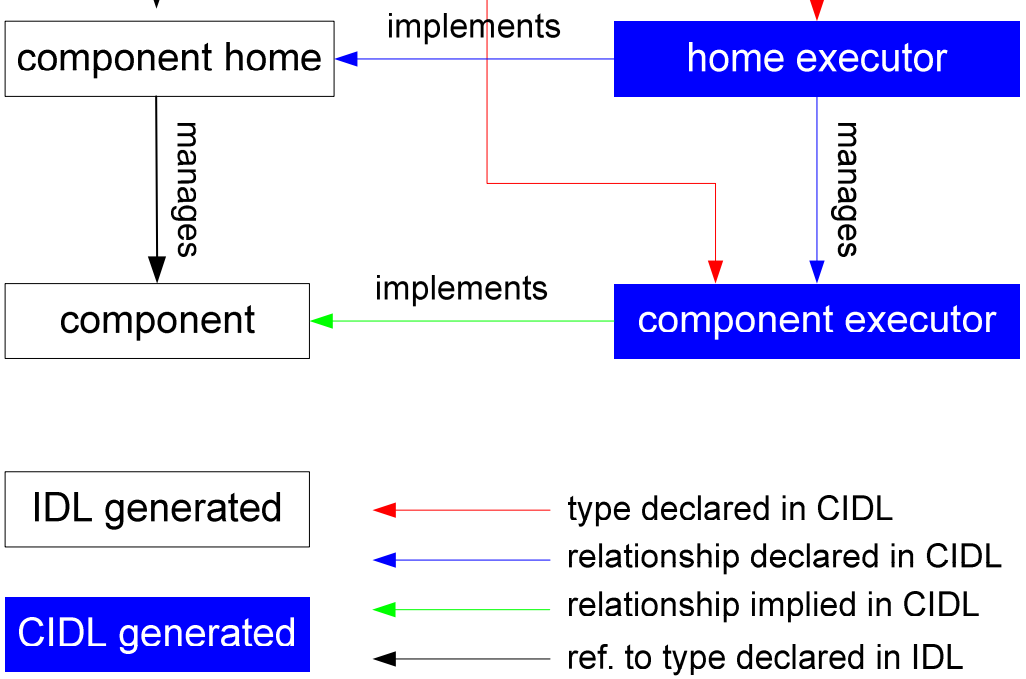
- CIDL & IDL 3.x compiler(s) generate infrastructure "glue" code that connects together component implementations (executors & servants) & containers that hosts them



- Infrastructure code in container intercepts invocations on executors
  - e.g., can be used to manage activation, security, transactions, persistency, & so on
- CCM CIF defines "executor mappings"
- CIDL file declaratively expresses container type

Facilitating Component Composition via CIDL

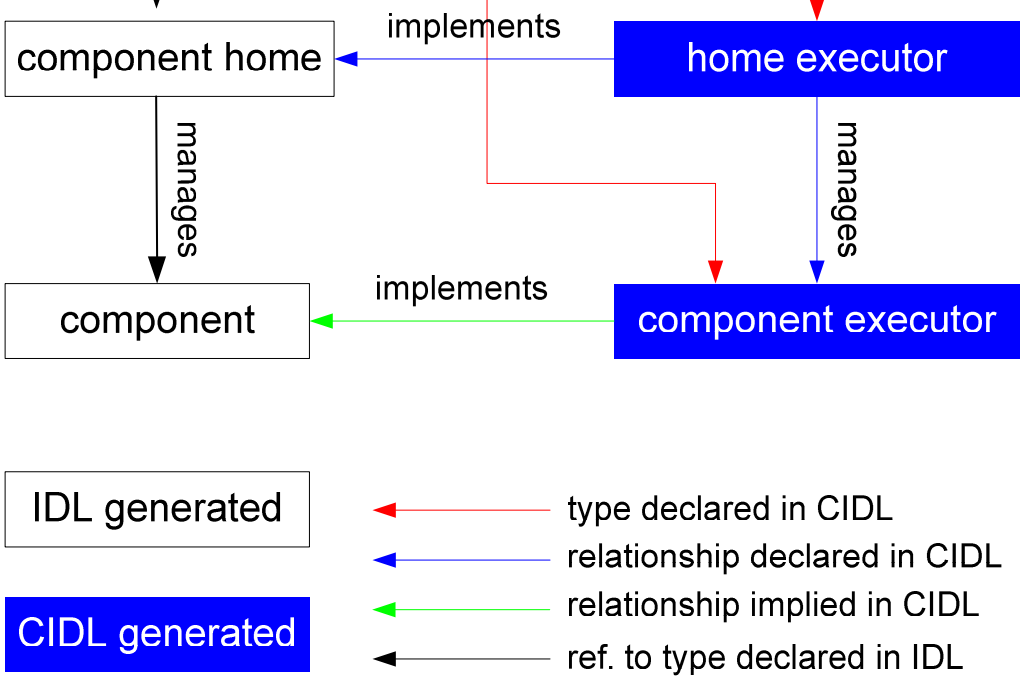
```
composition <category> <composition name> {
  home executor <home executor name> {
    implements <home type>;
    manages <executor name>;
  };
};
```



- **Composition features**
  - category
    - Specifies container (lifecycle) type (session, entity, etc.)
  - composition name
    - Specifies namespace for executor declarations
  - home executor name
    - Specify generated home name
  - executor name
    - Specify generated interface or class names
  - home type
    - Implicitly specifies managed component type

Facilitating Component Composition via CIDL

```
composition session Hello_Example {
  home executor HelloHome_Exec {
    implements HelloHome;
    manages HelloWorld_Exec;
  };
};
```



- **Composition features**
  - session
    - Keyless, conversation type of container
  - Hello_Example
    - Specifies namespace for executor declarations
  - HelloHome_Exec
    - Specify generated home name
  - HelloWorld_Exec
    - Specify generated interface or class names
  - HelloHome
    - Implicitly specifies managed component type

CCM Component Application Examples

www.cs.wustl.edu/~schmidt/cuj-19.doc

Steps for Developing CCM Applications

1. **Define your interfaces using IDL 2.x features**, e.g., use the familiar CORBA types (such as **struct**, **sequence**, **long**, **Object**, **interface**, **raises**, etc.) to define your interfaces & exceptions
2. **Define your component types using IDL 3.x features**, e.g., use the new CCM keywords (such as **component**, **provides**, **uses**, **publishes**, **emits**, & **consumes**) to group the IDL 2.x types together to form components
3. **Use IDL 3.x features to manage the creation of the component types**, e.g., use the new CCM keyword **home** to define factories that create & destroy component instances
4. **Implement your components**, e.g., using C++ or Java & the Component Implementation Definition Language (CIDL), which generates component servants, executor interfaces, associated metadata, & compositions
5. **Assemble your components**, e.g., group related components together & characterize their metadata that describes the components present in the assembly
6. **Deploy your components & run your application**, e.g., move the component assembly packages to the appropriate nodes in the distributed system & invoke operations on components to perform the application logic

Example 1: Hello World

```
// hello.idl
interface Hello {
  void sayHello (in string name);
};
interface Goodbye {
  void sayGoodbye (in string name);
};
component HelloWorld supports
Hello {
  provides Goodbye Farewell;
};
home HelloHome manages HelloWorld
{};
```

```
// hello.cdl
#include "hello.idl"
composition session
                Hello_Example
{
  home executor HelloHome_Exec
  {
    implements HelloHome;
    manages HelloWorld_Exec;
  };
};
```

- IDL 3 & CIDL definitions placed in **`hello.idl`** & **`hello.cdl`**, respectively

HelloWorld Component Executors

```
class HelloWorld_Exec_Impl
  : public virtual HelloWorld_Exec,
    public virtual ::CORBA::LocalObject
{
public:
  HelloWorld_Exec_Impl () {}
  ~HelloWorld_Exec_Impl () {}
  void sayHello (const char *name) {
    cout << "Hello World! -- from "
         << name << endl;
  }
};
```

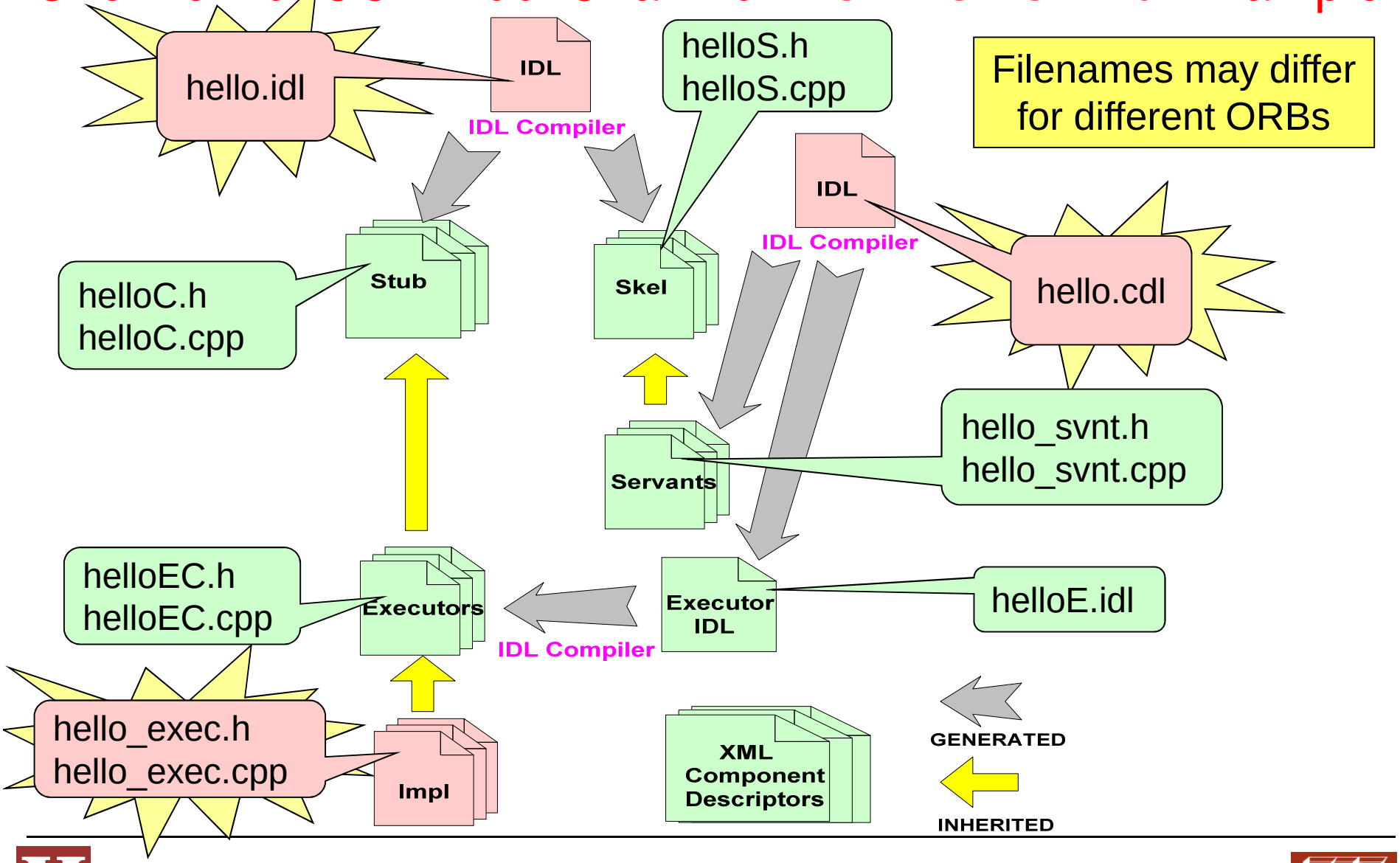
```
class HelloHome_Exec_Impl
  : public virtual HelloHome_Exec,
    public
virtual ::CORBA::LocalObject
{
public:
  HelloHome_Exec_Impl () {}
  ~HelloHome_Exec_Impl () {}

  Components::EnterpriseComponent_ptr
  create ()
  {
    return new HelloWorld_Exec_Impl;
  }
};
```

- **HelloWorld_Exec_Impl** executor implements **HelloWorld** component via **HelloWorld_Exec** executor IDL
- **HelloHome_Exec_Impl** executor implements lifecycle management of **HelloWorld** component

- **CORBA::LocalObject** is a variant of **CORBA::Object**
- Instances of of type **CORBA::LocalObject** cannot generate remote references

$CIAO_ROOT/docs/tutorial/Hello/

Overview of CCM Tool Chain for `HelloWorld` Example



`HelloWorld` IDL 3 File & Generated Stub/Skel Code

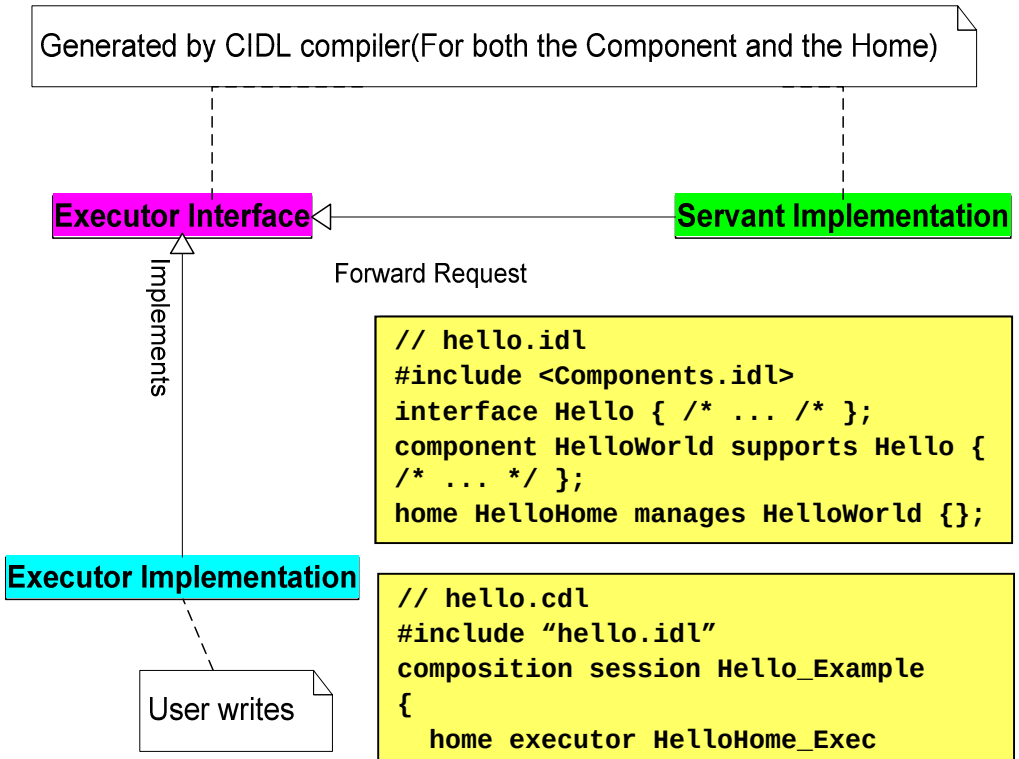
```
// hello.idl
#include <Components.idl>
interface Hello
{
  string sayHello (in string name);
};
component HelloWorld supports Hello
{ /* ... */
};
home HelloHome manages HelloWorld
{
};
```

- IDL file has IDL 3 keywords
  - e.g., **`component`**, **`home`**, **`supports`**, & **`manages`**
- Processed by IDL compiler that supports IDL 3 features
- Other tools could generate equivalent IDL 2

```
// helloC.h – Stub file
class HelloWorld
 : public virtual ::Components::CCMObject,
   public virtual ::Hello {};
// helloC.h – Stub file
class HelloHomeImplicit
 : public virtual ::Components::KeylessCCMHome {};
// helloC.h – Stub file
class HelloHomeExplicit
 : public virtual ::Components::CCMHome {};
// helloC.h – Stub file
class HelloHome
 : public virtual HelloHomeExplicit,
   public virtual HelloHomeImplicit {};
```

```
// helloS.h – Skeleton/Servant file
class POA_Hello
 : public virtual PortableServer::ServantBase {};
// helloS.h – Skeleton/Servant file
class POA_HelloWorld
 : public virtual POA_Components::CCMobject,
   public virtual POA_Hello { /* ... */ };
```

`HelloWorld` CIDL & Generated Servant Code




```
// hello.idl
#include <Components.idl>
interface Hello { /* ... /* };
component HelloWorld supports Hello {
/* ... */ };
home HelloHome manages HelloWorld {};
```

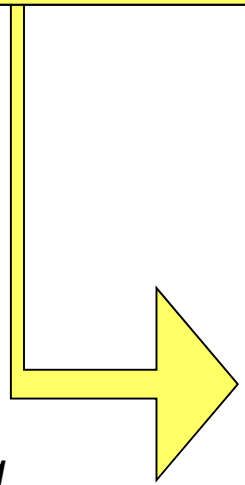
```
// hello.cdl
#include "hello.idl"
composition session Hello_Example
{
  home executor HelloHome_Exec
  {
    implements HelloHome;
    manages HelloWorld_Exec;
  };
};
```

Servant code also contains generated component-specific context classes

- CIDL compiler generates
  - *Servant code*, which is transparent to developers
  - *Executor IDL*, which developers then implement
- Servant code is generated for
  - Components
    - **`HelloWorld_Servant`**
    - **`HelloWorld_Context`**
  - Homes
    - **`HelloHome_Servant`**
  - Facets
    - **`<facet name>_Servant`**

HelloWorld CIDL-Generated Servants (hello_svnt.*)

```
// hello.cdl
#include "hello.idl"
composition session Hello_Example
{
  home executor HelloHome_Exec
  {
    implements HelloHome;
    manages HelloWorld_Exec;
  };
};
```



Compiling for CIF/C++

```
class HelloWorld_Context :
  public virtual ::CCM_HelloWorld_Context,
  public virtual ::CORBA::LocalObject
{
  // Operations from Components::CCMContext
  // Operations from Components::SessionContext
  // Operations from CCM_HelloWorld_Context
};
```

```
class HelloWorld_Servant :
  public virtual POA_HelloWorld
{
  // Supported operations
  // Operations on the navigation interface
  // Operations for the receptacle interfaces
};
```

```
class HelloHome_Servant :
  public virtual POA_HelloHome
{
  // Supported interface operations
  // Home operations
  // Factory & attribute operations
  // ImplicitHome operations
  ::HelloWorld_ptr create ();
};
```

HelloWorld CIDL-Generated Servant Details (1/6)

```
// hello.idl
#include <Components.idl>

interface Hello
{};

component HelloWorld supports Hello
{};

home HelloHome manages HelloWorld
{};
```

```
// hello.cdl
#include "hello.idl"

composition session Hello_Example
{
  home executor HelloHome_Exec
  {
    implements HelloHome;
    manages HelloWorld_Exec;
  };
};
```

```
// hello_svnt.h
#include "helloEC.h"
#include "helloS.h"

namespace Hello_Example
{
  class HelloWorld_Servant;

  class HelloWorld_Context
    : public virtual CCM_HelloWorld_Context {
    friend class HelloWorld_Servant;

    // Operation overrides from base classes -
    // Components::SessionContext and
    // Components::CCMContext
  };
}
```

- Composition name maps to C++ namespace
  - Not spec-required
  - Helps implementors avoid name clashes
- CIDL compiler navigates through implements & (IDL) manages
  - Gets component name
  - Maps name to servant, context, & base class names

HelloWorld CIDL-Generated Servant Details (2/6)

```
// hello.idl
#include <Components.idl>

interface Hello {};

interface Goodbye {};

eventtype MsgTrigger {};

component HelloWorld supports Hello {
  uses Goodbye GetGoodbye;
  publishes MsgTrigger GotMsg;
};

home HelloHome manages HelloWorld {};
```

```
// hello_svnt.h
#include "helloEC.h“
#include “helloS.h”

namespace Hello_Example {
  class HelloWorld_Servant;

  class HelloWorld_Context
    : public virtual CCM_HelloWorld_Context {
  public:
    friend class HelloWorld_Servant;

    virtual Goodbye_ptr get_connection_GetGoodbye ();

    virtual void push_GotMsg (MsgTrigger *ev);
  protected:
    virtual void connect_GetGoodbye (Goodbye_ptr obj);

    virtual Goodbye_ptr disconnect_GetGoodbye ();

    virtual Components::Cookie *
    subscribe_GotMsg (MsgTriggerConsumer_ptr c);

    virtual MsgTriggerConsumer_ptr
    unsubscribe_GotMsg (Components::Cookie *ck);
  };
}
```

- Receptacle (**uses**) declarations
  - Interface type maps to context op params
  - Name maps to context op names
- Event source (**publishes**) declarations
  - Type maps to params (event consumer)
  - Port name maps to subscribe/unsubscribe operations

HelloWorld CIDL-Generated Servant Details (3/6)

```
// hello.idl
#include <Components.idl>

interface Hello
{
  void SayHello (in string msg);
};

interface Goodbye
{
  void SayGoodbye (in string msg);
};

component HelloWorld supports Hello
{
  provides Goodbye Farewell;
  attribute string Message;
  consumes Trigger Listener;
};

home HelloHome manages HelloWorld
{};
```

```
// hello_svnt.h
#include "helloEC.h“
#include “helloS.h”

namespace Hello_Example
{
  class Goodbye_Servant
    : public virtual POA_Goodbye
  {
  public:
    virtual void SayGoodbye (const char *msg);
  };
}
```

- Facet (`provides`) declarations maps to C++ servant class
  - Separate servant class is implementation-specific
  - Helps C++ compiler reduce footprint
- Facet type maps to servant & base class name generation
- Facet interface operations mapped directly to servant class
  - Operation names map directly
  - Operation parameters map with the usual CORBA rules
- If no port declarations or supported interface operations
  - No new operations generated in servant class

HelloWorld CIDL-Generated Servant Details (4/6)

```
// hello.idl
#include <Components.idl>

interface Hello
{
  void SayHello (in string msg);
};

interface Goodbye
{
  void SayGoodbye (in string msg);
};

component HelloWorld supports Hello
{
  provides Goodbye Farewell;
  attribute string Message;
  consumes Trigger Listener;
};

home HelloHome manages HelloWorld
{};
```

```
// hello_svnt.h
#include "helloEC.h"
#include "helloS.h"

namespace Hello_Example {
  class HelloWorld_Servant
    : public virtual POA_HelloWorld
  {
  public:
    virtual void SayHello (const char *msg);

    virtual Goodbye_ptr provide_Farewell ();

    virtual char *Message ();

    virtual void Message (const char *Message);

    class TriggerConsumer_Listener_Servant
      : public virtual POA_TriggerConsumer
    {
    public:
      virtual void push_Trigger (Trigger *evt);
      virtual void push_event (Components::EventBase *e);
    };

    virtual TriggerConsumer_ptr get_consumer_Listener ();
  };
}
```

- Supported op maps directly to component servant op
- Facet type maps to the return type of the accessor op
- Facet name maps to component servant accessor op
- Attribute maps to get/set ops in the component servant
- Event sink (consumes) maps to nested class (impl-specific)
  - Also maps to accessor op for the event consumer

HelloWorld CIDL-Generated Servant Details (5/6)

```
// hello.idl
#include <Components.idl>

interface Hello
{};

component HelloWorld supports Hello
{};

home HelloHome manages HelloWorld
{};
```

```
// hello.cdl
#include "hello.idl"

composition session Hello_Example
{
  home executor HelloHome_Exec
  {
    implements HelloHome;
    manages HelloWorld_Exec;
  };
};
```

```
// hello_svnt.h
#include "helloEC.h"
#include "helloS.h"

namespace Hello_Example
{
  class HelloHome_Servant
    : public virtual POA_HelloHome
  {
    // Operation overrides from base class
    // Components::CCMHome
  };
}
```

- CIDL compiler navigates through `implements` to home type
  - Maps home type to home servant class
  - Also to generated base class name
- If home has no supported interfaces, operations, attributes, factories or finders
  - No new operations generated in servant class

HelloWorld CIDL-Generated Servant Details (6/6)

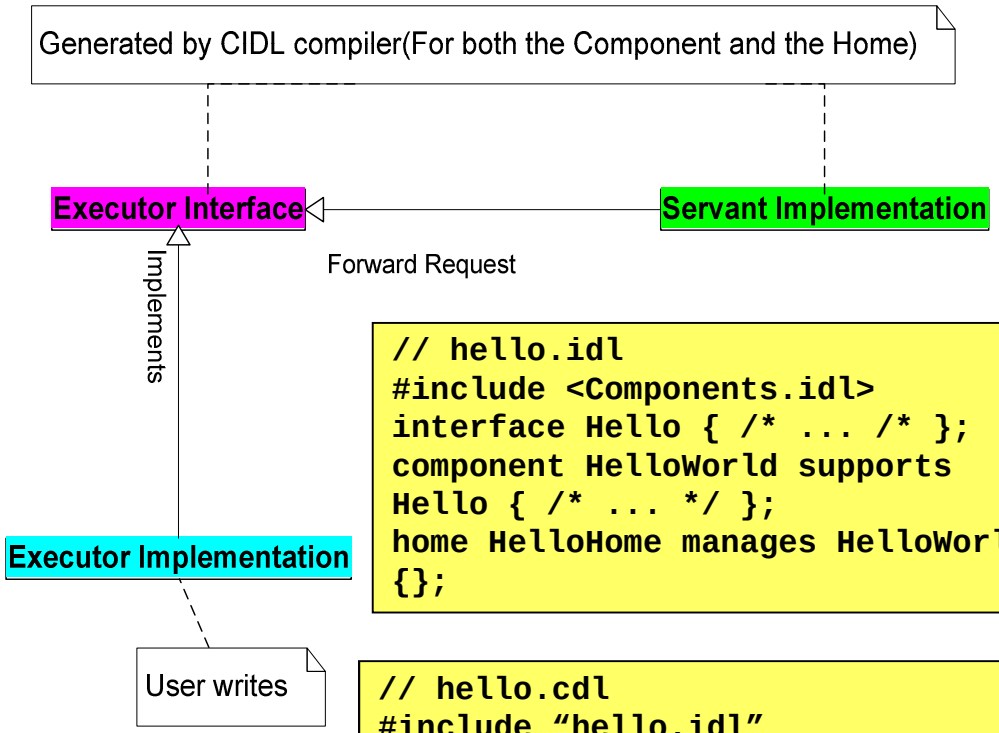
```
// hello.idl
#include <Components.idl>

interface Hello
{};

component HelloWorld supports Hello
{
  attribute string Message;
};

home HelloHome manages HelloWorld
{
  void utilityOp ();

  factory generate (in string msg);

  finder lookup (in long key);

  attribute long defaultKey;
};
```

```
// hello_svnt.h
#include "helloEC.h“
#include “helloS.h”

namespace Hello_Example {
  class HelloHome_Servant
    : public virtual POA_HelloHome
  {
  public:
    virtual void utilityOp ();

    virtual HelloWorld_ptr generate (const char *msg);

    virtual HelloWorld_ptr lookup (CORBA::Long key);

    virtual ::CORBA::Long defaultKey ();

    virtual void defaultKey (CORBA::Long default_key)
  };
}
```

- Home operations map directly to home servant
- `attribute` maps the same as for components

- Component type maps to implicit return type of
  - Operations generated from `factory` declarations
  - Operations generated from `finder` declarations
- Factory & finder operations can have only `in` parameters (if any)

`HelloWorld` CIDL & Generated Executor Code




```
// hello.idl
#include <Components.idl>
interface Hello { /* ... /* };
component HelloWorld supports
Hello { /* ... */ };
home HelloHome manages HelloWorld
{};
```

```
// hello.cdl
#include "hello.idl"
composition session Hello_Example {
  home executor HelloHome_Exec {
    implements HelloHome;
    manages HelloWorld_Exec;
  };
};
```

- *Executor interfaces* are IDL or C++/Java code
  - Generated by CIDL compiler
  - Must be implemented by component developers
- Generated code has interfaces for
  - Implicit & explicit homes
  - Main home executor
  - Main and/or monolithic component executors
  - Facet & consumer executor
  - Component context

- All executor interfaces are "locality constrained," i.e., use **`local`** keyword

Component application developers extend & implement executor interfaces

HelloWorld CIDL-Generated Executor IDL (helloE.idl)

```
// hello.cdl
#include "hello.idl"
composition session Hello_Example
{
  home executor HelloHome_Exec
  {
    implements HelloHome;
    manages HellowWorld_Exec;
  };
```

Component Executor Interface

```
local interface CCM_HelloWorld :
  Components::EnterpriseComponent,
  ::Hello {};
```

Component Context Interface

```
local interface CCM_HelloWorld_Context :
  ::Components::SessionContext {};
```

Explicit Home interface

Implicit Home interface

```
local interface CCM_HelloHomeImplicit
{
  ::Components::EnterpriseComponent create ()
   raises (::Components::CCMException);
};
```

```
local interface CCM_HelloHomeExplicit :
  ::Components::HomeExecutorBase {};
```

```
local interface CCM_HelloHome :
  CCM_HelloHomeExplicit,
  CCM_HelloHomeImplicit {};
```

```
local interface HelloWorld_Exec :
  CCM_HelloWorld,
  Components::SessionComponent {};

local interface HelloHome_Exec :
  ::CCM_HelloHome {};
```

Main Component Interface

Main Home Interface

These interface names are spec-compliant & generated by examining the CIDL file & included IDL files

HelloWorld CIDL-Generated Executor IDL Details (1/3)

```
// hello.idl
#include <Components.idl>

interface Hello
{};

component HelloWorld supports Hello
{};

home HelloHome manages HelloWorld
{};
```

```
// hello.cdl
#include “hello.idl”

composition session Hello_Example
{
  home executor HelloHome_Exec
  {
    implements HelloHome;
    manages HelloWorld_Exec;
  };
};
```

- Supported (`supports`) interface maps to component interface base interface

```
// helloE.idl
#include "hello.idl“

local interface CCM_HelloWorld
  : Components::EnterpriseComponent, ::Hello {};

local interface CCM_HelloWorld_Context
  : Components::SessionContext {};

local interface CCM_HelloHomeImplicit {
  Components::EnterpriseComponent create ()
    raises (Components::CCMException);
};

local interface CCM_HelloHomeExplicit
  : Components::HomeExecutorBase {};

local interface CCM_HelloHome
  : CCM_HelloHomeExplicit, CCM_HelloHomeImplicit {};
```

- Component type is mapped to 2 local interfaces
- Home type is mapped to 3 local interfaces
  - Implicit home interface declares spec-defined operations
  - Explicit home interface maps user-defined operations (if any)
  - Equivalent home interface inherits from both
- Composition type (`session`) maps to executor context base class

HelloWorld CIDL-Generated Executor IDL Details (2/3)

```
// hello.idl
#include <Components.idl>

interface Hello
{};

component HelloWorld supports Hello
{};

home HelloHome manages HelloWorld
{};
```

```
// helloE.idl
#include "hello.idl"

module Hello_Example
{
  local interface HelloWorld_Exec
     : CCM_HelloWorld, Components::SessionComponent
  {};

  local interface HelloHome_Exec : CCM_HelloHome
  {};
};
```

```
// hello.cdl
#include "hello.idl"

composition session Hello_Example
{
  home executor HelloHome_Exec
  {
    implements HelloHome;
    manages HelloWorld_Exec;
  };
};
```

- Composition name maps to IDL `module`
- Home executor name maps to IDL `local interface`
- Implemented home type maps to base interface (shown in previous slide)
- Component executor name maps to `local interface`
- Managed component type maps to base interface (shown in previous slide)
- Composition type (`session`) maps to a middleware base interface of the component executor

HelloWorld CIDL-Generated Executor IDL Details (3/3)

```
// hello.idl
#include <Components.idl>

interface Hello
{
  void SayHello (in string msg);
};

interface Goodbye
{
  void SayGoodbye (in string msg);
};

component HelloWorld supports Hello
{
  provides Goodbye Farewell;
  attribute long uuid;
};

home HelloHome manages HelloWorld
{
  factory genComp (in long id);
};
```

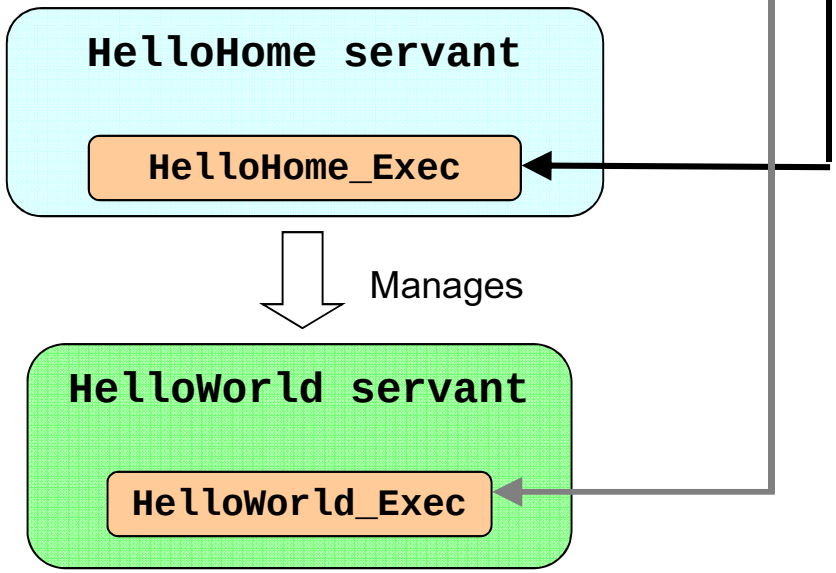
- Facet (`provides`) type maps to local interface, base class, & return type of accessor operation

```
// helloE.idl
#include "hello.idl"

local interface CCM_Goodbye : Goodbye {};

local interface CCM_HelloWorld
  : Components::EnterpriseComponent, Hello {
  CCM_Goodbye get_Farewell ();
  attribute long uuid;
};

local interface CCM_HelloWorld_Context
  : Components::SessionContext {};

local interface CCM_HelloHomeImplicit {
  Components::EnterpriseComponent create ()
    raises (Components::CCMException);
};

local interface CCM_HelloHomeExplicit
  : Components::HomeExecutorBase {
  Components::EnterpriseComponent genComp (in long id);
};

local interface CCM_HelloHome
  : CCM_HelloHomeExplicit, CCM_HelloHomeImplicit {};
```

- Facet name maps to accessor operation
- `attribute` maps with no change to component executor IDL
- `factory` declaration maps to implicit (base class) return type
- Factory name maps to IDL operation
- Factory parameters map with no change

Implementing **HelloWorld** Executor (hello_exec.*)

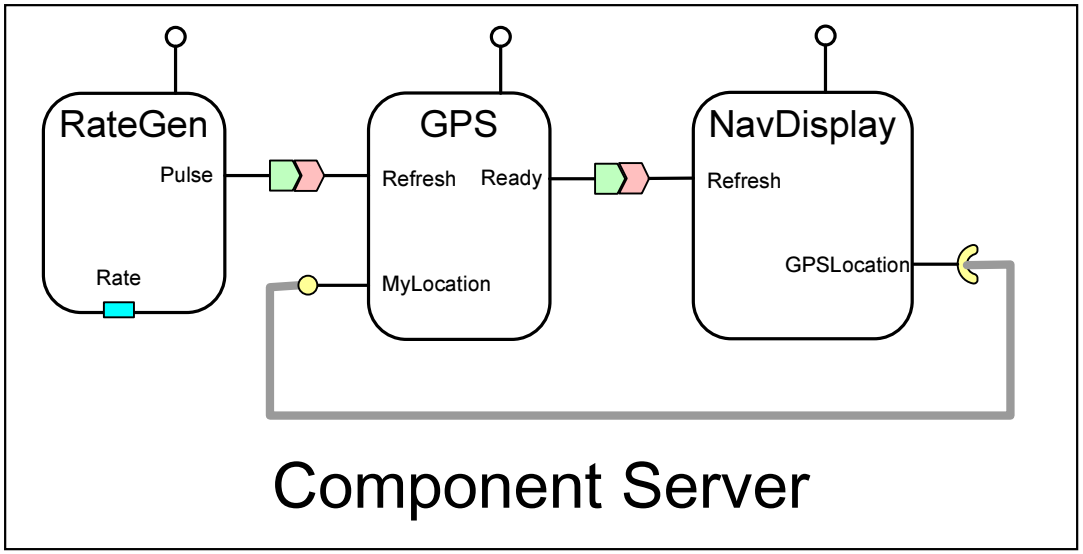
```
// hello.cdl
#include “hello.idl”
composition session Hello_Example
{
    home executor HelloHome_Exec
    {
        implements HelloHome;
        manages HelloWorld_Exec;
    }
};
```



- An executor is where a component/home is implemented
  - The component/home’s servant forwards a client’s *business logic* request to component’s executor
- Developers subclass & implement the following ***_Exec local** interfaces generated by CIDL:
  - **HelloHome_Exec**
  - **HelloWorld_Exec**
- *Our* (CIAO’s) convention is to give these executor implementations stylized names, such as
  - **HelloHome_Exec_Impl**
  - **HelloWorld_Exec_Impl**

Example 2: Heads Up Display (HUD)

Rate Generator | Positioning Sensor | Displaying Device

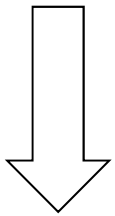


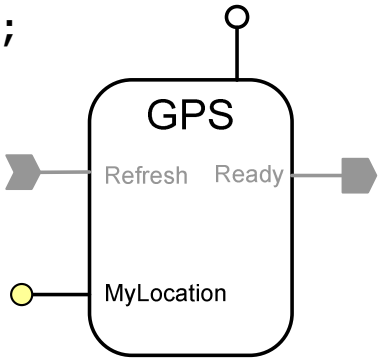
- Component developers must implement
  - Executors for "provided" ports that are invoked by its clients
    - Facets
    - Event sinks
  - Executors that invoke operations on the component's "required" ports
    - Receptacles
    - Event sources

This is the majority of the code implemented by component developers!

Implementing GPS Facet Local Interface

```
// IDL 3
interface position
{
  long get_pos ();
};
component GPS
{
  provides position
           MyLocation;
  …
};
```






```
// Equivalent IDL 2
interface GPS :
  Components::CCMObject
{
  position
    provide_MyLocation ();
  …
};
```

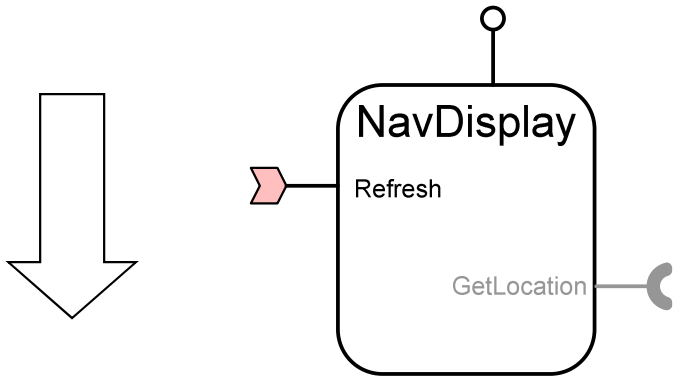
```
// Executor IDL generated by CIDL compiler
local interface CCM_position : position {};
local interface GPS_Exec :
  CCM_GPS,
  Components::SessionComponent
{
  CCM_position get_MyLocation ();
};
```

```
// Implemented by executor developers
class position_Exec_Impl :
  public CCM_position, … {
  virtual ::CORBA::Long get_pos ()
  { return cached_current_location_; }
};

class GPS_Exec_Impl :
  public virtual GPS_Exec,
  public virtual ::CORBA::LocalObject {
public:
  virtual CCM_position_ptr
  get_MyLocation ()
  { return new position_Exec_Impl; }
};
```

Factory method

NavDisplay Component Event Sink

```
// IDL 3
component NavDisplay
{
 …
 consumes tick Refresh;
};
```



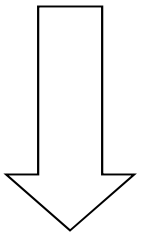

```
// Equivalent IDL 2
interface NavDisplay :
  Components::CCMObject
{
  …
  tickConsumer get_consumer_Refresh ();
  …
};
```

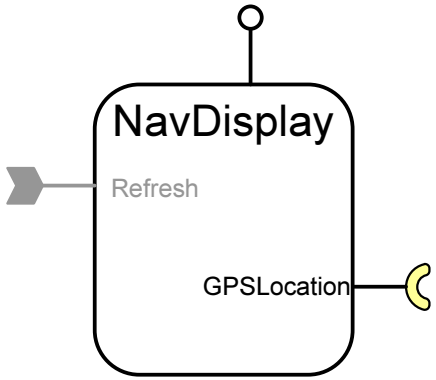
- Components can be connected to event consumer interfaces, similar to facets
- CIDL generates event consumer servants
- Executor mapping defines typed push operations directly

```
class NavDisplay_Exec_Impl :
  public virtual NavDisplay_Exec,
  public virtual ::CORBA::LocalObject {
public:
  ...
  virtual void push_Refresh (tick *ev) {
    // Call a user-defined method
    // (see next page) to perform some
    // work on the event.
    this->refresh_reading ();
  }
  ...
};
```

Using NavDisplay Receptacle Connections

```
// IDL 3
component NavDisplay
{
 …
 uses position GPSLocation;
 …
};
```



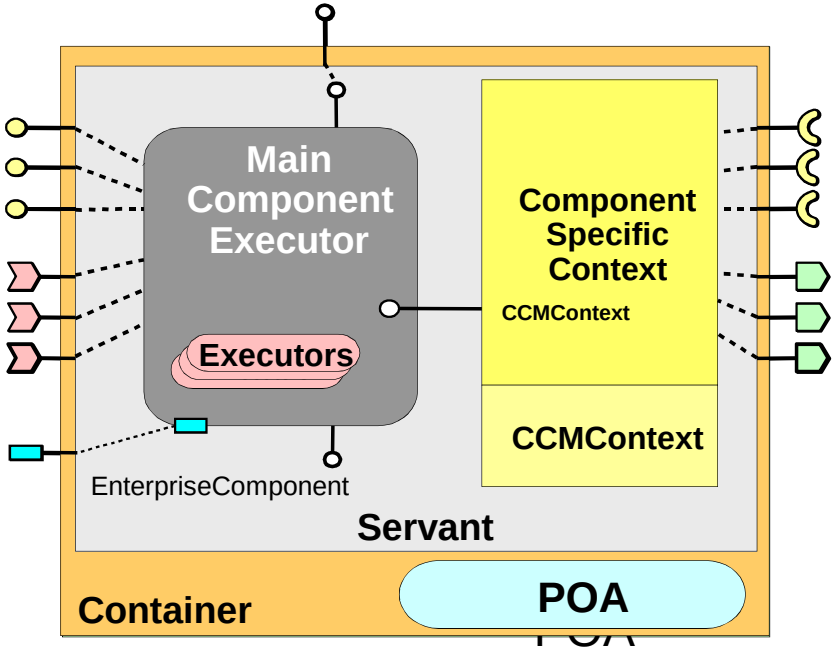



```
// Equivalent IDL 2
interface NavDisplay :
  Components::CCMObject
{
 …
 void connect_GPSLocation (in position c);
 position disconnect_GPSLocation();
 position get_connection_GPSLocation ();
 …
};
```

- Component-specific context manages receptacle connections
- Executor acquires its connected receptacle reference from its component-specific context

```
class NavDisplay_Exec_Impl :
  public virtual NavDisplay_Exec,
  public virtual ::CORBA::LocalObject {
public:
  ...
  virtual void refresh_reading (void) {
    // Local call
    position_var cur =
      this->context_->
        get_connection_GPSLocation ();
    // Remote call
    long coord = cur->get_pos ();
    ...
  }
  ...
};
```

Initializing NavDisplay Component Context



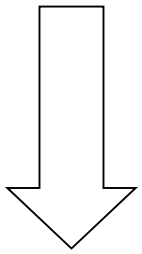
- Calls to set context information are invoked automatically as *callbacks* from containers during deployment
- Component developers implement these callbacks in their executor code

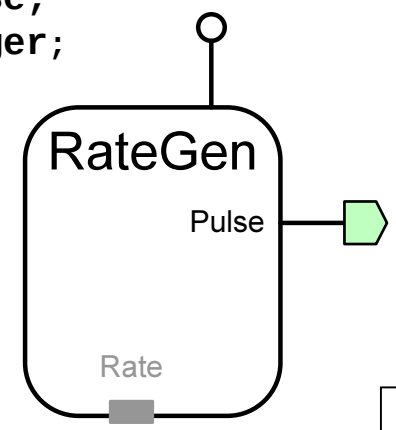
- Component-specific context manages connections & subscriptions
- Container passes component its context via callbacks, e.g.
  - **`set_session_context()`**
  - **`set_entity_context()`**

```
class NavDisplay_Exec_Impl :
  public virtual NavDisplay_Exec,
  public virtual ::CORBA::LocalObject {
private:
  CCM_NavDisplay_Context_var context_;
public:
  ...
  // Called back by container
  void set_session_context
    (Components::SessionContext_ptr c) {
    this->context_ =
      CCM_NavDisplay_Context::_narrow (c);
  }
  ...
};
```

Pushing Events from a RateGen Component

```
// IDL 3
component RateGen
{
  publishes tick Pulse;
  // emits tick Trigger;
  …
};
```






```
// Equivalent IDL 2
interface RateGen :
  Components::CCMObject
{
  Components::Cookie
    subscribe_Pulse
    (in tickConsumer c);
  tickConsumer
    unsubscribe_Pulse
    (in Components::Cookie ck);

  …
};
```

- Component-specific context also
  - Manages consumer subscriptions (for publishers) & connections (for emitters)
  - Provides the event pushing operations & relays events to consumers

```
class RateGen_Exec_Impl :
  public virtual RateGen_Exec,
  public virtual ::CORBA::LocalObject {
public:
  ...
  virtual void send_pulse (void) {
    tick_var ev = new tick;
    this->context_->push_Pulse (ev.in ());
  }
  ...
};
```

Runs in a loop at **`rateHz Rate`**

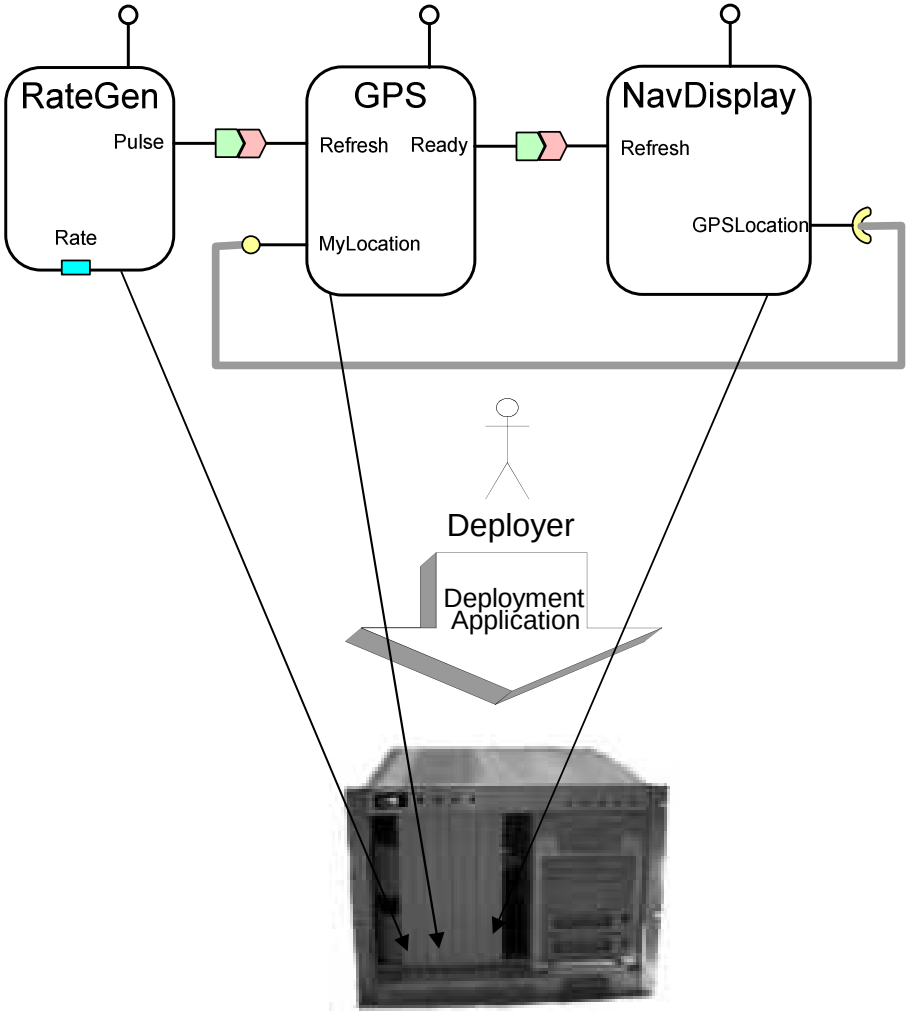
Summary of Server OMG IDL Mapping Rules

- A **`component`** type is mapped to three **`local`** interfaces that correspond to different component roles/ports
  - The *component executor interface*
    - Inherits from **`Components::EnterpriseComponent`** & provides operations for attributes, supported interfaces, & receiving events
  - A *facet executor interface*
    - Operations to obtain facet object references
  - The *component-specific context interface*
    - Operations to publish events & access component receptacles
- A **`home`** type is mapped to four **`local`** interfaces
  - An *explicit executor* interface for user-defined operations
    - Inherits from **`Components::HomeExecutorBase`**
  - An *implicit executor* interface for **`create()`** operation
  - A *main executor* interface inheriting from both previous interfaces
  - A *composition executor* interface inheriting from the main executor interface

We explored all of these mappings in detail in previous slides

Component Packaging, Assembly, & Deployment

Overview of Deployment & Configuration Process



- Goals
  - Ease component reuse
  - Build complex applications by assembling existing components
  - Deploy component-based application into heterogeneous domain(s)
- Separation of concerns & roles
  - Component development & packaging
  - Application assembly
  - Application configuration
  - Application deployment
  - Server configuration

Component Configuration Problem

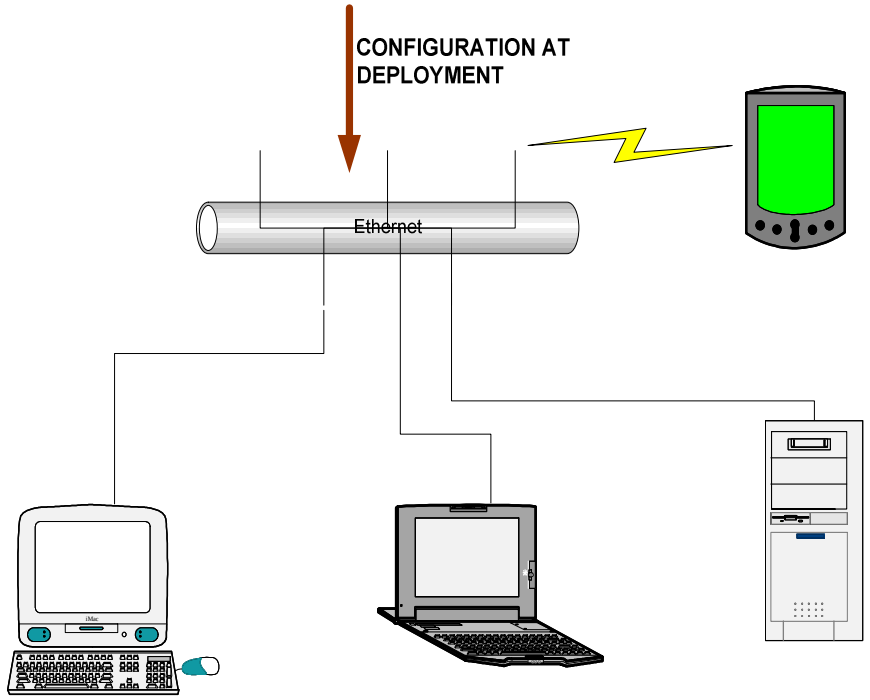
Component middleware & applications are characterized by a large *configuration space* that maps known variations in the *application requirements space* to known variations in the *solution space*

- Components interact with other software artifacts & environment to achieve specific functions
  - e.g., using a specific run-time library to encrypt & decrypt data
- Some prior knowledge of the run-time environment may be required during development
  - e.g., rates of certain tasks based on the functional role played
- Need to configure the middleware for specific QoS properties
  - e.g., transport protocols, timeouts, event correlation, concurrency/synchronization models, etc.
- Adding environment & interaction details with the business logic leads to overly tight coupling
  - e.g., tightly coupled code leads to poor reusability & limited QoS

CCM Configuration Concept & Solution

Concept

- Configure run-time & environment properties late in the software lifecycle, i.e., during the deployment process



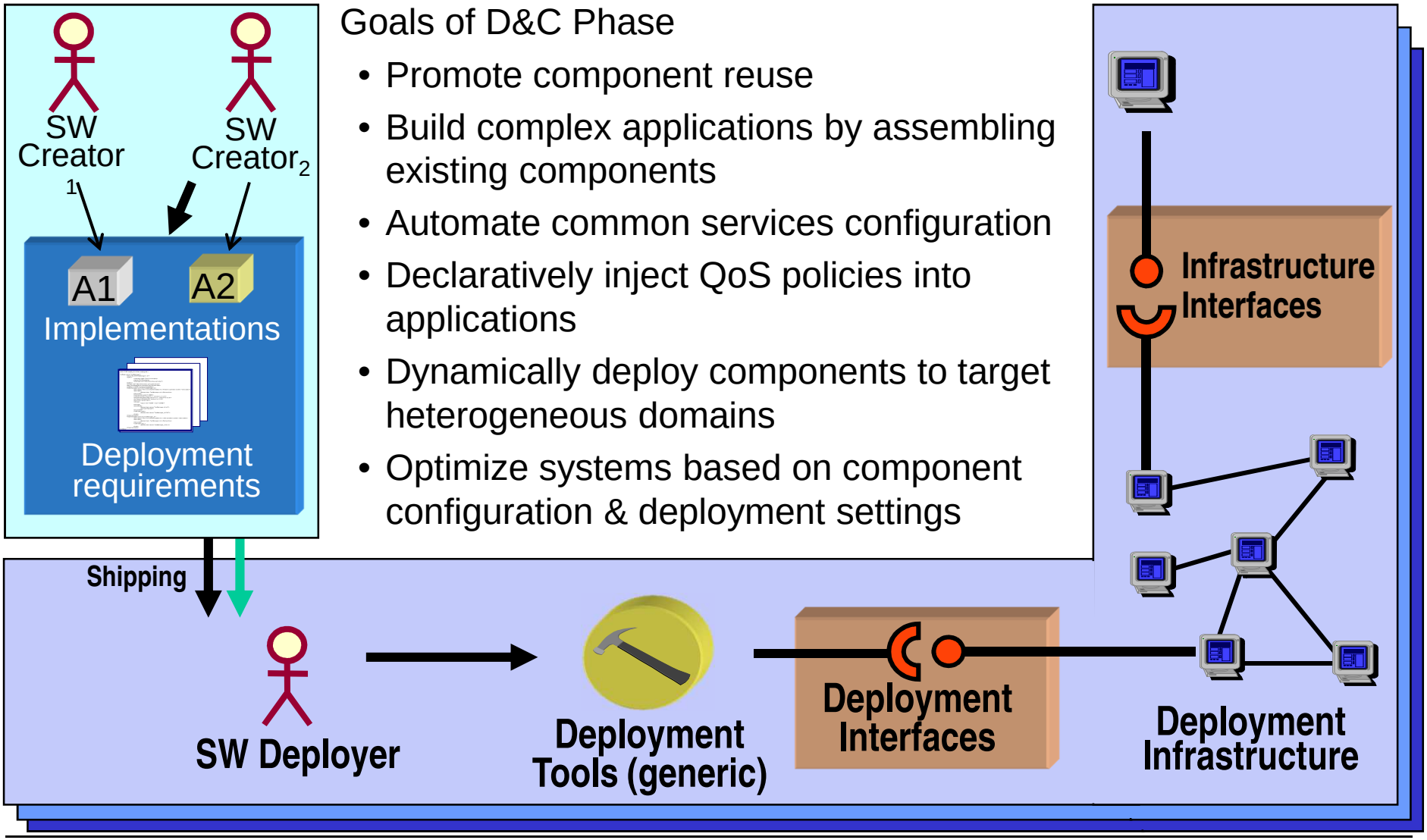
Solution

- **Well-defined exchange formats** to represent configuration properties
  - Can represent a wide variety of data types
  - Well-defined semantics to interpret the data
- **Well-defined interfaces** to pass configuration data from “off-line” tools to components
- **Well-defined configuration boundary** between the application & the middleware

Component Deployment Problem

- Component implementations are usually hardware-specific
  - Compiled for Windows, Linux, Java – or just FPGA firmware
  - Require special hardware
    - e.g., GPS sensor component needs access to GPS device via a serial bus or USB
    - e.g., Navigation display component needs … a display
      - not as trivial as it may sound!
- However, computers & networks are often heterogeneous
  - Not all computers can execute all component implementations
- The above is true for each & every component of an application
  - i.e., each component may have different requirements

OMG Component Deployment & Configuration Spec (1/2)

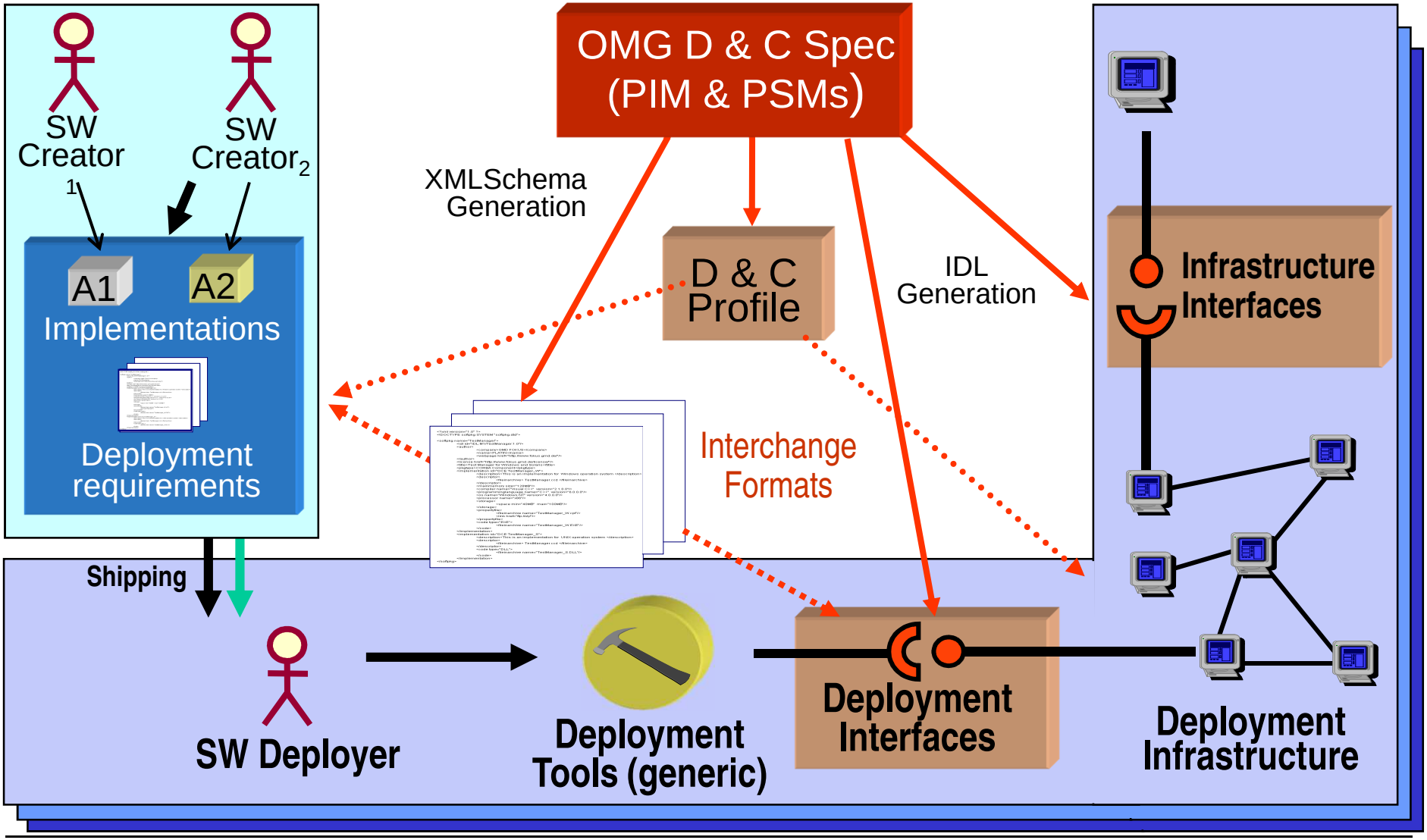


Goals of D&C Phase

- Promote component reuse
- Build complex applications by assembling existing components
- Automate common services configuration
- Declaratively inject QoS policies into applications
- Dynamically deploy components to target heterogeneous domains
- Optimize systems based on component configuration & deployment settings

OMG Deployment & Configuration (D&C) specification (ptc/05-01-07)

OMG Component Deployment & Configuration Spec (1/2)



OMG Deployment & Configuration (D&C) specification (ptc/05-01-07)

CCM Deployment Solution

- **Well-defined exchange format**
  - Defines what a software vendor delivers
  - Requires "off-line" data format that can be stored in XML files
- **Well-defined interfaces**
  - Infrastructure to install, configure, & deploy software
  - Requires "on-line" IDL data format that can be passed to/from interfaces
- **Well-defined software metadata model**
  - Annotate software & hardware with interoperable, vendor-independent, deployment-relevant information
  - Generate "on-line" & "off-line" data formats from models
    - e.g., CoSMIC at www.dre.vanderbilt.edu/cosmic

Overview of Lightweight CCM Specification

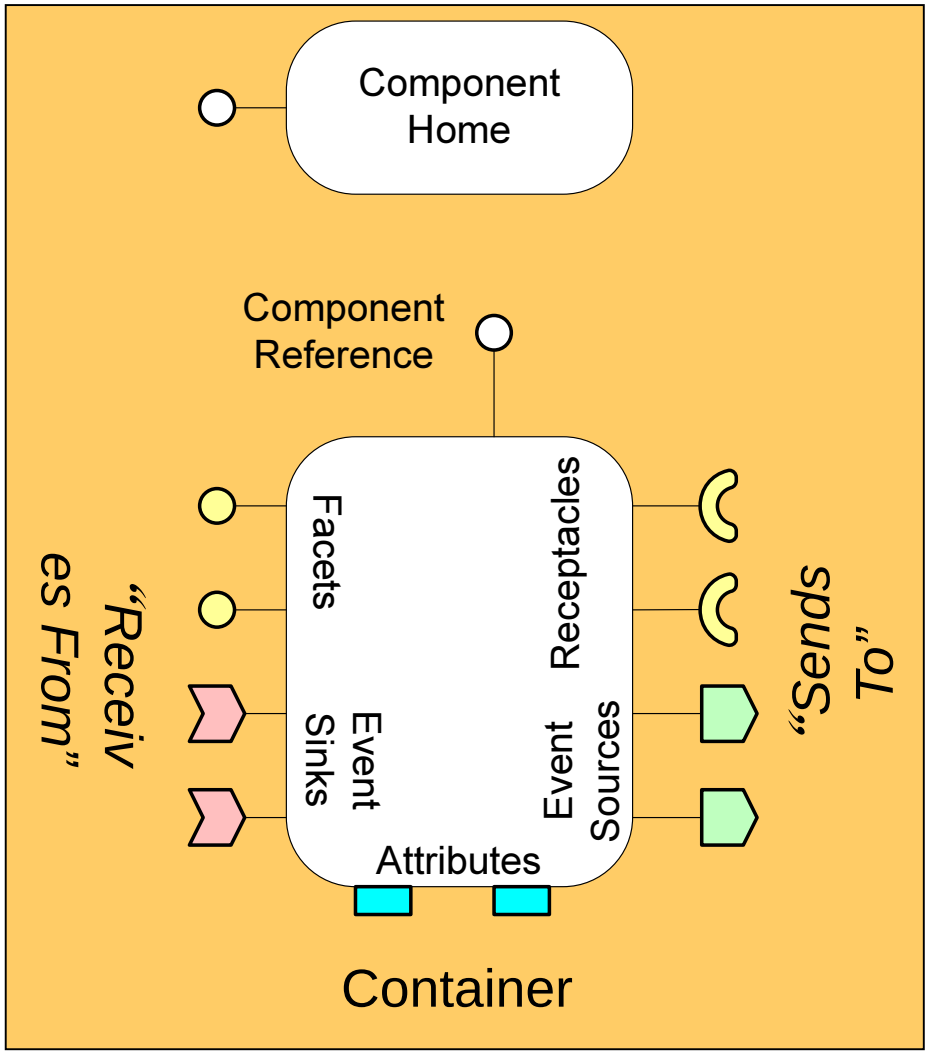
www.omg.org/cgi-bin/doc?realtime/2003-05-05

Motivation for Lightweight CCM (LwCCM)

- Many DRE CORBA applications can't use "enterprise" CCM due to constraints
  - e.g., small code size in embedded environments & limited processing overhead for performance-intensive applications
- These constrained environments need "lightweight" CCM functionality
- ORB vendors, or other third-party vendors, can then support this lightweight version in a standard package
- In the *Lightweight CCM* specification, each section is explicitly treated & either retained as is, profiled, or removed

CCM Features Retained in LwCCM Subset

- All types of ports, i.e.,
  - Facets
  - Receptacles
  - Event sources & sinks
  - Attributes
- Component homes
- Generic port management operations in **`CCMObject`**
- Monolithic implementations
- Session/service component/ container types



CCM Features Excluded from LwCCM Subset

- Keyed homes
  - Large overhead & complexity
- Process & Entity containers
  - Persistence often not relevant in DRE systems domain
- Component segmentation
  - Unnecessary with introduction of D&C
- CIDL
  - May not be needed after removal of PSDL & segmentation
  - IDL 3 may be sufficient
- **`CCMObject`** introspection
  - Useful in managing dynamic applications & debugging
  - Debugging can be done in full CCM
  - Application management can be done using D&C
  - Dynamic applications often not relevant in DRE systems domain
- Equivalent IDL for port management
  - Redundant, can use generic port operations
  - Generic interface is required for D&C

Lightweight CCM should be treated like Minimum CORBA, i.e., *advisory*

IDL 3+

Grouping Related Services

- **Context** - Components may have one or more facets or receptacles that are related as part of the implementation of a cohesive service
- **Problem**
  - No semantic information about these groupings are captured in IDL
  - Makes it difficult and error prone to plan deployments that are correct by construction
- **Solution** – Create an 'extended' port type that may be offered by a component
  - Requires new IDL keywords to define a grouping of related facets and receptacles
  - Provides for parameterization for services that may be generic with respect to its data type.

IDL3+

- **Extension of the IDL3 language**
  - Defined in the DDS for Lightweight CCM specification (DDS4CCM)
  - Provides an IDL3 equivalent mapping so the new constructs can be converted to IDL3
- **Provides new keywords for**
  - Specifying grouping of related facets/receptacles (*porttype)*
  - Declaring ports within a component (*port, mirrorport*)
  - Support for new entity intended to provide “glue” between several ports (*connector)*
  - A template syntax for parameterized modules

Fixed Extended Ports

```
interface Data_Pusher {
  void push(in Data dat);
};
interface FlowControl {
  void suspend ();
  void resume();
  readonly attribute nb_waiting;
};

// extended port definition
porttype ControlledConsumer {
  provides Data_Pusher consumer;
  uses FlowControl control;
};
// Component supporting a port
component Sender{
  port ControlledConsumer data_out;
};
```

Facets/Receptacles are declared in the same way as a regular component

Port and mirrorport are used to define opposite ends of the connection.

```
component Receiver {
   mirrorport ControlledConsumer data_in;
};
```

Fixed Extended Port Equivalent IDL3

```
// extended port definition
porttype ControlledConsumer {
  provides Data_Pusher consumer;
  uses FlowControl control;
};
// Component supporting a port
component Sender{
  port ControlledConsumer data;
};
```

```
component Sender{
  provides Data_Pusher data_consumer;
  uses FlowControl data_control;
};
```

```
component Receiver{
  mirrorport ControlledConsumer input;
};
```

```
component Receiver{
  uses Data_Pusher input_consumer;
  provides FlowControl input_control;
};
```

Facets/Receptacles map directly into the component in place of port declaration

Name of port prepended to new facet/receptacle names

Mirrorport keyword inverts facets and receptacles

Parameterized Extended Port

- Some IDL construct may vary in only the data type used as a parameter
- Need to avoid proliferation of type-specific port type declarations
- Syntax similar to C++ template programming
- All parameterized types are contained in a typed module
- Typed modules is instantiated

```
module Typed <typename T>{
  interface Pusher {
    void push(in T data);
  };
  porttype Consumer {
    provides Pusher consumer;
    uses FlowControl control;
  };
};
module Typed < Data> DataModule;
component Sender {
   port DataModule::Consumer data;
};
component Receiver {
  mirrorport DataModule::Consumer data;
}
```

Parameterized Port Equivalent IDL3

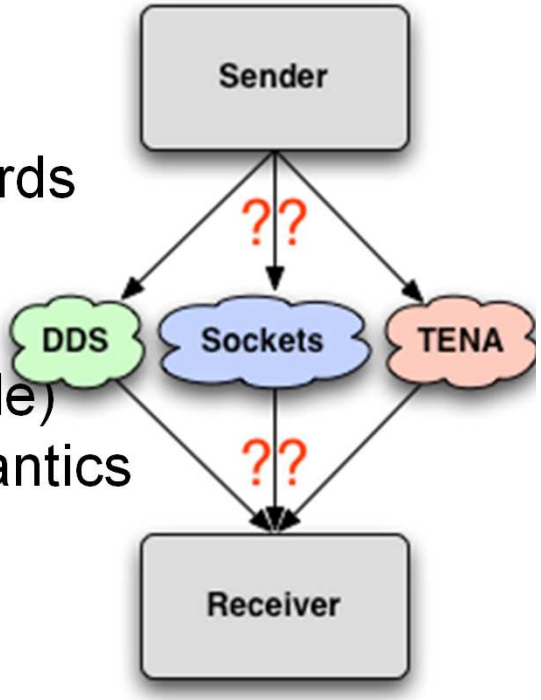
```
module Typed <typename T>{
  interface Pusher {
    void push(in T data);
  };
  porttype Consumer {
    provides Pusher consumer;
    uses FlowControl control;
  };
};
module Typed < Data> DataModule;
component Sender {
   port DataModule::Consumer data;
};
```

```
module DataModule
{
  interface Pusher {
    void push(in Data data);
  };
  porttype Consumer {
    provides Pusher consumer;
    uses FlowControl control;
  };

component Sender {
  provides DataModule::Pusher
   data_consumer;
  uses FlowControl data_control
};
```

CONNECTORS

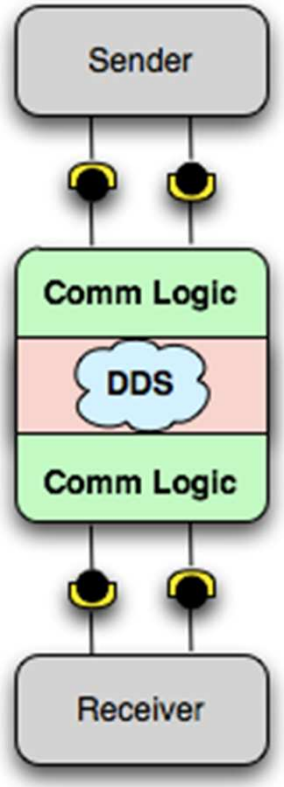
Motivating Connectors (1/2)

- **Context** – Many applications must leverage non-CORBA communication mechanisms
  - Interfacing with legacy code
  - Complying with mandated architectural standards
  - High level systems integration
- Problem
  - Existing CCM ports (pub/sub or facet/receptacle) may not properly capture communication semantics
  - Addition of new communication middleware/semantics requires
    - Mixing communication logic with business logic
    - Modifying the container to extend/change semantics of existing ports
  - Precludes use of D&C middleware for configuration



Motivating Connectors (2/2)

- **Solution** – Create a separate, deployable entity to contain communication logic
  - Leverage extended ports to create well-defined interfaces
  - Can be re-used across different applications
  - Allows D&C infrastructure to coordinate configuration
- Implemented using new *connector* IDL3+ keyword
  - Collects ports (regular and extended) into a coherent interface
  - Can have attributes which may be used for D&C
  - Can be placed in a parameterized module



Defining Connectors

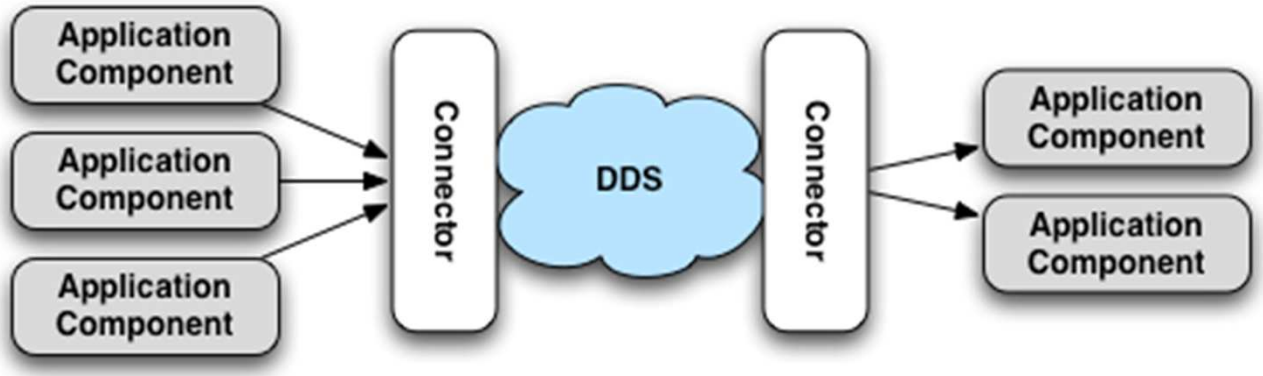
- Fixed and parameterized connectors are possible
- Connectors support ports, attributes, and inheritance

```
connector Cnx {
  mirrorport Data_ControlledConsumer cc;
  provides Data_Pusher p;
  attribute string configuration;
};
module Typed <typename T> {
  connector Cnx {
    port ControlledConsumer cc;
    mirrorport Pusher p;
    attribute T configuration;
  };
};
```

Implementing and Deploying Components

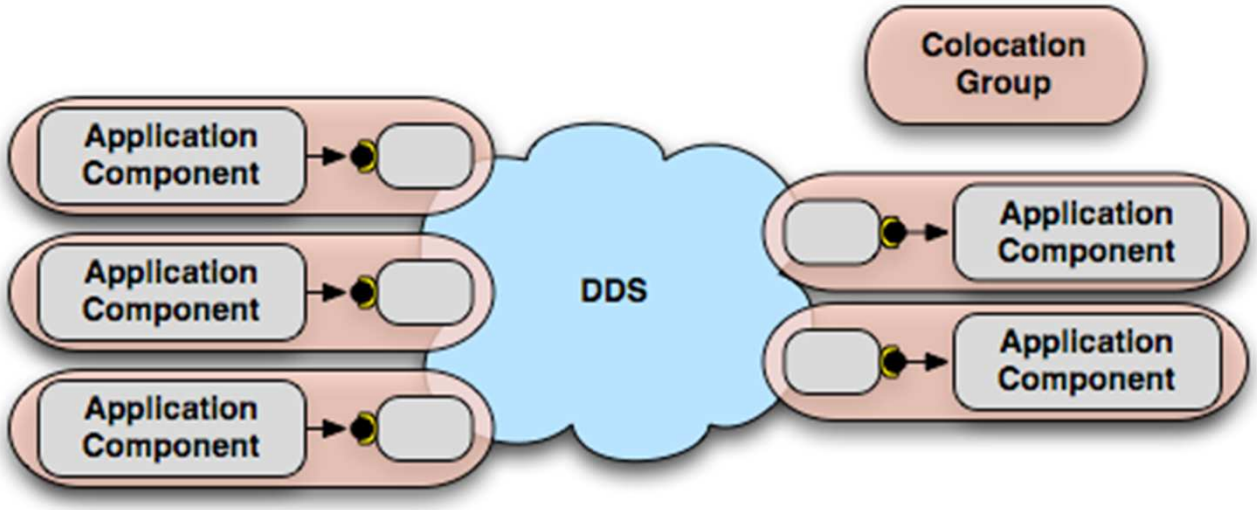
- Connectors may be divided into fragments
  - Each fragment is co-located with the component instance it is connected to
  - One or more fragments may be associated with a particular component
- Connector fragment is derived from CCMObject
  - Connector fragment can be deployed as a regular component
  - Configuration takes place via standard attributes defined in the interface
  - Fragment must implement Navigation, Receptacles, and, KeylessCCMHome

Connector Modeling vs. Deployment (1/2)



- From a modeling standpoint, connectors appear monolithic
- Need not be concerned with ‘fragments’ or how they are deployed
- Connections are made from a component to the desired port on the connector

Connector Modeling vs. Deployment (2/2)



- The modeling tool will generate descriptors for the appropriate fragments
- Each application component is deployed collocated with a dedicated fragment for communication
- Application components and their connectors communicate over *local* interfaces

Connector fragment instantiation

- Application components cannot share connector fragment instances
- An application component that defines multiple extended port connectors to the same unique connector should utilize a single connector fragment instance to connect
- A single extended port of an application component cannot connect to more than one connector fragment
- When a model uses a connection to an extended port the generated CDP file will connect all basic ports
- When a receptacle is tagged as “asynchronous” a collocated AMI4CCM connector fragment is only deployed with the component that has the client port
- For each connection as part of a uses multiple an ami4ccm connector fragment is deployed

DDS FOR LIGHTWEIGHT CCM

DDS for Lightweight CCM (1/3)

- The Data Distribution Service (DDS) provides robust and high performance communication
  - Accommodates any flavor of pub/sub communication
  - Rich API for configuring behavior and Quality of Service (QoS)
- Flexibility and robustness comes at the price of increased complexity
  - Configuration can be tedious and error-prone
  - Developers must write boilerplate code to bootstrap and configure their application
  - Must develop ad-hoc and proprietary ways to store and apply configuration data

The DDS for Lightweight CCM (DDS4CCM) resolves these challenges

DDS for Lightweight CCM (2/3)

- Provides a simpler API to the application developer
  - Completely removes configuration from the scope of the application developer
  - Defines ready-to-use ports intended to hide complexity
  - Well-defined DDS patterns are codified in connectors with associated QoS settings
- Provides robust deployment and configuration support to DDS
  - Provides a container (via CCM) to perform application bootstrapping and configuration
  - Application binary distribution to distributed, heterogeneous domains
  - Coordinated application startup and teardown across distributed nodes

DDS for Lightweight CCM (3/3)

- Specification tries not to prevent advanced DDS usage
  - All ports provide access to a more detailed interface
  - All involved DDS entities can be discovered using this starting point

DDS4CCM Basic Ports (1/3)

- The basic ports are grouped into three categories: *Data Access – Publishing, Data Access – Subscribing,* and *Status Access*
- **Data Access – Publishing**
  - **Writer** – Allows publication of data on a topic without regard to the instance lifecycle.
  - **Updater**– Allows publication of data with management of instance lifecycle. Allows creation, update, and deletion of instances.

DDS4CCM Basic Ports (2/3)

- **Data Access – Subscribing**
  - **Reader** – Allows access to one or more instances with non-blocking semantics.
  - **Getter** – Allows access to one or more instances with blocking semantics.
  - **Listener** – Provides a callback mechanism to the application when new data arrives, regardless of instance state
  - **StateListener** – Provides a callback mechanism to the application when new data arrives, with different operations depending on state

DDS4CCM Basic Ports (3/3)

- **Status Access**
  - **PortStatusListener** – Delivers status related to ports, this information is relevant to data subscribers.
  - **ConnectorStatusListener** – Delivers status updates that are relevant system-wide.

DDS4CCM Extended Ports (1/2)

- The extended ports – in most cases – combine a basic port with the corresponding DCPS IDL interface
  - Provides the opportunity to access advanced DDS features
  - Increases code portability by not exposing DDS implementation directly
  - Subscriber ports also include a PortStatusListener

```
porttype DDS_Write {
  uses Writer data;
  uses DDS::DataWriter dds_entity;
};
```

```
porttype DDS_Read {
  uses Reader data;
  attribute QueryFilter filter;
  uses ContentFilterSetting filter_config;
  uses DDS::DataReader dds_entity;
  provides PortStatusListener status;
};
```

DDS4CCM Extended Ports (2/2)

- Listener ports (updates are pushed to the component) contain both Reader and Listener ports
  - 'Reader' portion of the extended port is used to configure criteria used to select updates
  - Selected updates are pushed to the component

```
porttype DDS_Listen {
  uses Reader data;
  uses DataListenerControl control;
  provides Listener data_listener;
  attribute QueryFilter filter;
  uses ContentFilterSetting filter_config;
  uses DDS::DataReader dds_entity;
  provides PortStatusListener status;
};
```

```
porttype DDS_StateListen {
  uses Reader data;
  uses ListenerControl data_control;
  provides StateListener data_listener;
  attribute QueryFilter filter;
  uses ContentFilterSetting filter_config;
  uses DDS::DataReader dds_entity;
  provides PortStatusListener status;
};
```

DDS4CCM Standard Connectors (1/2)

- Gathers connectors for all roles in a given use pattern
  - One or more DDS4CCM Extended Ports (as mirrorports)
  - Configuration meta-data (domain ID, topic name, QoS profiles)
- Two standard defined 'base connectors
  - DDS_Base
  - DDS_TopicBase
- Two standard defined connectors
  - Pattern State Transfer
  - Pattern Event Transfer

```
connector DDS_Base {
  uses ConnectorStatusListener
   error_listener;
  attribute DDS::DomainId_t domain_id
    setraises (NonChangable);
  attribute string qos_profile
    setraises (NonChangable);
};

connector DDS_TopicBase : DDS_Base {
  attribute string topic_name
    setraises (NonChangable);
  attribute StringSeq key_fields
    setraises (NonChangable);
};
```

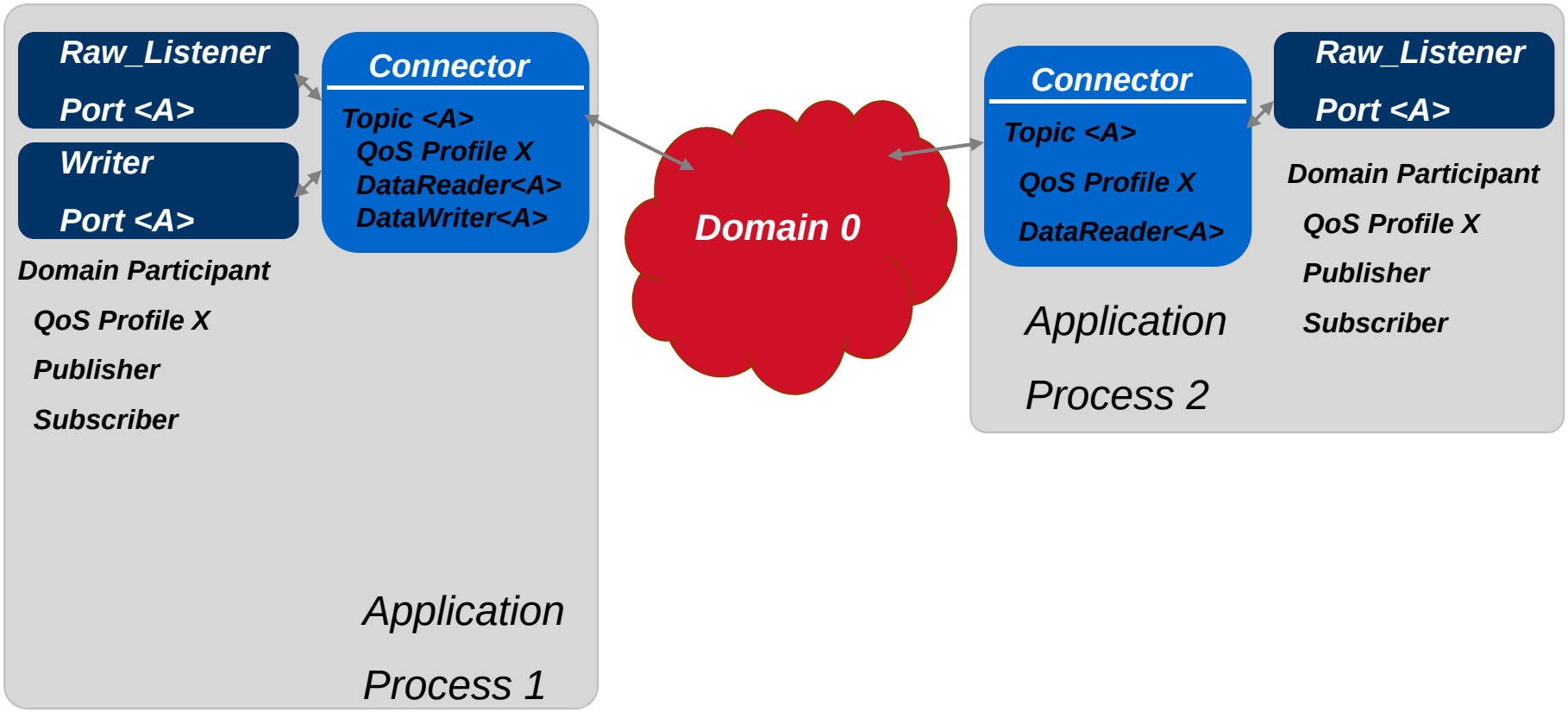
DDS4CCM Standard Connectors (2/2)

- Standard provides two predefined connectors
- Each connector corresponds to a DDS usage pattern
- **Pattern State Transfer**
  - *Observable* – Components that publish state
  - *Observer* – Components that subscribe to that information
- **Pattern Event Transfer**
  - *Supplier* – Components that send events over DDS
  - *Consumer* – Components that subscribe to those events

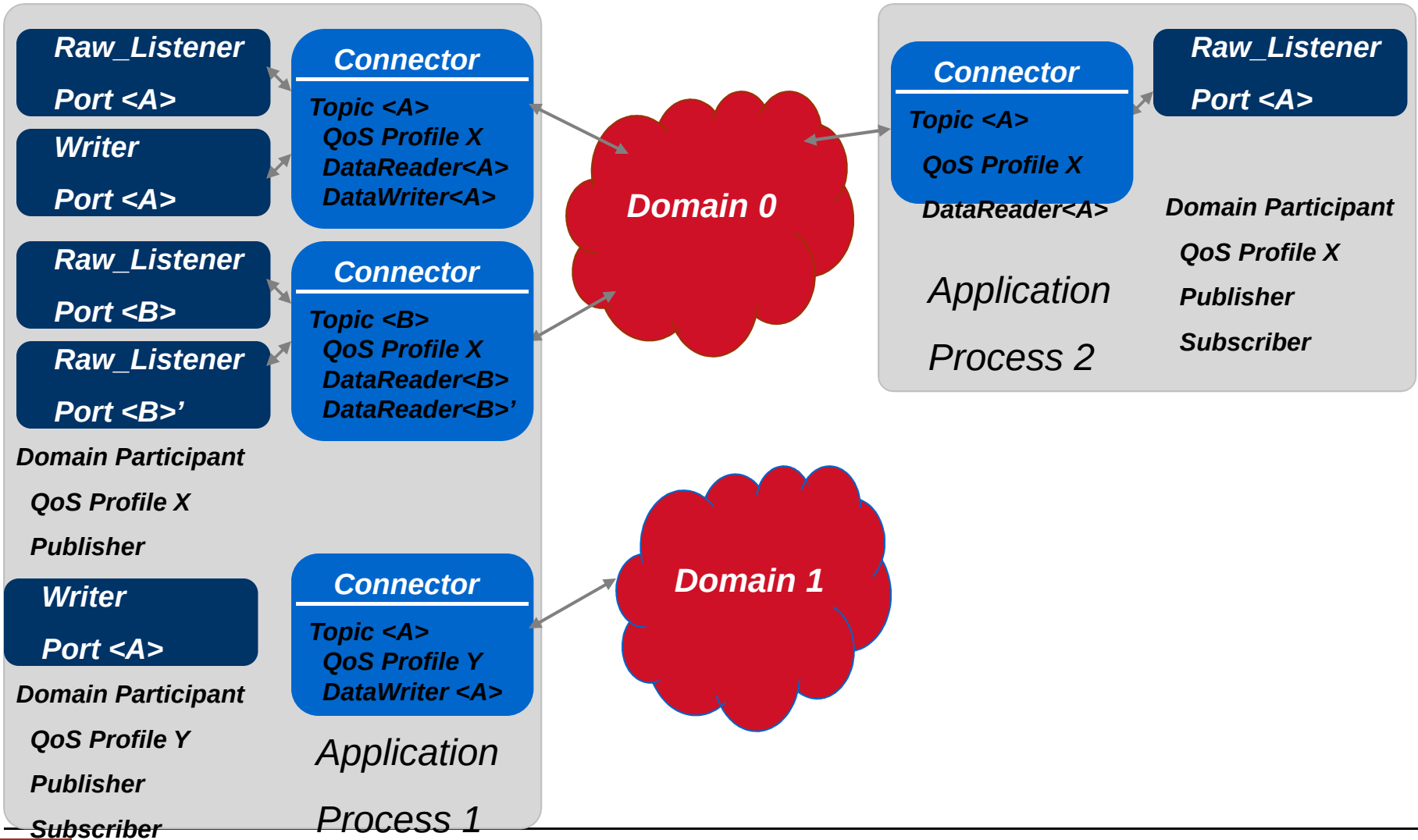
```
connector DDS_State : DDS_TopicBase {
  mirrorport DDS_Update observable;
  mirrorport DDS_Read passive_observer;
  mirrorport DDS_Get pull_observer;
  mirrorport DDS_Listen push_observer;
  mirrorport DDS_StateListen
    push_state_observer
};

connector DDS_Event : DDS_TopicBase {
  mirrorport DDS_Write supplier;
  mirrorport DDS_Get pull_consumer;
  mirrorport DDS_Listen push_consumer;
};
```

End users are free to define connectors more appropriate for their use cases!

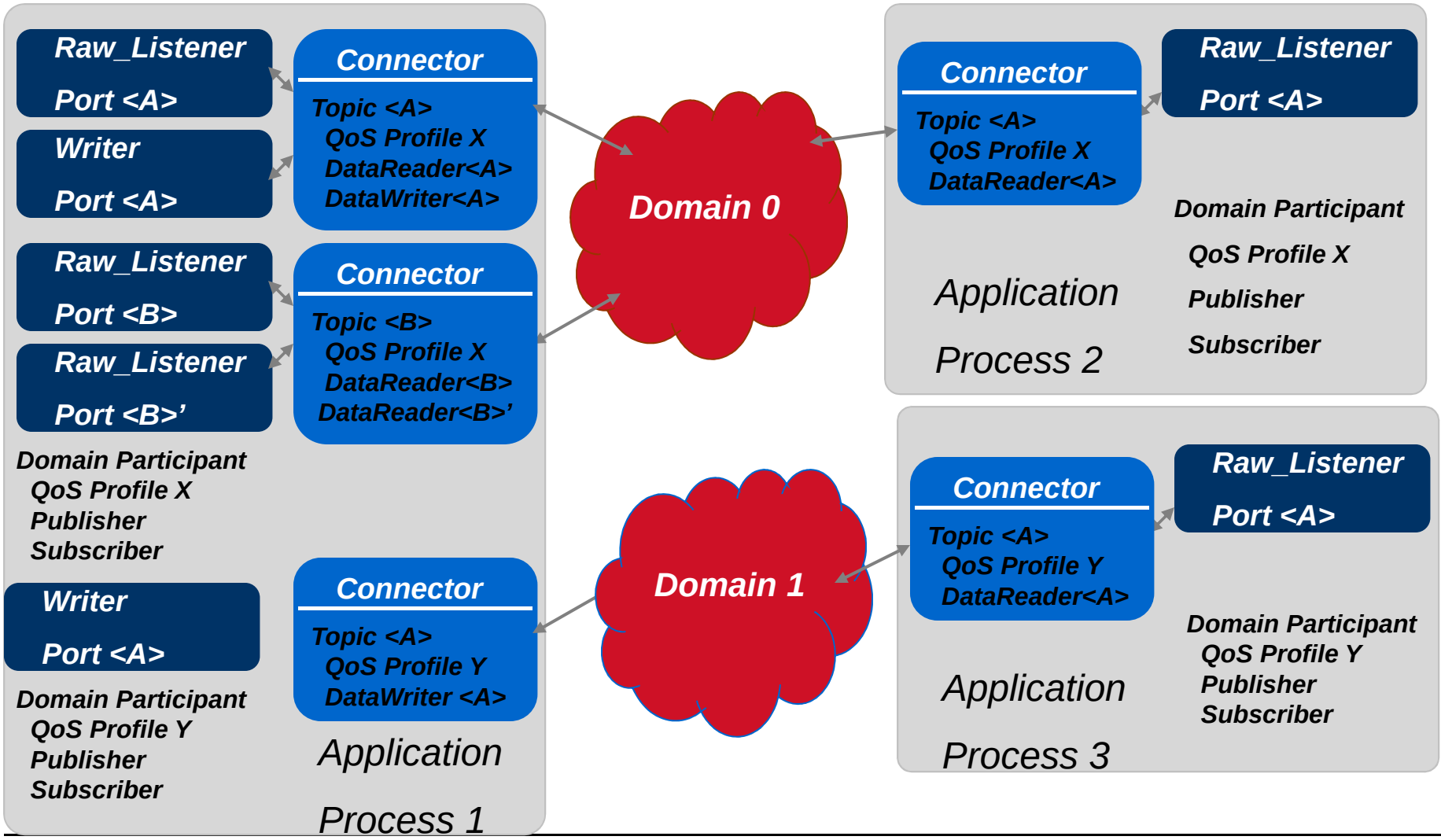
CCM/DDS Entity Mapping (1/3)



CCM/DDS Entity Mapping (2/3)



CCM/DDS Entity Mapping (3/3)



Example

- Initial proof of concept is a derivation of the “Hello, World!” CCM example

- The example uses DDS to communicate a simple string between two components

- Example includes a connector implemented as two fragments to accomplish DDS communication

Hello, World! IDL

```
//Datastruct
struct Hello {
  string msg;
};
```

```
component Sender {
  port Hello::DDS_Write msg;
}
```

```
component Receiver {
  port Hello::DDS_Listen msg;
}
```


```
module ::CCM_DDS::Typed < ::Hello, ::Hello_Seq> Hello;
```

The connector code is completely generated! We will focus on the component code.

Connector IDL Sender

`$CIAO_ROOT/connectors/dds4ccm/idl/ccm_dds.idl`

```
module CCM_DDS {
  module Typed <typename T, sequence<T> TSeq> {
    local interface Writer : InstanceHandleManager {
      void write_one (in T datum, in DDS::InstanceHandle_t instance_handle)
        raises (InternalError);
      /* ……… */
    }
    /* ……… */
    porttype DDS_Write {
      uses Writer data;
      uses DDS::DataWriter dds_entity;
    };
    connector DDS_Event : DDS_TopicBase {
      mirrorport DDS_Write supplier;
      /* ……… */
    };
  };
};
```

Typed module with T=Hello

DDS Writer port. Writer interface is called ‘data’

We’re using an Event connector. The extended DDS_Write port is called ‘supplier’.

Implementing the Sender Component

```
// Equivalent IDL3
module CCM_DDS
{
  module Typed <typename T, sequence<T> TSeq> {
    local interface Writer:InstanceHandleManager
      void write_one (in T datum,
          in DDS::InstanceHandle_t
                    instance_handle)
        raises (InternalError);
      /* … */
    };
  /* … */
  porttype DDS_Write {
    uses Writer data;
    uses DDS::DataWriter dds_entity;
  };
};

module Hello {
  component Sender {
    port Hello::Writer msg;
  };
};
```

A component-specific context maintains connection info

ccm_activate is invoked on a component instance to indicate that it may begin execution

```
void Sender_e    _i::ccm_activate ()
{
  ::Hello::Writer_var writer =
    this->context_->
      get_connection_msg_data ();

    Hello * ne    _msg;
    new_msg.ms    = "Hello World!";
    writer->wr    _one (new_msg,
        ::DDS:    DLE_NIL);}
```

This passes the Hello struct over the local interface to the connector. All DDS setup (domain, QoS, topic, etc) is handled by the connector and is entirely hidden from the application developer

Obtaining the writer interface of the DDS4CCM connector.

Connector IDL Receiver

`$CIAO_ROOT/connectors/dds4ccm/idl/ccm_dds.idl`

```
module CCM_DDS {
  module Typed <typename T, sequence<T> TSeq> {
    local interface Listener {
      void on_one_data (in T datum, in ReadInfo info);
      void on_many_data (in TSeq data, in ReadInfoSeq infos);
    }
    /* ..... */
    porttype DDS_Listen {
      /* ..... */
      provides Listener data_listener;
      /* ..... */
    };
    connector DDS_Event : DDS_TopicBase {
      mirrorport DDS_Listen push_consumer;
      /* ..... */
    };
  };
};
```

Typed module with T=Hello

DDS_Listen port. Listener interface is called 'data_listener'

This time the DDS4CCM connector uses the Receiver component to callback to.

Receiver Equivalent IDL3

```
module Hello {
  component Receiver {
    port Hello::DDS_Listen info_out;
    provides CCM_DDS::ConnectorStatusListener info_out_connector_status;
  };
};
```

Pushes new instance updates to the receiver upon receipt

Provides status information to the receiver (e.g., missed deadlines, lost samples, etc.)

The Hello, World prototype currently implements the Listener port

Implementing the Receiver

```
class Receiver_exec_i
  : public virtual Receiver_Exec,
    public virtual ::CORBA::LocalObject
{
 virtual
 ::Hello::CCM_Listener_ptr
    get_msg_data_listener (void) {
  return new DDSHello_Listener_exec_i ();
  }
};
```

Type-specific Listener Interface

Listener Facet Implementation

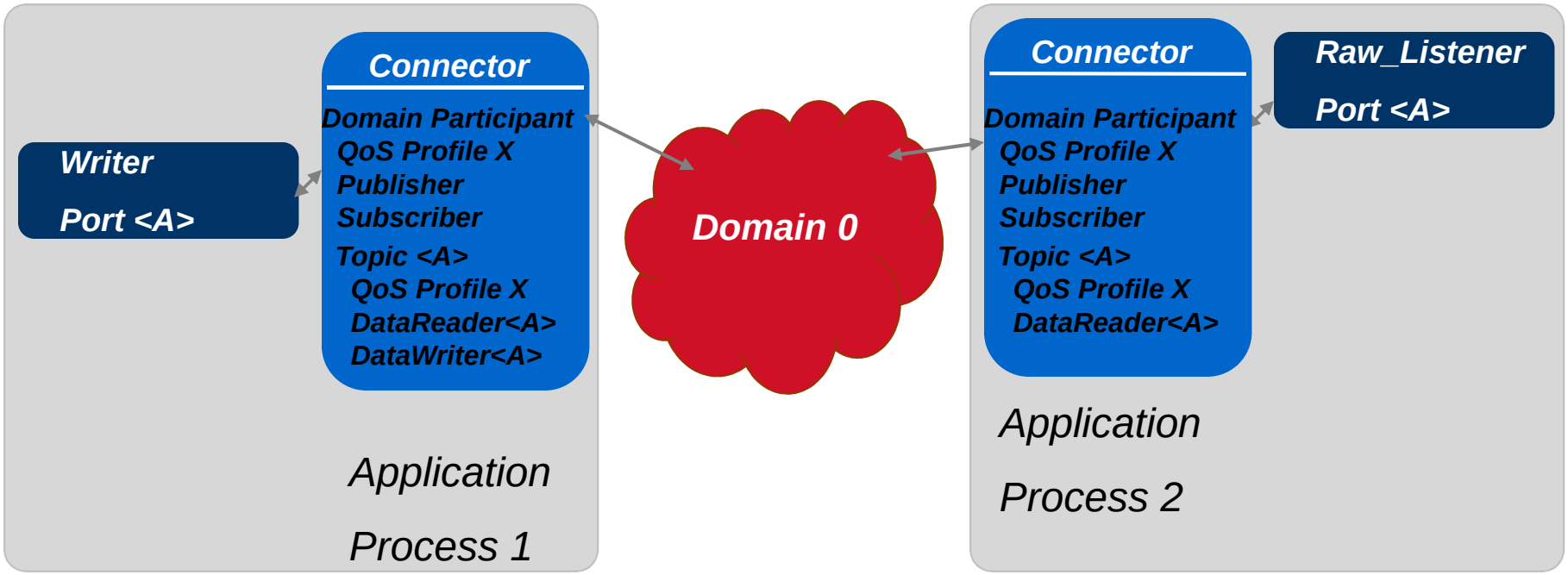
Component executor get_msg_data_listener navigation method. This methods is invoked by the DDS4CCM connector in order to retrieve the callback interface (which the Receiver component provides)

```
class DDSHello_Listener_exec_i
   : public virtual
      ::CCM_DDS::CCM_Listener,
     public virtual
      ::CORBA::LocalObject
{
  public:
  virtual void on_one_data
     (const Hello & an_instance,
      const ::CCM_DDS::ReadInfo & info)
  {
    ACE_DEBUG ((LM_DEBUG,
        "Received message <%C>\n",
         an_instance.msg);
  }
};
```

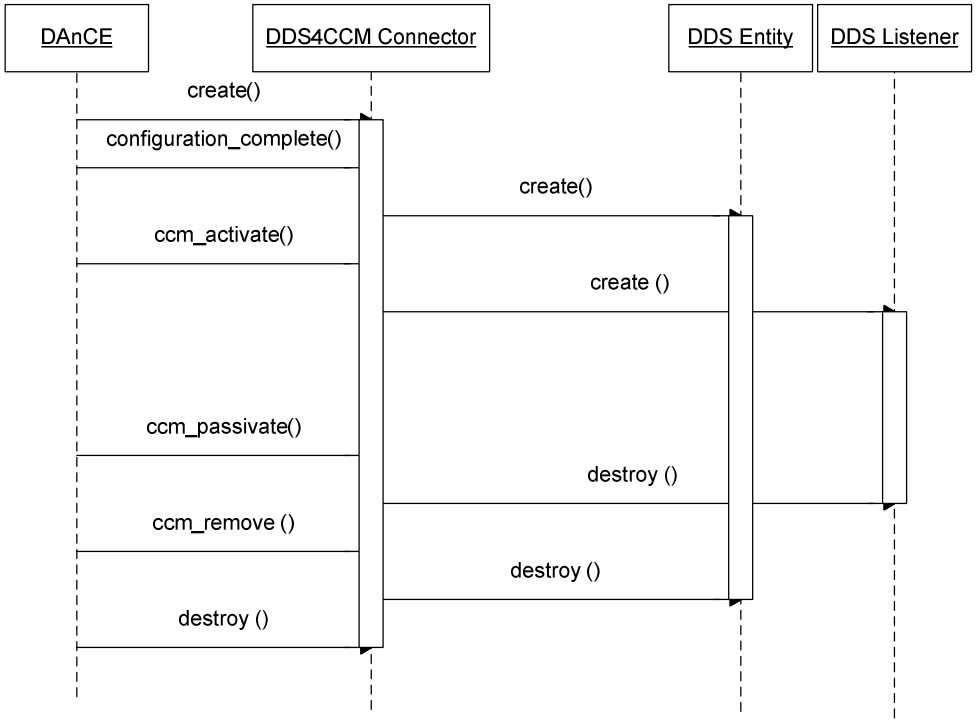
Hello, World Prototype

- $CIAO_ROOT/connectors/dds4ccm/examples/Hello
- A full implementation has been created
  - Sender component and connector fragment
  - Receiver component and connector fragment
  - Non-CCM DDS sender
  - Non-CCM DDS receiver
- Multiple test scenarios
  - One to one
  - Many to one
  - One to many
  - Interoperability with non-CCM DDS programs

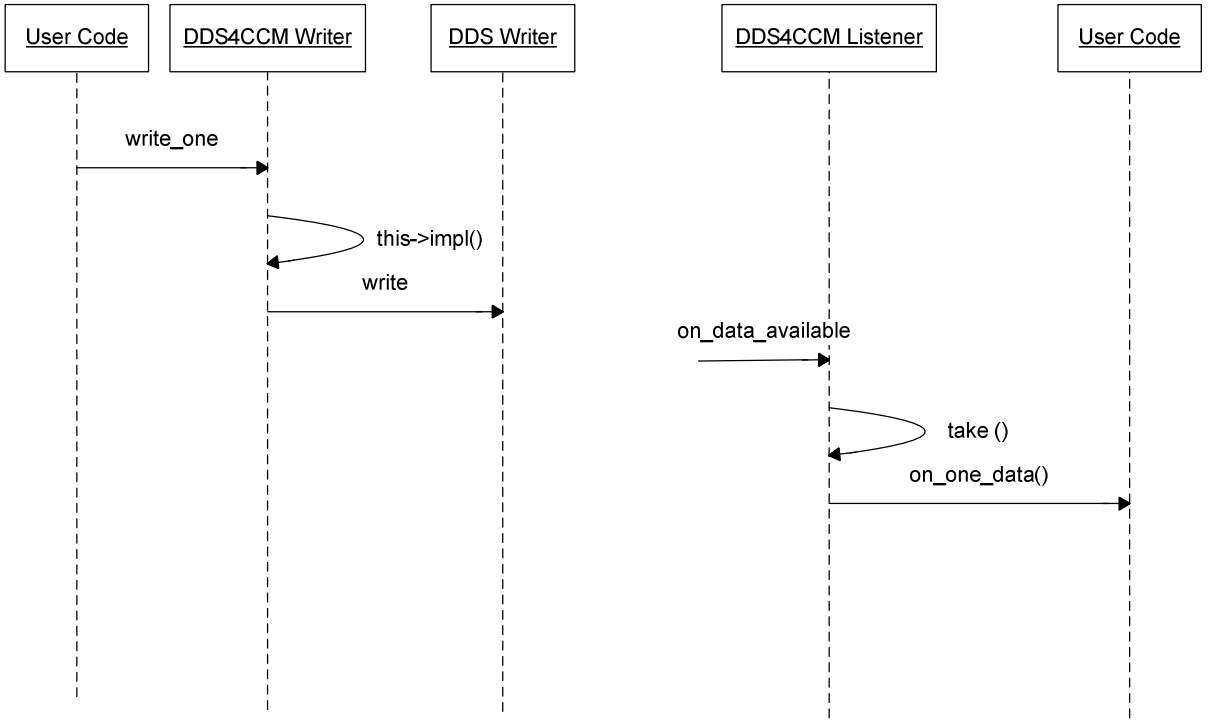
Prototype Entity Mapping



DDS4CCM LifeCycle



Writing and receiving one sample



Debugging DDS4CCM

- The environment variable DDS4CCM_NDDS_LOG_VERBOSITY controls the output of RTI NDDS

Output	Value
Silent	0
Errors	1
Warnings	2
Local status	4
Remote status	8
All statuses	255

- DDS4CCM's internal logging can be set by setting the CIAO_LOG_LEVEL environment variable (1=error/10=info).
- One should use the ACE_DEBUG and ACE_ERROR logging macros in the user code.

Implementation remarks

- StateListenerControl::is_filter_interpreted is only working for true
- The qos_profile string is interpreted as library_name#profile_name, which is RTI specific
- The core of DDS4CCM is not vendor independent yet, part of the code (including all *_with_profile) are specific to RTI
- For now, the only supported DDS implementation is version 4.5b Rev 01 of RTI. Add ndds=1 and dds4ccm_ndds=1 to your default.features and platform_macros.GNU file before you generate your makefiles
- Additional DDS vendors have to be added through partial template specialization using the DDS4CCM Vendor enum

Decisions (1/2)

- The life cycle of the DDS4CCM connector entities differ from the life cycle of the DDS entities (see previous sheets).
- RTI DDS entities are defined, using DDS as prefix. DDS4CCM implements a DDS module in IDL in which the interface to DDS is defined. Therefore the DDS4CCM representation of a DDS entity is defined as ::DDS::xxx
- Internally every DDS entity has got an DDS4CCM proxy class (prefixed with CCM_DDS_xxx). These proxy classes store a pointer to the actual DDS entity.
- Extended ports are created on demand. When an extended is not configured as a connection in the deployment plan, it is not created.

Decisions (2/2)

- The DDS4CCM basic port maps the DDS API to the user
- The DDS4CCM extended ports group the basic ports. They share the DataReader and QueryCondition (if defined)
- DomainParticipants with the same QoS profile are shared because of performance reasons.

Threading

- All DDS4CCM callbacks are delivered to the user component using the CCM threads
- When all callbacks have to be delivered on the DDS middleware threads add the following define to your ace/config.h file

```
#define CIAO_ DDS4CCM_CONTEXT_SWITCH 1
```

AMI FOR CCM

Motivating Asynchronous Method Invocation

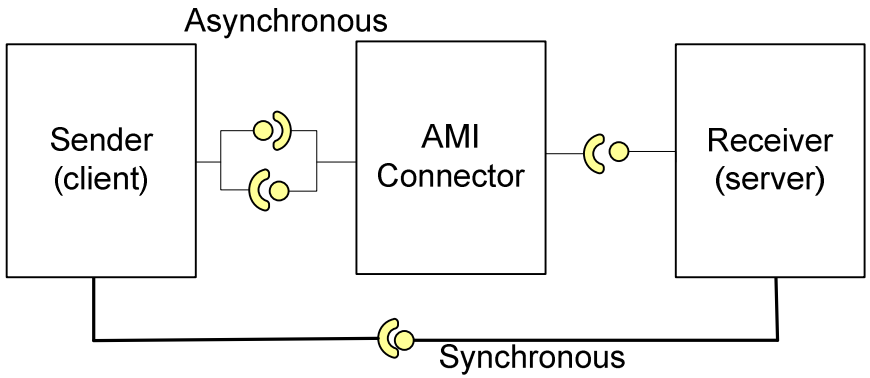
- **Context** – Asynchronous Method Invocation (AMI) can be extremely useful in component applications
  - Decrease end-to-end latency for multiple requests
  - Efficiently communicate with large numbers of connected components
  - Avoid unreliable and non-scalable workarounds (e.g., oneway operations, multi-threading synchronous invocations)
- **Problem –** CCM has no built-in support for AMI
  - Using AMI requires access to the ORB to register reply handlers
  - There is no standards-based way for an executor implementation to directly access all TAO entities needed
  - Using AMI directly introduces many of the complexities CCM attempts to avoid!

Asynchronous Method Invocation for CCM (AMI4CCM)

- **Solution** – Implement connectors to provide AMI services to components, similar to the DDS4CCM specification
- Provide a standardized AMI mechanism to CCM
- Uses IDL3 and IDL3+ language features
  - Extends the implicit IDL2 rules for AMI to IDL3
  - Use templated modules for the AMI4CCM connector
- Supports only the callback AMI model

AMI for CCM Overview (AMI4CCM)

- Tooling will generate an AMI connector
- AMI connector will handle details of executing the asynchronous invocation and callback to the user component
- Connector could use alternate communication middleware to accomplish AMI
- Server side components aren't aware of any AMI clients!
- Reply Handler is local IDL and not in the cdp



AMI4CCM Implied IDL (1/7)

The interface the application programmer created (Hello.idl) :

Used for enabling AMI4CCM per interface

```
#pragma ciao ami4ccm interface “MyFoo”

exception InternalError
{
  long id;
  string error_string;
}

interface MyFoo
{
  long foo (in string in_str, out string answer)
    raises (InternalError);
  attribute short rw_attrib
    getraises (InternalError)
    setraises (InternalError;
};
```

AMI4CCM Implied IDL (2/7)

Implied base idl (HelloA.idl) defining the AMI4CCM implied interfaces

```
// Implied ReplyHandler interface from AMI connector
// to Sender-Component
local interface AMI4CCM_MyFooReplyHandler :
  ::CCM_AMI::ReplyHandler
{
  void foo (
    in long ami_return_val,
    in string answer);
  void foo_excep (
    in CCM_AMI::ExceptionHolder exception_holder);
  void get_rw_attrib (
    in short rw_attrib);
  void get_rw_attrib_excep (
     in CCM_AMI::ExceptionHolder exception_holder);
  void set_rw_attrib ();
  void set_rw_attrib_excep (
     in CCM_AMI::ExceptionHolder exception_holder);
};
```

Type-specific reply handler

AMI4CCM Implied IDL (3/7)

continuation implied base idl (HelloA.idl)

```
// AMI interface. Sender calls AMI interface on the AMI connector. The AMI connector
// calls Receiver using the MyFoo interface.
local interface AMI4CCM_MyFoo
{
  void sendc_foo (
    in AMI4CCM_MyFooReplyHandler ami4ccm_handler,
    in string in_str);
  void sendc_get_rw_attrib (
     in AMI4CCM_MyFooReplyHandler ami4ccm_handler);
  void sendc_set_rw_attrib (
     in AMI4CCM_MyFooReplyHandler ami4ccm_handler);,
     in short rw_attrib );
};

module CCM_AMI::Connector_T<MyFoo, AMI4CCM_MyFoo> AMI4CCM_MyFoo_Connector;
```

Type-specific asynch interface for making invocations

Connector : uses MyFoo, the interface of the receiver.

provides AMI4CCM_MyFoo, the interface for the sender.

AMI4CCM Implied IDL (4/7)

The interface to write for the Sender (Hello_Sender.idl) :

```
#pragma ciao ami4ccm receptacle “Sender::run_my_foo"

// SENDER COMPONENT
component Sender
  {
    // For synchronous invocation
    uses MyFoo run_my_foo;
  };
```

Pragma is used for implying the asynch receptacle.

For asynchronous invocations, the Sender component uses the AMI4CCM_MyFoo interface of the AMI component and provides the AMI4CCM_MyFooReplyHandler interface to the AMI component.

For synchronous invocations, the Sender component uses the MyFoo interface (which the Receiver provides).

```
local interface CCM_Sender_Context: ::Components::SessionContext
  {
    ::MyFoo get_connection_run_my_foo ();
    ::AMI4CCM_MyFoo get_connection_sendc_run_my_foo ();
  };
```

Part of the implied interface for the Sender(Hello_SenderE.idl)

AMI4CCM Implied IDL (5/7)

The Connector, the AMI-Component, will be generated completely by the IDL compiler by implying ami4ccm.idl

```
connector AMI4CCM_Base
{
};

module Connector_T<interface T, interface AMI4CCM_T>
{
   porttype AMI4CCM_Port_Type
     {
       provides AMI4CCM_T ami4ccm_provides;
       uses T ami4ccm_uses;
     };

    connector AMI4CCM_Connector : AMI4CCM_Base
     {
       port AMI4CCM_Port_Type ami4ccm_port;
     };
};
```

Part of ami4ccm.idl, defining the connector interface

AMI4CCM Implied IDL (6/7)

The Receiver just implements the interface provided by the application programmer. The Receiver component has no idea which component (in this case Sender or AMI) uses his interface!

The interface to write for the Receiver (Hello_Receiver.idl) :

```
// RECEIVER COMPONENT
component Receiver
  {
    provides MyFoo do_my_foo;
  };
```

AMI4CCM Implied IDL (7/7)

The ExceptionHolder only delivers the functionality to rethrow the original exception

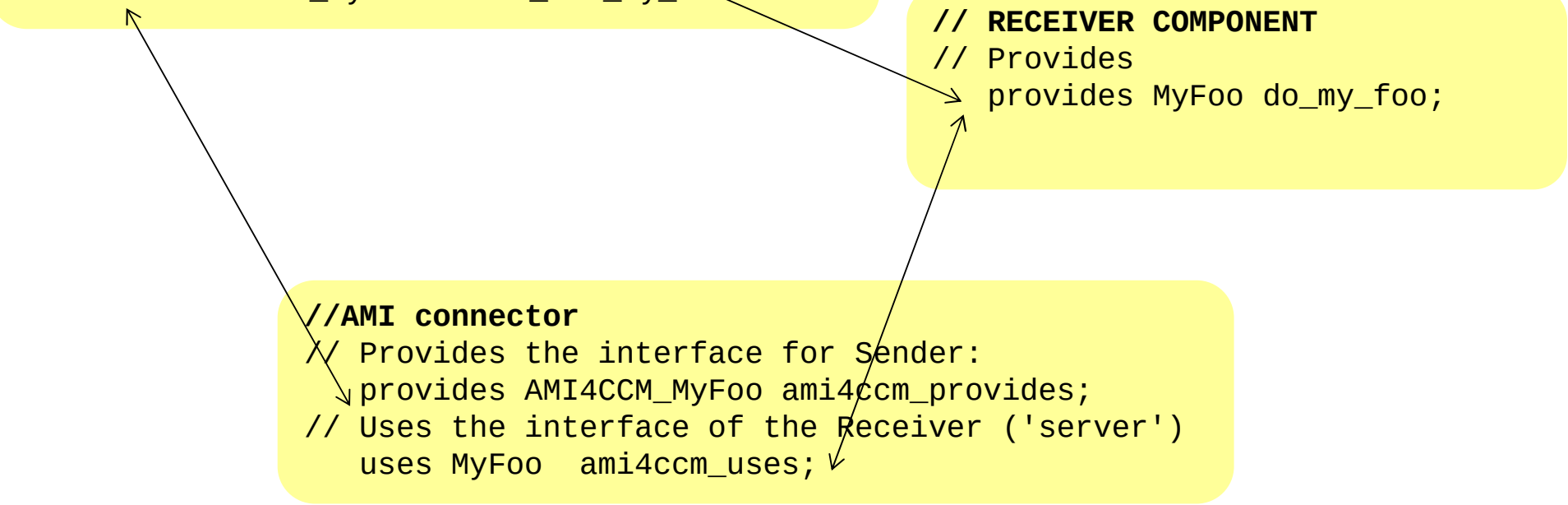
```
native UserExceptionBase;
local interface ExceptionHolder
  {
    void raise_exception() raises (UserExceptionBase);
  };
```

Overview of the implied ports

```
// SENDER COMPONENT
// For synchronous invocation
    uses MyFoo run_my_foo;
// For asynchronous invocation
    uses AMI4CCM_MyFoo sendc_run_my_foo
```

```
// RECEIVER COMPONENT
// Provides
   provides MyFoo do_my_foo;
```

```
//AMI connector
// Provides the interface for Sender:
   provides AMI4CCM_MyFoo ami4ccm_provides;
// Uses the interface of the Receiver ('server')
   uses MyFoo  ami4ccm_uses;
```



The implied ports with 'uses multiple'

For asynchronous invocation, with *uses multiple* , one Sender (client) is connected with N AMI connectors to N Receivers (servers).

The connection section of the deployment plan defines the N connections between the client and the N connectors and between the N connectors and the N servers.

```
// SENDER COMPONENT
// For asynchronous invocation
    uses multiple AMI4CCM_MyFoo sendc_run_my_foo
```

```
//AMI connector 1
// Provides the interface for Sender:
   provides AMI4CCM_MyFoo ami4ccm_provides;
// Uses the interface of the Receiver
   uses MyFoo  ami4ccm_uses;
```

```
// RECEIVER COMPONENT 1
// Provides
   provides MyFoo do_my_foo;
```

```
//AMI connector 2
// Provides the interface for Sender:
   provides AMI4CCM_MyFoo ami4ccm_provides;
// Uses the interface of the Receiver
   uses MyFoo  ami4ccm_uses;
```

```
// RECEIVER COMPONENT 2
// Provides
   provides MyFoo do_my_foo;
```

....

```
//AMI connector N
// Provides the interface for Sender:
   provides AMI4CCM_MyFoo ami4ccm_provides;
// Uses the interface of the Receiver
   uses MyFoo  ami4ccm_uses;
```

```
// RECEIVER COMPONENT N
// Provides
   provides MyFoo do_my_foo;
```


AMI4CCM Example Implementation (1/3)

Implied Reply Handler IDL.

```
AMI4CCM_MyFooReplyHandler :
 ::CCM_AMI::ReplyHandler
{
  void foo (
    in long ami_return_val,
    in string answer);
  void foo_excep (
    in CCM_AMI::ExceptionHolder
      exception_holder);
  void get_rw_attrib (
    in short rw_attrib);
  void get_rw_attrib_excep (
    in CCM_AMI::ExceptionHolder
      exception_holder);
};
```

Implementation Sender_exec.h

```
class MyFoo_RH_exec_i
    : public virtual
      ::Hello::AMI4CCM_MyFooReplyHandler,
      public virtual ::CORBA::LocalObject
  {
  public:
    MyFoo_RH_exec_i ();
    virtual ~ MyFoo_RH_exec_i ();

    virtual void foo (::CORBA::Long
      ami_return_val, const char * answer);
    virtual void foo_excep
      ::CCM_AMI::ExceptionHolder_ptr
      excep_holder);
```

AMI4CCM Example Implementation (2/3)

Implied Reply Handler IDL.

```
AMI4CCM_MyFooReplyHandler :
 ::CCM_AMI::ReplyHandler
{
  void foo (
    in long ami_return_val,
    in string answer);
  void foo_excep (
    in CCM_AMI::ExceptionHolder
      exception_holder);
  void get_rw_attrib (
    in short rw_attrib);
  void get_rw_attrib_excep (
    in CCM_AMI::ExceptionHolder
      exception_holder);
};
```

Implementation Sender_exec.cpp

```
MyFoo_RH_exec_i :: MyFoo_RH_exec_i(void)
{}
MyFoo_RH_exec_i :: ~ MyFoo_RH_exec_i (void)
{}
void
MyFoo_RH_exec_i ::foo (
            ::CORBA::Long ami_return_val,
            const char *  answer)
{ printf ("Answer <%s>\n", answer); }

void
MyFoo_RH_exec_i ::foo_excep (
 const ::CCM_AMI::ExceptionHolder * exception_holder);
{ try {
    exception_holder->raise_exception ();
  }
  catch (…)
  {
  }
}
```

AMI4CCM Example Implementation (3/3)

Implementation in Sender_exec.cpp:

```
::AMI4CCM_MyFoo_var my_foo_ami =
     this->context_->get_connection_sendc_run_my_foo();
my_foo_ami_->sendc_foo (new MyFoo_RH_exec_i (), "Do something asynchronous");
```

Get the asynch receptacle

Make the asynch invocation

Reply handler, can be a global one or one per request

```
::MyFoo_var my_foo =
     this->context_->get_connection_run_my_foo();
my_foo_->foo ("Do something synchronous“, answer);
```

Get the synch receptacle

Make the synch invocation

AMI4CCM Example Implementation with "uses multiple" (1/1)

// The implementation of Sender_exec.cpp:

```
::Sender::sendc_run_my_fooConnections_var my_foo_ami_ =
        this->context_->get_connections_sendc_run_my_foo ();

 for (CORBA::Ulong I = 0; I < my_foo_ami_->length (); ++i)
  {
    my_foo_ami_[i].objref_->sendc_foo (new MyFoo_RH_exec_i (), "Do something asynchronous");
  }
```

Get all of the asynch receptacles

Make multiple asynch invocations

AMI4CCM Prototype

- $CIAO_ROOT/connectors/ami4ccm/examples/Hello
- Sender component
- Receiver component
- AMI connector
- Deployment plan for these 3 components

AMI4CCM 'uses multiple' example

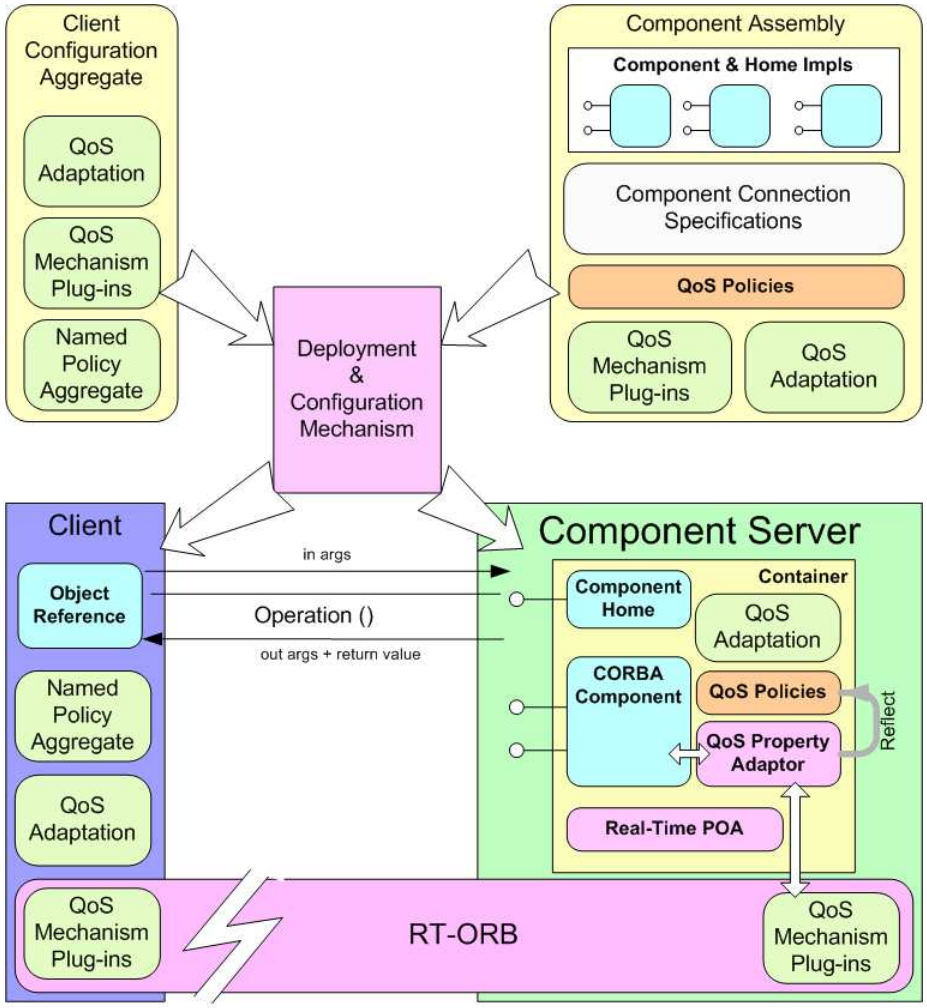
- $CIAO_ROOT/connectors/ami4ccm/tests/UsesMulti
- 1 Sender component
- 3 Receiver components
- 3 AMI connectors
- Deployment plan for these components

Overview of CIAO & Future R&D Directions

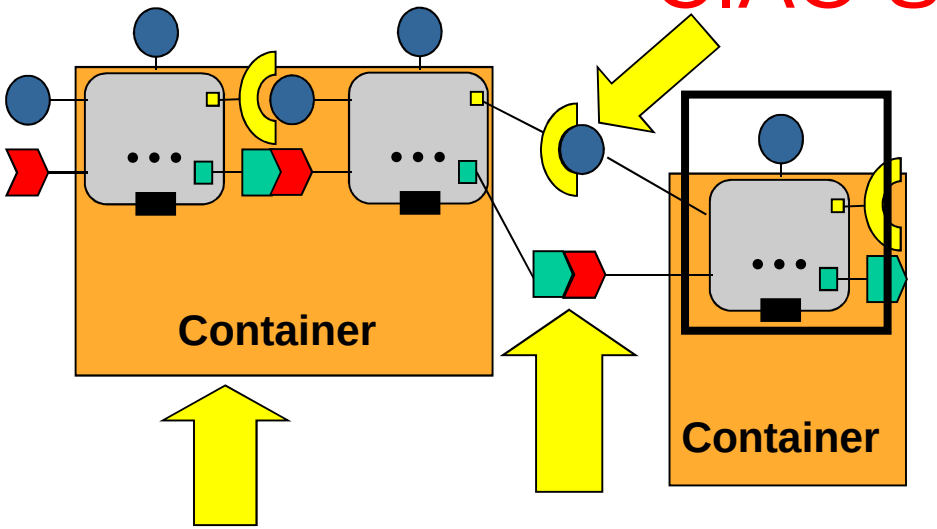
Overview of CIAO

- Component Integrated ACE ORB
  - Lightweight CCM implementation atop TAO
  - Supports component-oriented paradigm for DRE applications
    - Provides Real-time CORBA policies & mechanisms required for DRE applications
    - Key DRE aspects are supported as first-class metadata
- First official release (CIAO 0.4) was at end of December 2003
- Latest release is downloadable from

  download.dre.vanderbilt.edu



CIAO Status



- Support for IDL 3 (**`component, home`** & related keywords) & most CIDL features have been added
- Support for all types of ports: facets (**`provides`**), receptacles (**`uses, uses multiple`**), event sources (**`emits, publishes`**) & event sinks (**`consumes`**)
- Support for the Session container via IDL compiler
- Components can be built as shared libs or static libs
- Component server supported
- MDD tools to install, host, load, & manage component implementations are available
- The CIAO Deployment and Configuration Engine (DAnCE) provides support for component assemblies in compliance with ptc/06-04-01
- CIAO also supports Real-time CCM extensions
  - www.cs.wustl.edu/~schmidt/CIAO.html

CIAO Next Steps

- **Deployment & Configuration (Leads: Will Otte & Johnny Willemsen)**
  - Implementing the new deployment & configuration specification, ptc/06-04-02
  - Changes to the deployment & assembly toolset to support lightweight components, as prescribed by ptc/06-04-01
- **Core CCM Infrastructure (Leads: Will Otte & Johnny Willemsen)**
  - Additional support for Real-time CORBA Policies at the ORB level & object level
    - i.e., at the object reference level of a component receptacle
  - Integration of different event propagation mechanisms (such as Event & Notification Services) within the container
- **Modeling tool support for CIAO (Leads: James Hill & Jeff Parsons)**
  - See www.dre.vanderbilt.edu/cosmic for details

How to Learn about CCM & CIAO Programming

- Examples available with the distribution
  - **`CIAO/docs/tutorial/Hello`**, a simple example that illustrates the use of some basic CCM concepts
  - **`CIAO/examples/BasicSP`**
    - A simple example that shows the interaction between 4 components
  - **`CIAO/examples/Display`**
    - Similar to the BasicSP, but has an additional feature showing integration with Qt toolkit
- Step-by-step to create & deploy components based on CIAO available at
  - **`CIAO/examples/Hello`**
- "Quick CORBA 3", Jon Siegel, John Wiley & Sons provides a quick start
- C/C++ User Journal articles with Steve Vinoski
  - www.cs.wustl.edu/~schmidt/report-doc.html

Wrapping Up

Tutorial Summary

- **CCM spec**
  - Extends the CORBA object model to support application development via composition
  - CORBA Implementation Framework (CIF) defines ways to automate the implementation of many component features
  - Defines standard run-time environment with Containers & Component Servers
  - Specifies deployment & configuration framework
- **Deployment & Configuration** specification separates key configuration concerns
  - Server configuration
  - Object/service configuration
  - Application configuration
  - Object/service deployment

Additional Information on CORBA & CCM

OMG specifications pertaining to CCM

- CORBA Component Model (CCM)
  - formal/06-04-01
- QoS for CCM
  - formal/08-10-02
- Streams for CCM
  - ptc/05-07-01
- UML Profile for CCM
  - formal/08-04-07
- Deployment & Configuration (D&C)
  - formal/06-04-02
- DDS4CCM
  - ptc/10-05-08

Books pertaining to CCM

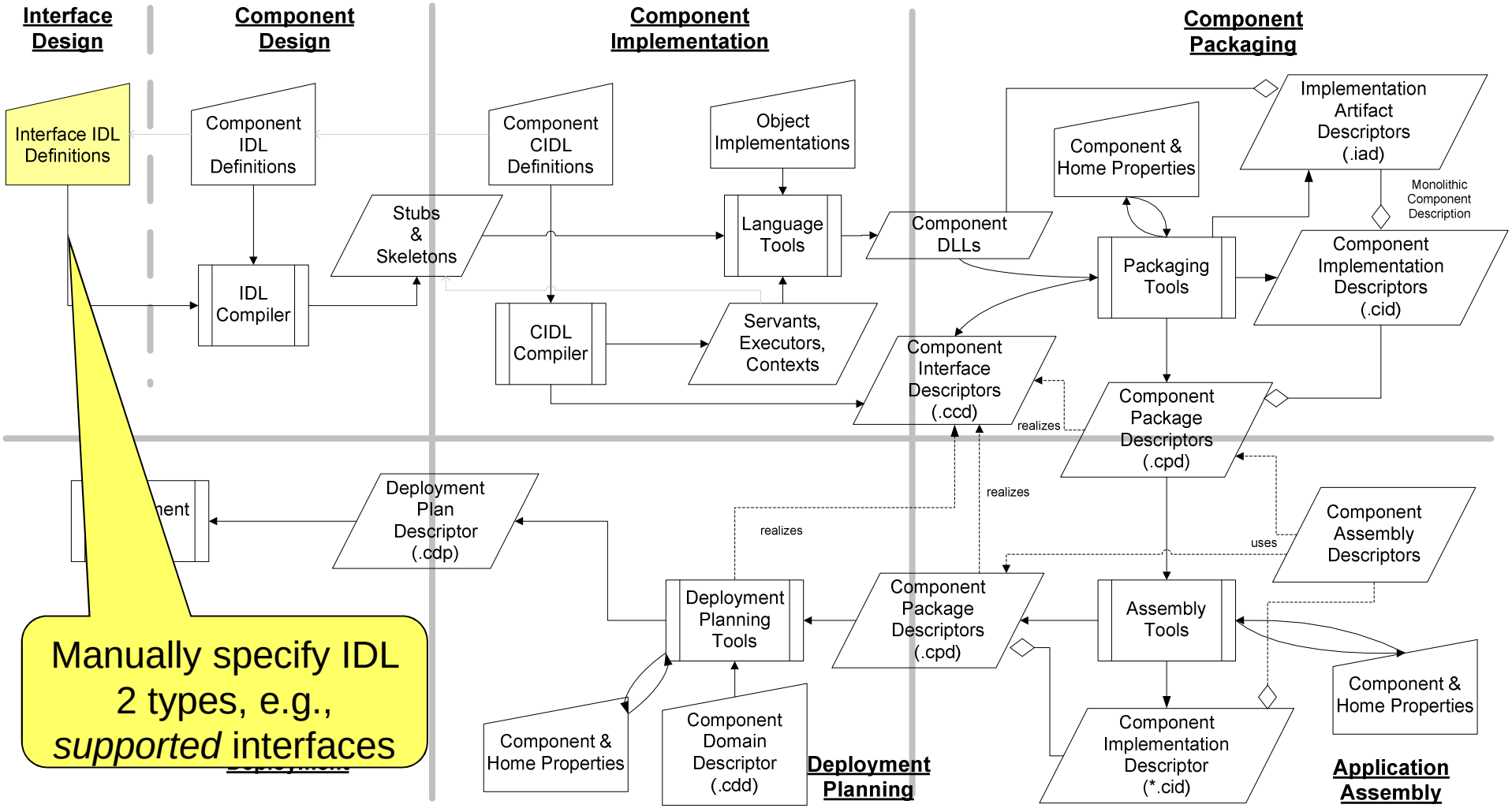
- *CORBA 3 Fundamentals & Programming*, Dr. John Siegel, published at John Wiley & Sons

Web resources pertaining to CCM

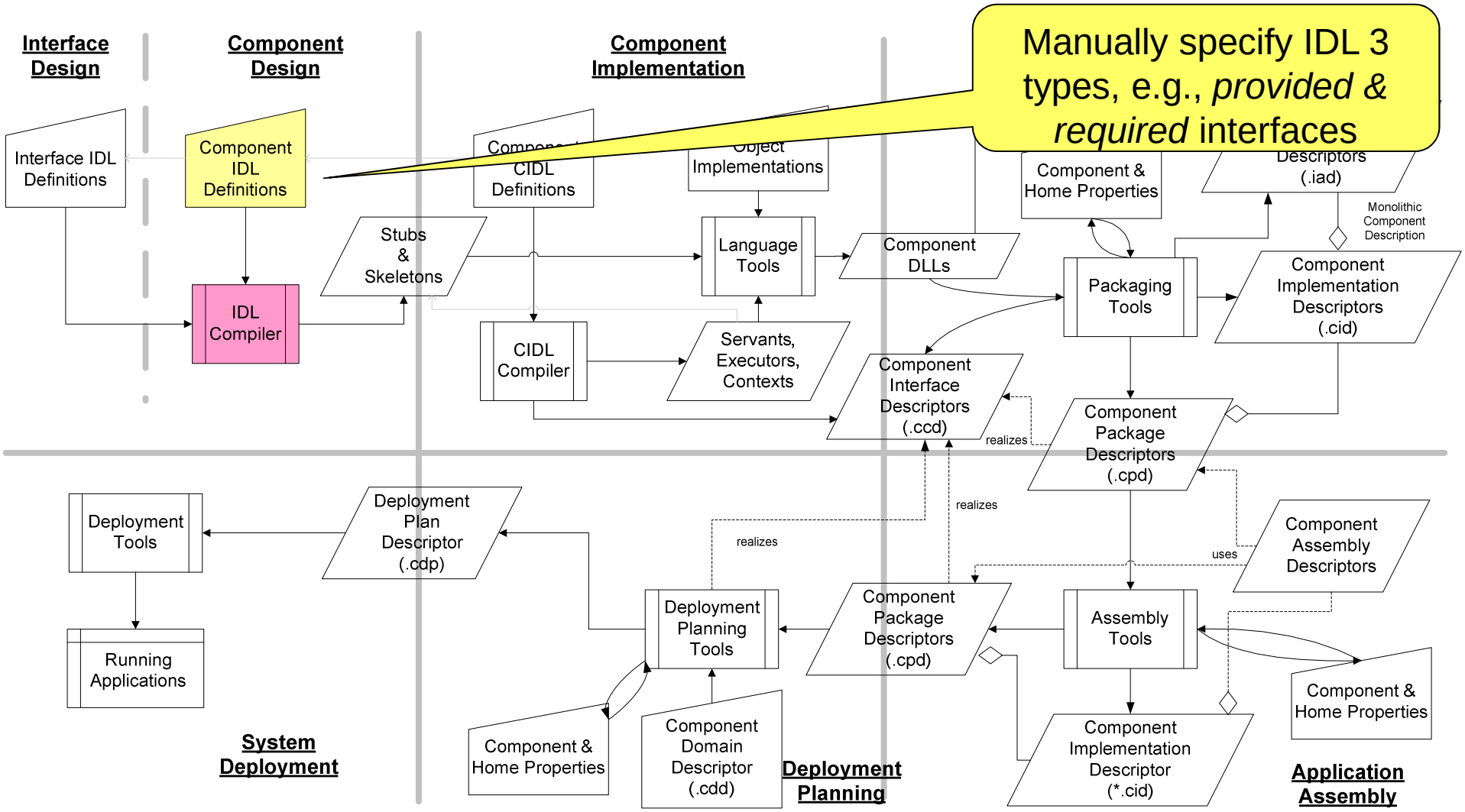
- "The CCM Page" by Diego Sevilla Ruiz
  - www.ditec.um.es/~dsevilla/ccm/
- OMG CCM specification
  - www.omg.org/technology/documents/formal/components.htm
- CUJ columns by Schmidt & Vinoski
  - www.cs.wustl.edu/~schmidt/report-doc.html

Overview of an Conventional Component Application Development Lifecycle

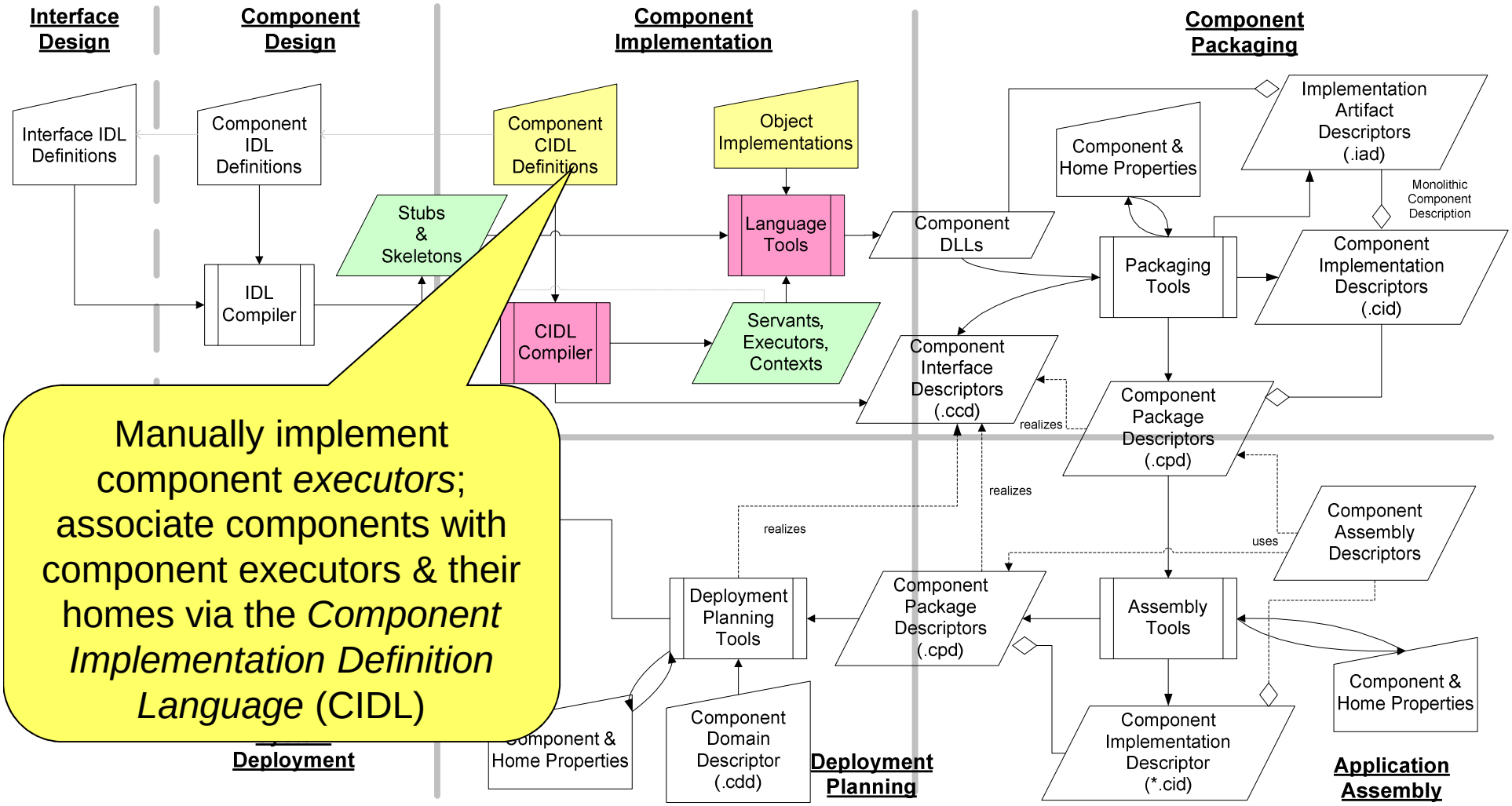
Component Application Development Lifecycle



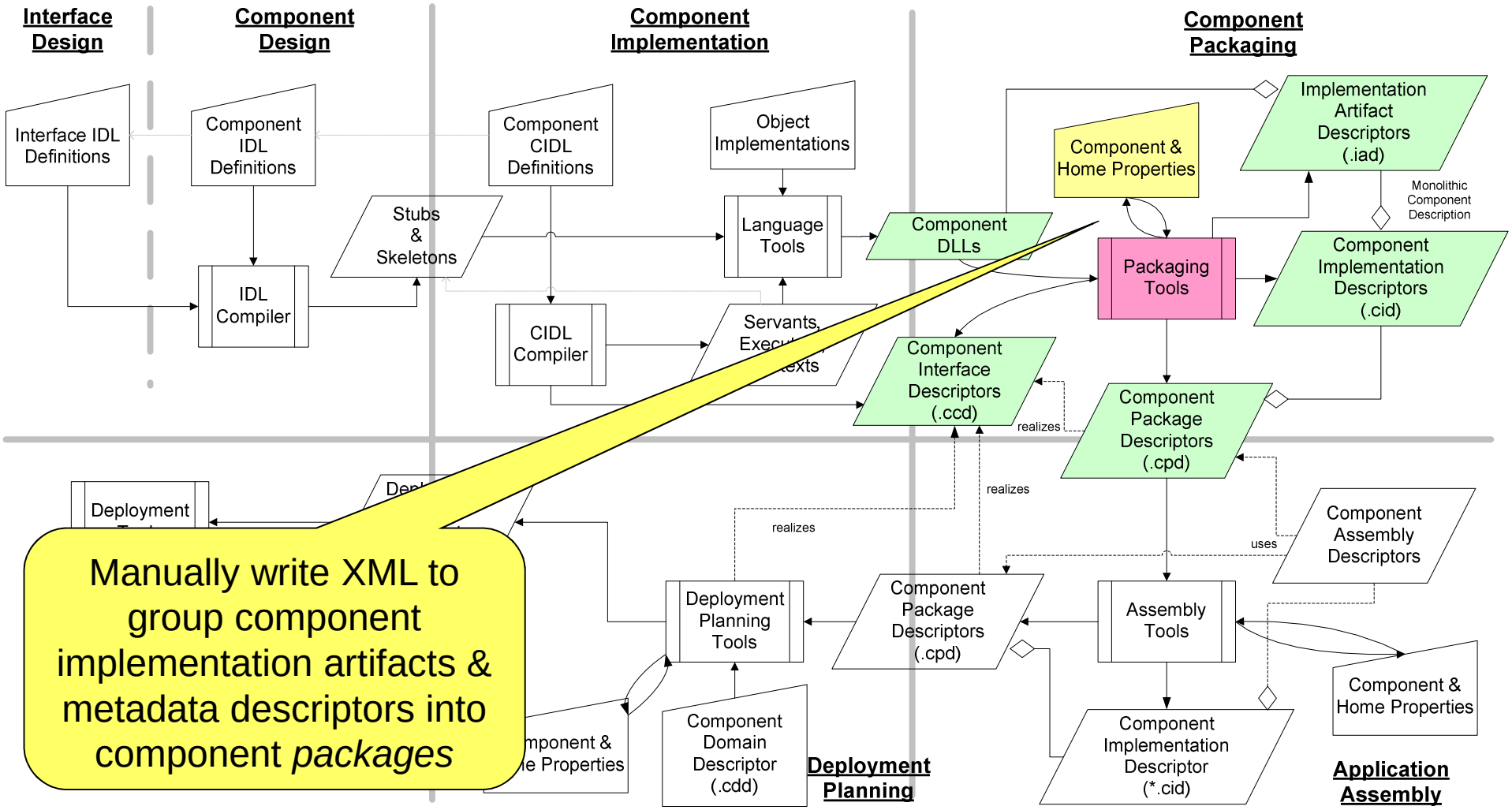
Component Application Development Lifecycle



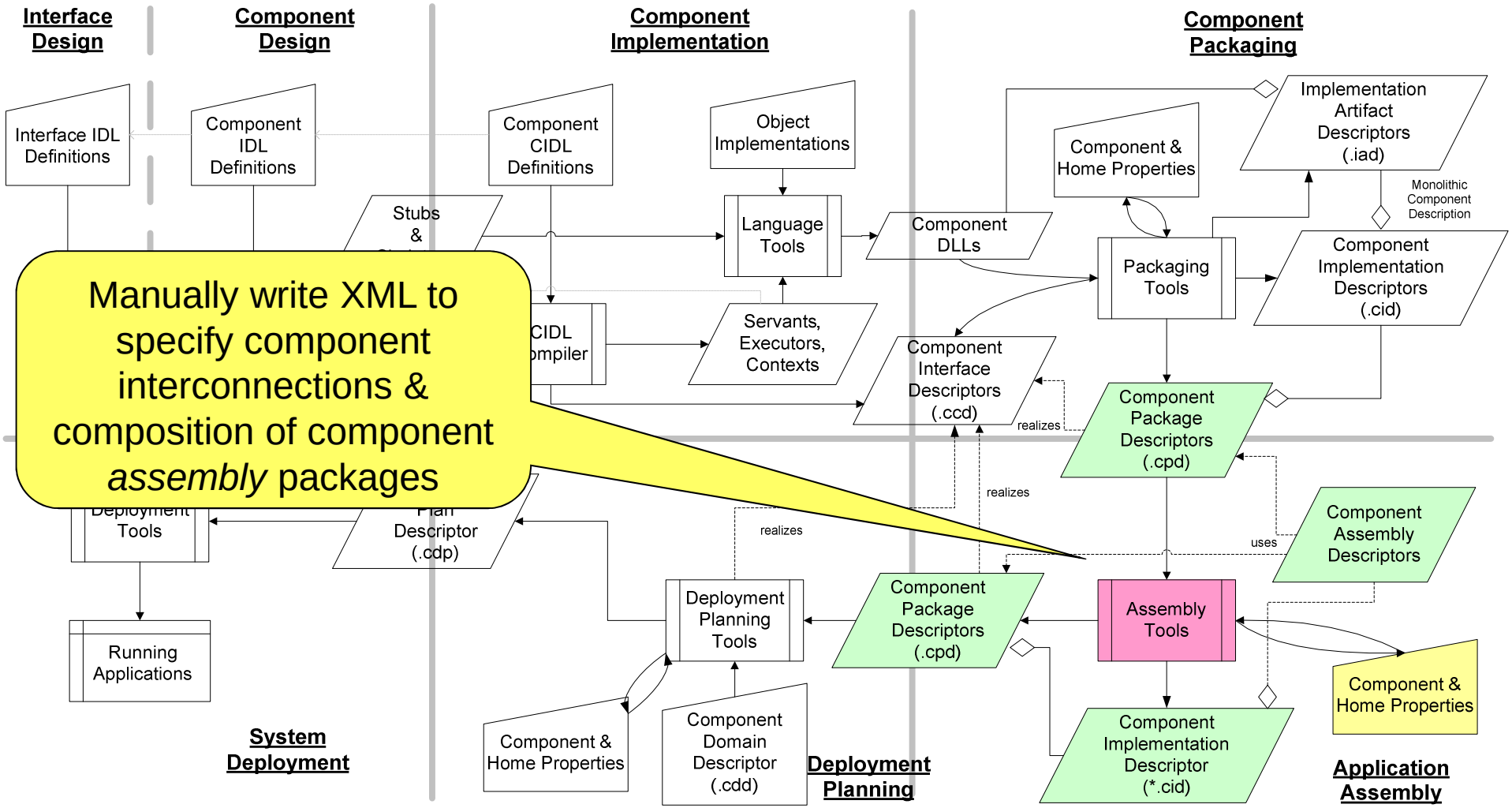
Component Application Development Lifecycle



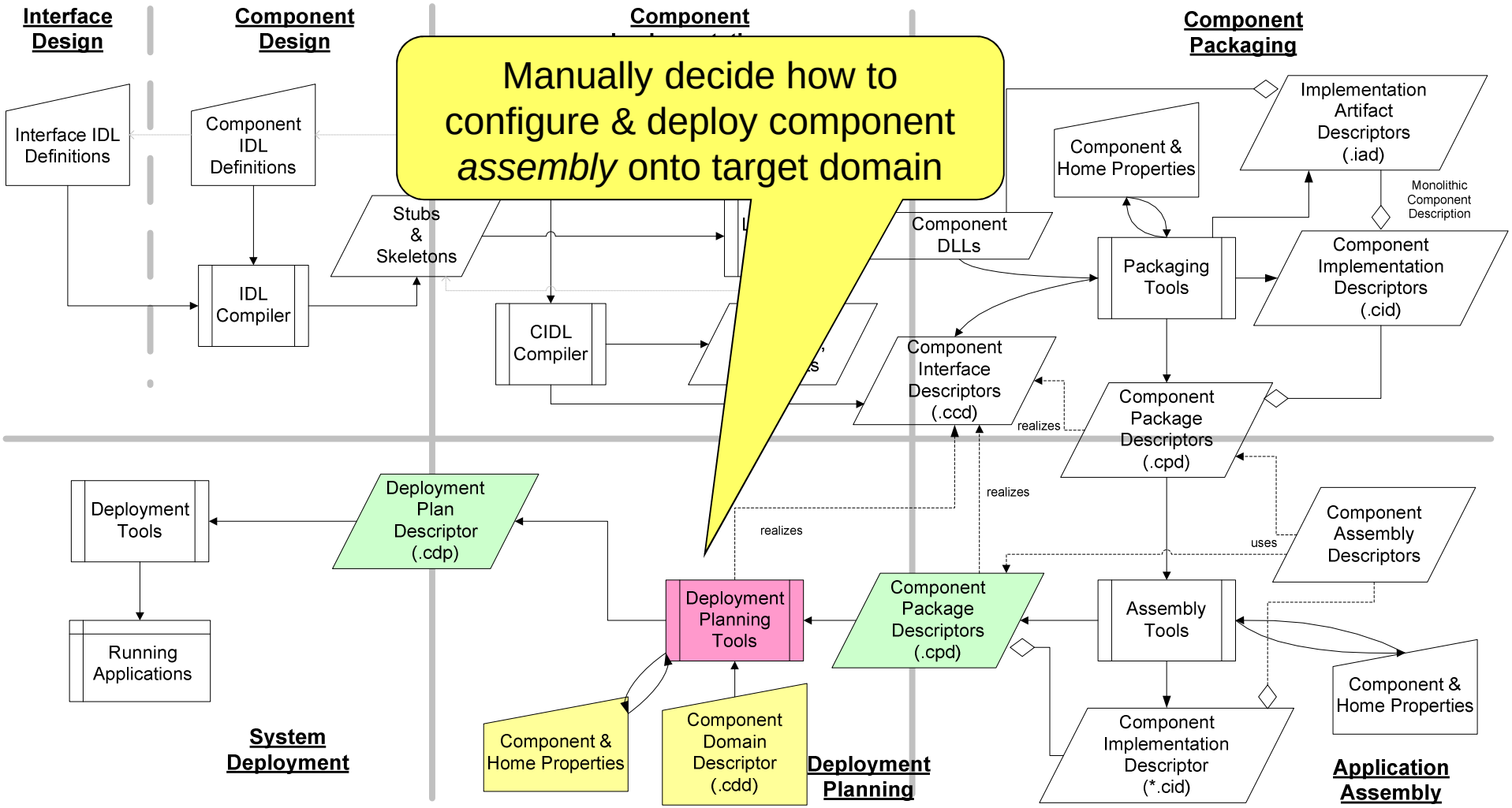
Component Application Development Lifecycle



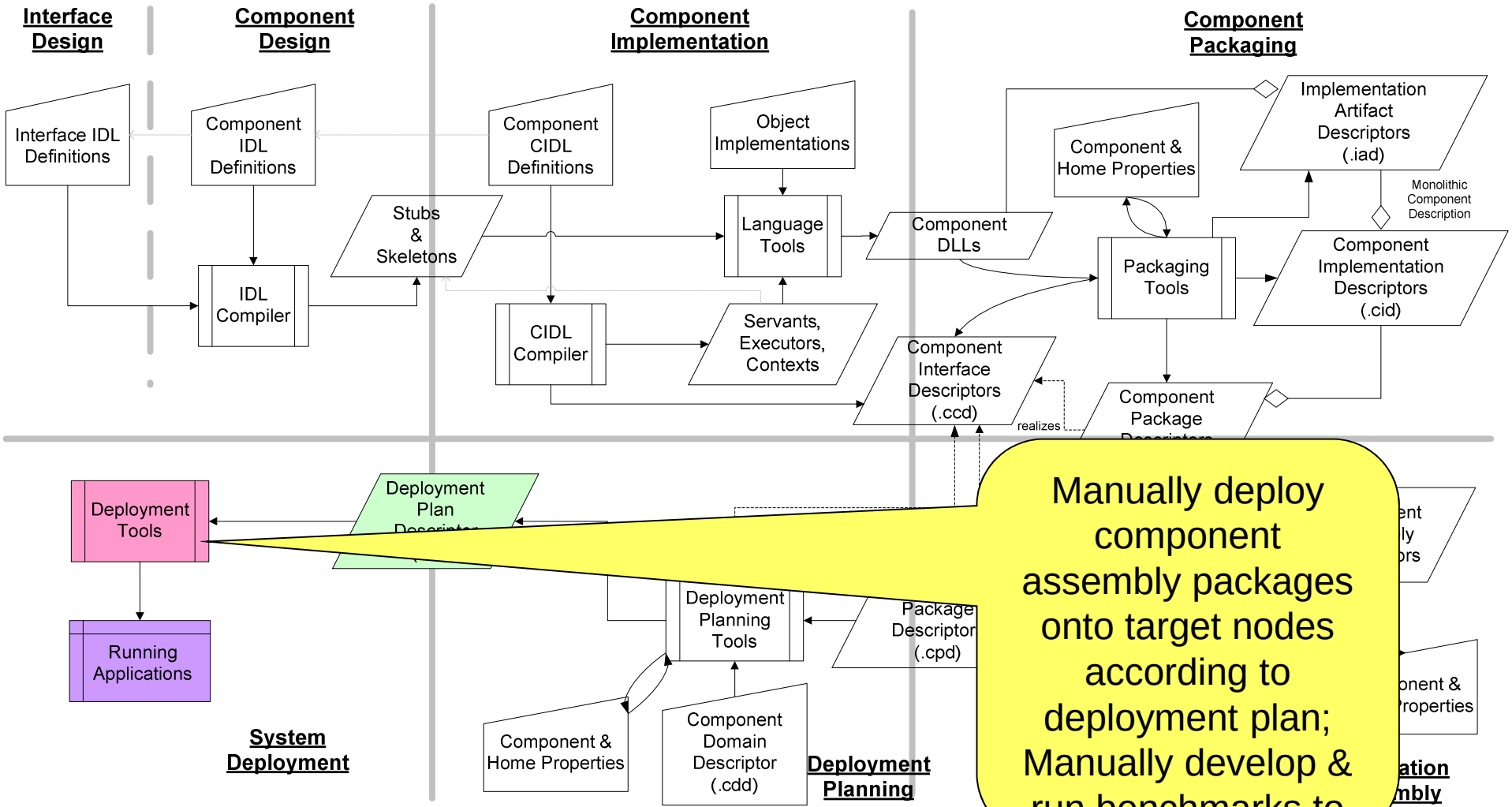
Component Application Development Lifecycle



Component Application Development Lifecycle

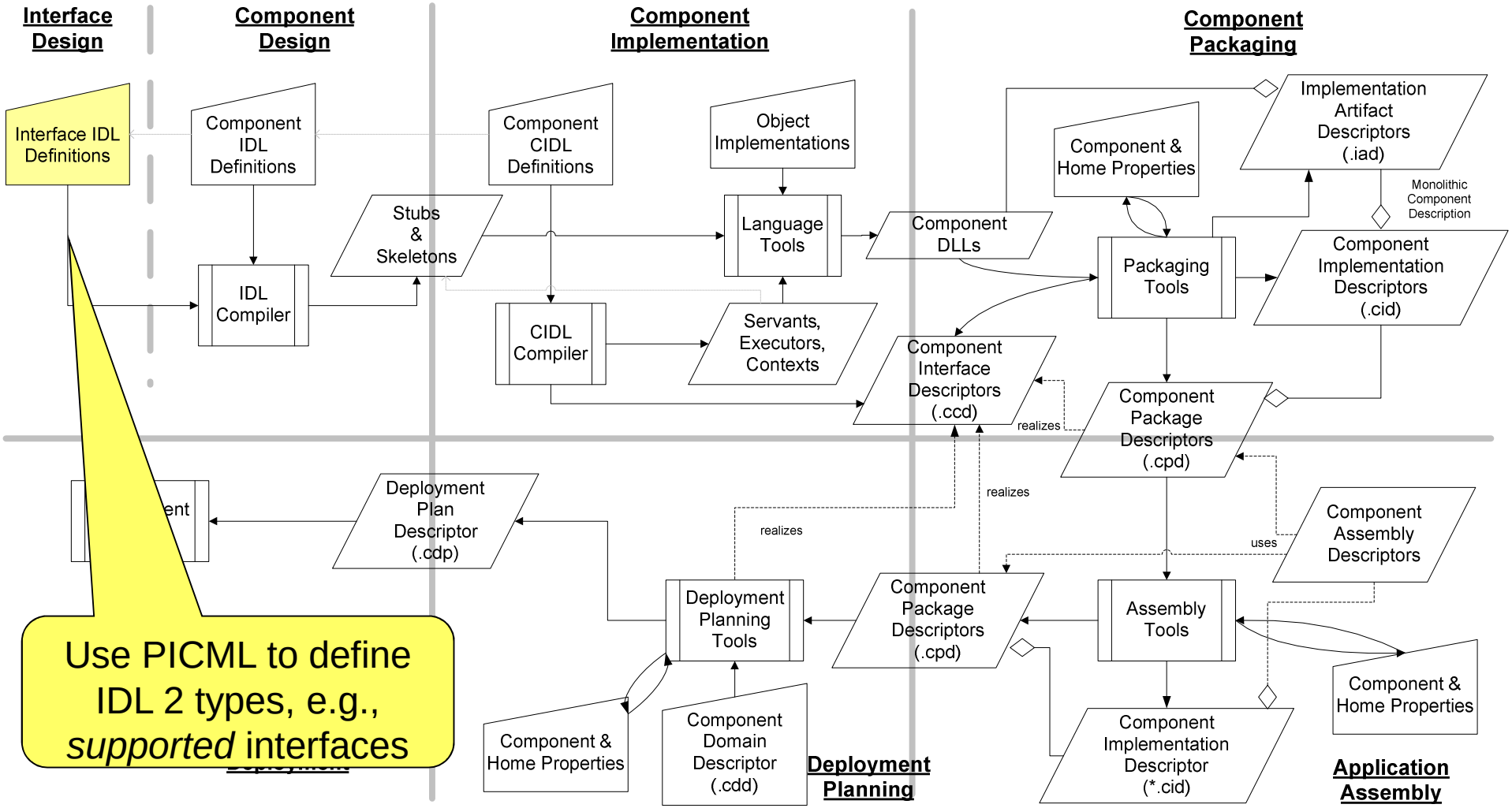


Component Application Development Lifecycle

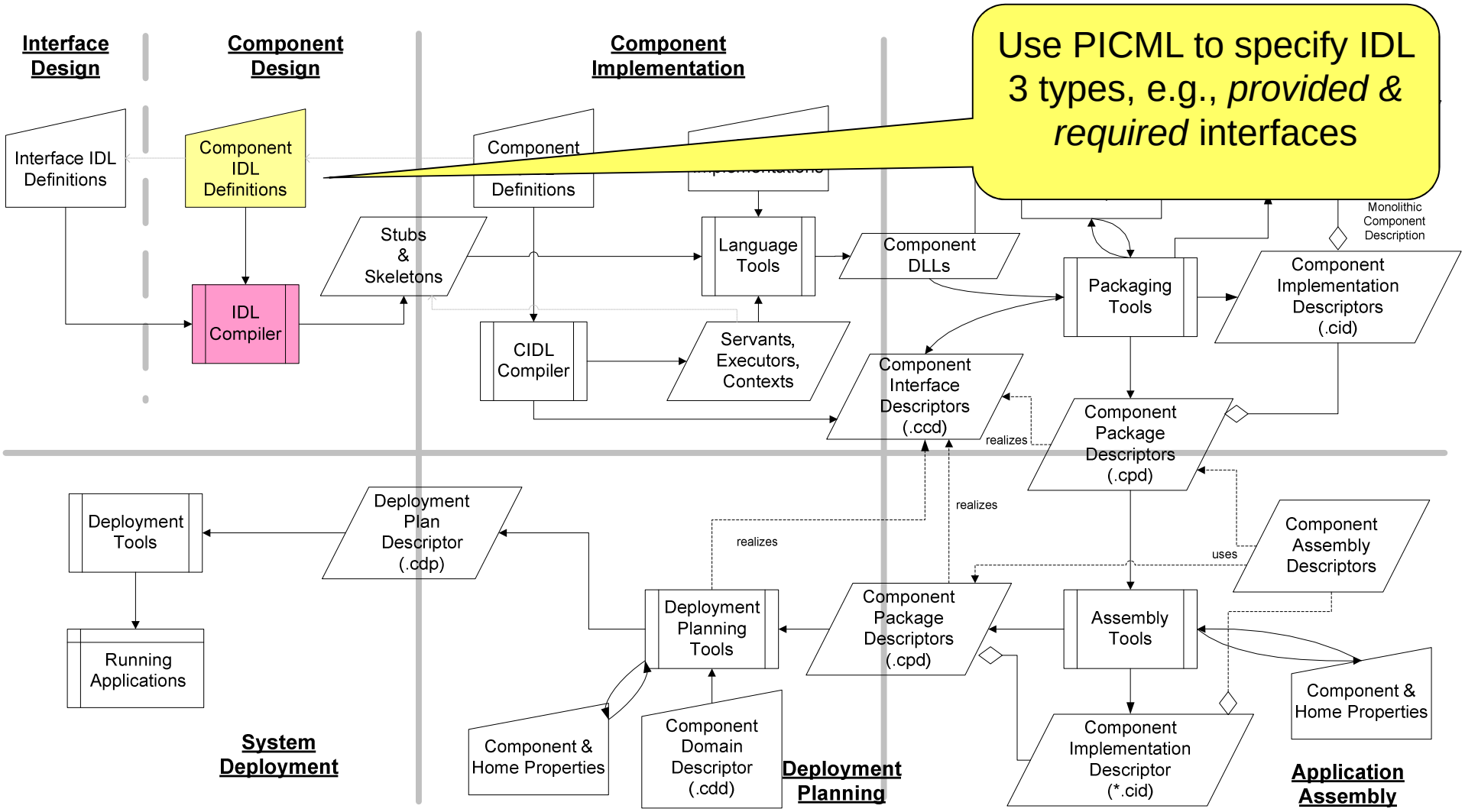


Overview of an MDD-based Component Application Development Lifecycle

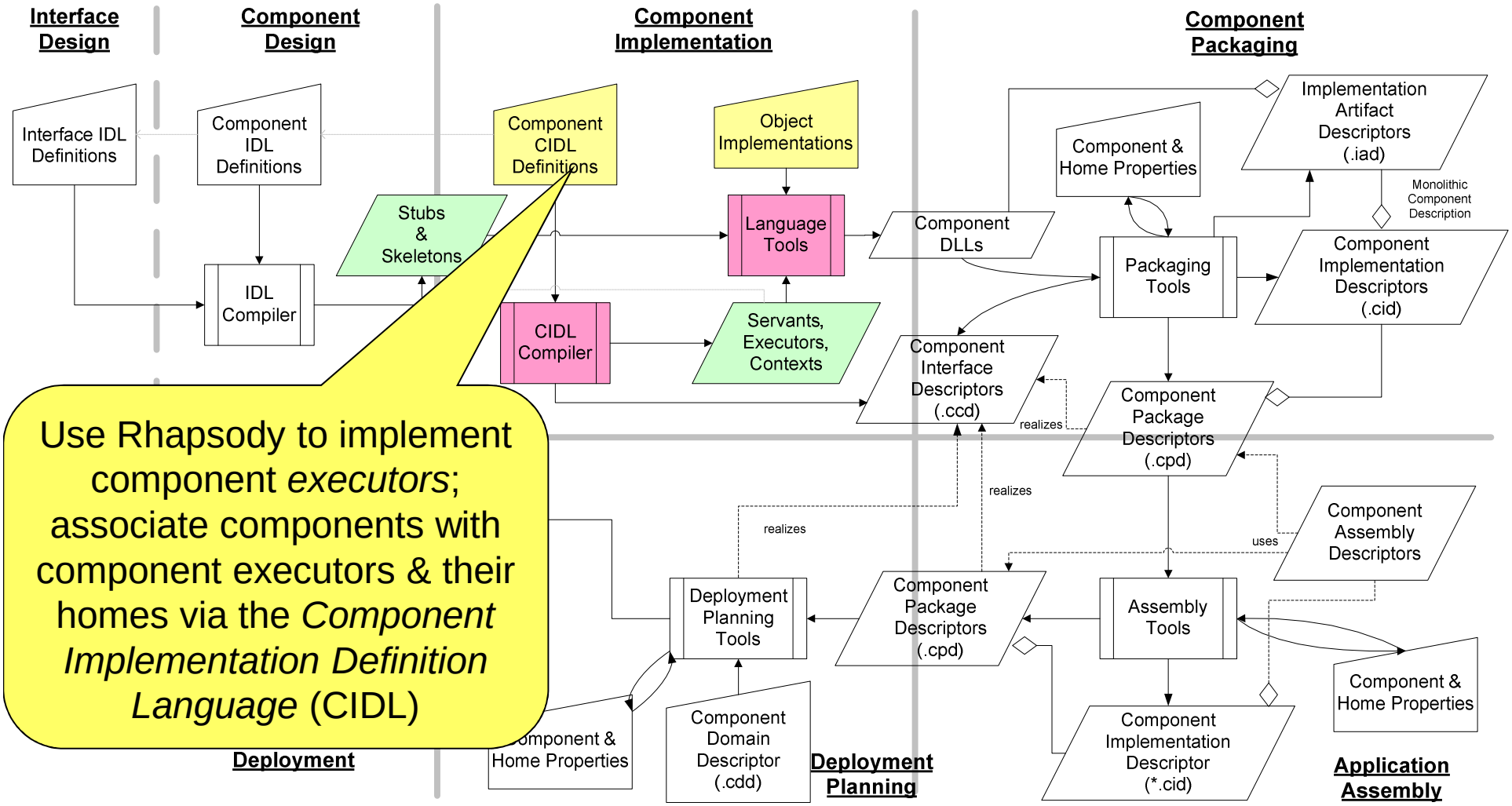
MDD-based Component Application Development Lifecycle



MDD-based Component Application Development Lifecycle

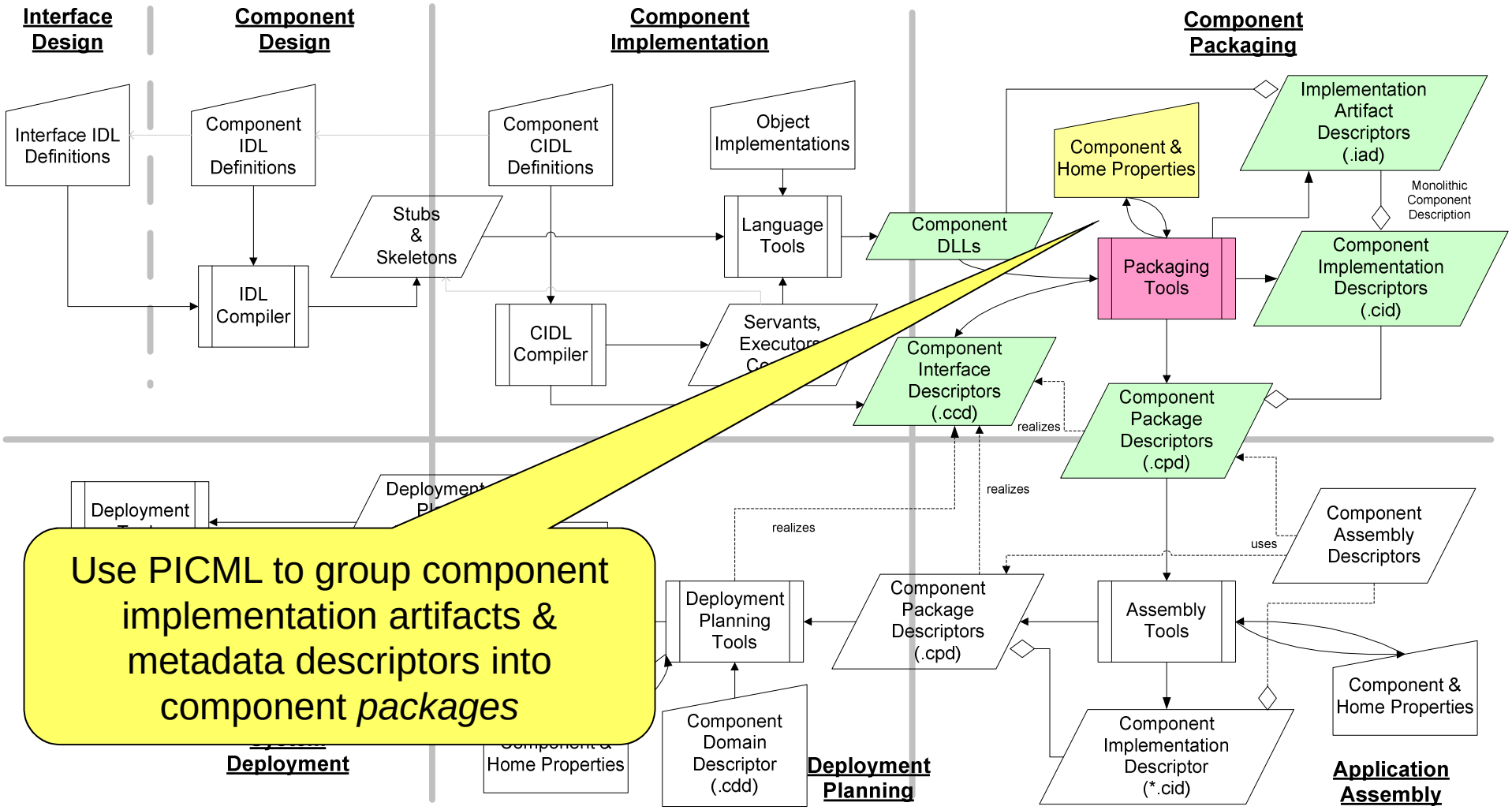


MDD-based Component Application Development Lifecycle

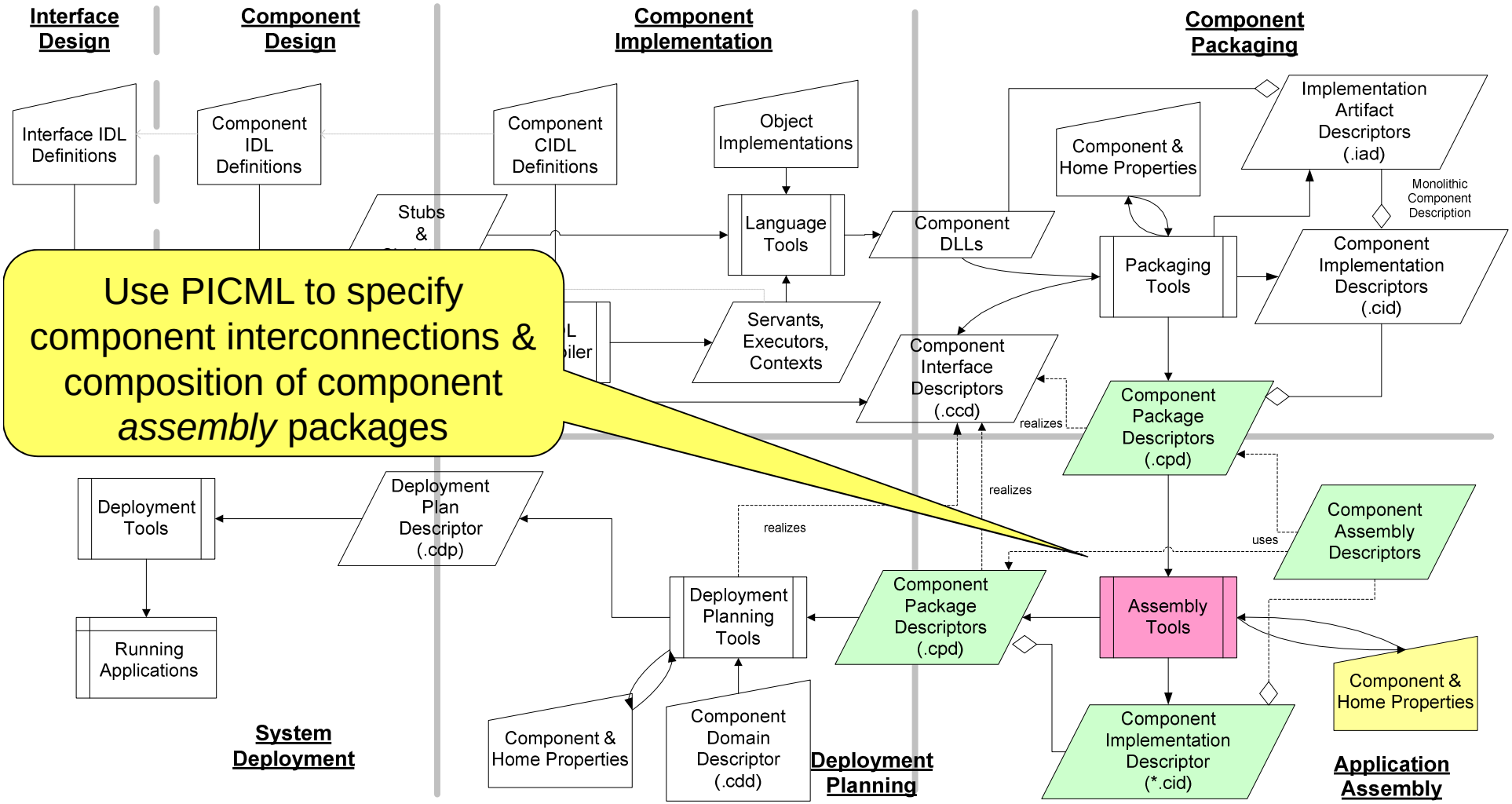


Use Rhapsody to implement component *executors*; associate components with component executors & their homes via the *Component Implementation Definition Language* (CIDL)

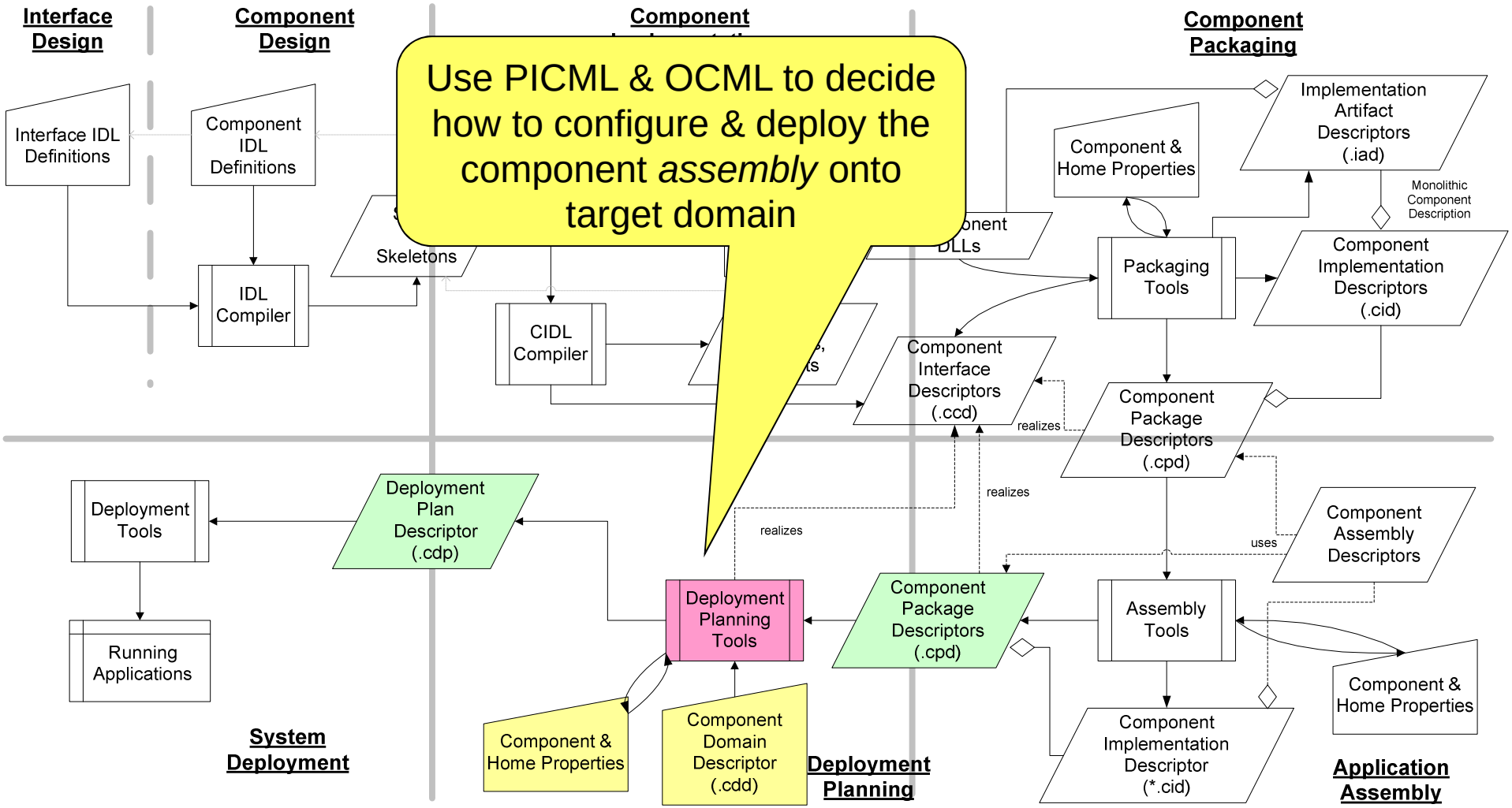
MDD-based Component Application Development Lifecycle



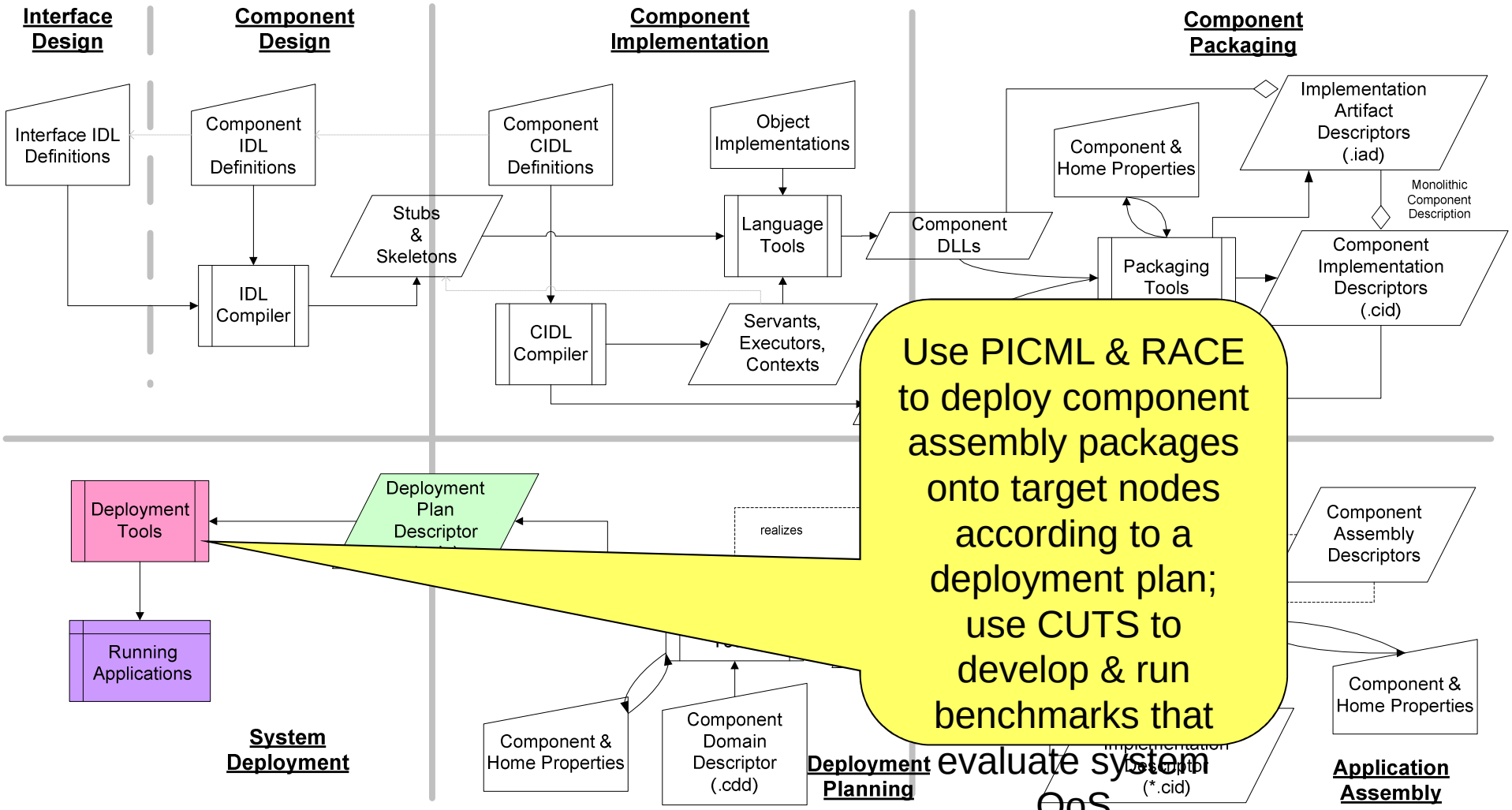
MDD-based Component Application Development Lifecycle



MDD-based Component Application Development Lifecycle



MDD-based Component Application Development Lifecycle



Summary of MDD-based Component Development Lifecycle