Tutorial on the Lightweight CORBA Component Model (CCM)

Industrializing the Development of Distributed Real-time & Embedded Applications

Douglas C. Schmidt

schmidt@dre.vanderbilt.edu

Vanderbilt University & ISIS

Frank Pilhofer

fp@mc.com

Mercury Computing Systems

OMG Real-time & Embedded Systems Workshop

July 12th & 13th, 2004

Other contributors included Kitty Balasubramanian, Tao Lu, Bala Natarajan, Jeff Parsons, Craig Rodrigues, & Nanbor Wang

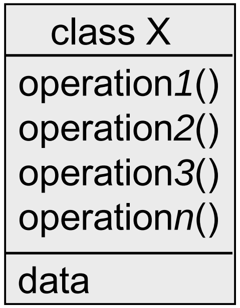
Tutorial Overview

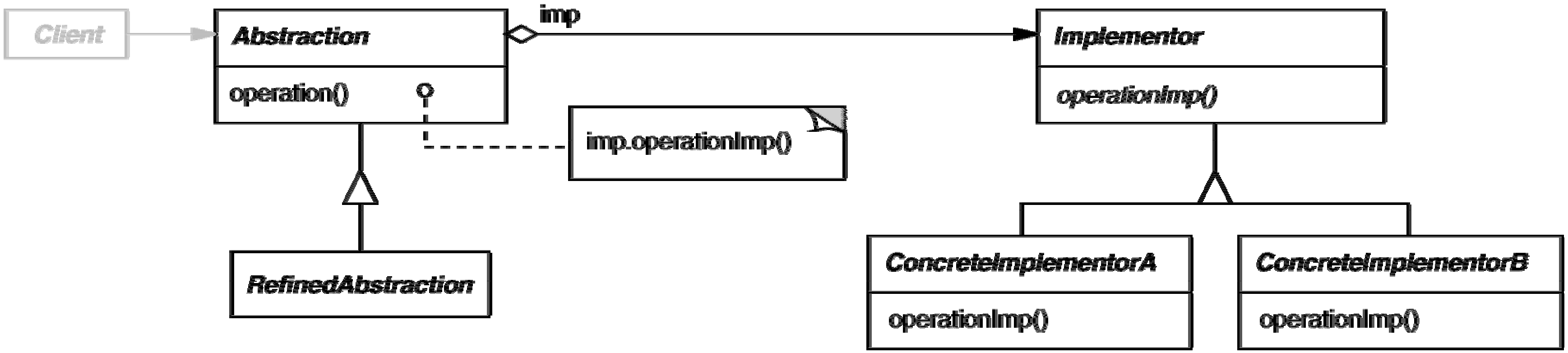
- The purpose of this tutorial is to
  - Motivate the need for the CORBA Component Model (CCM) & contrast it with the CORBA 2.x distributed object computing (DOC) model
  - Introduce CCM features most relevant to distributed real-time & embedded (DRE) applications
    - i.e., Lightweight CCM & the new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

Motivation & Overview of Component Middleware

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

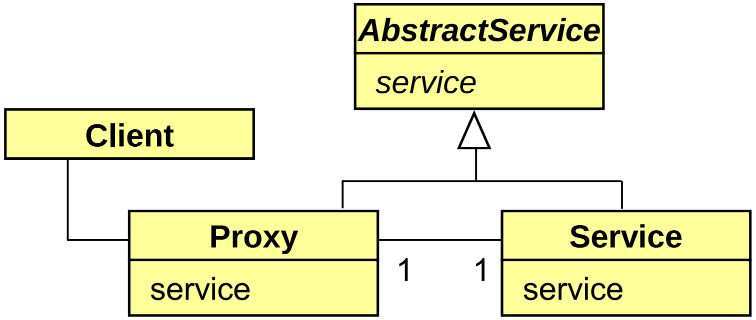


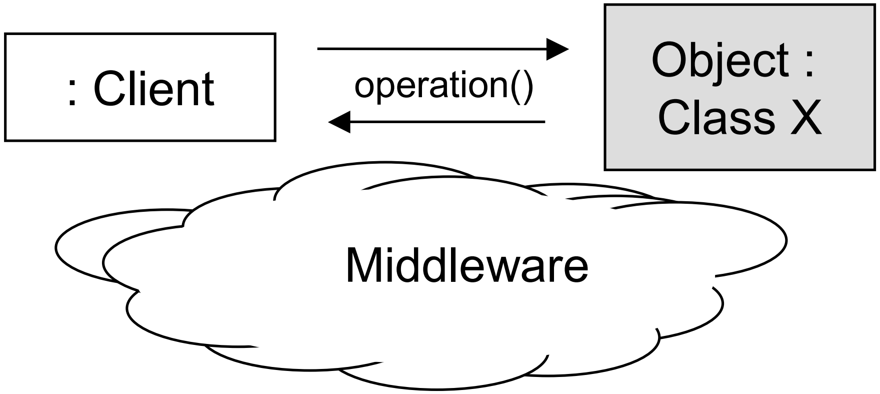


Well-written OO programs exhibit recurring structures that promote abstraction, flexibility, modularity, elegance

Motivations for Applying OO to Network Programming

- Abstract away lower-level OS & protocol-specific details for network programming
- Create distributed systems which are easier to model & build
- Result: robust distributed systems built with *distributed object computing middleware*
  - e.g., CORBA, Java RMI, DCOM, etc.

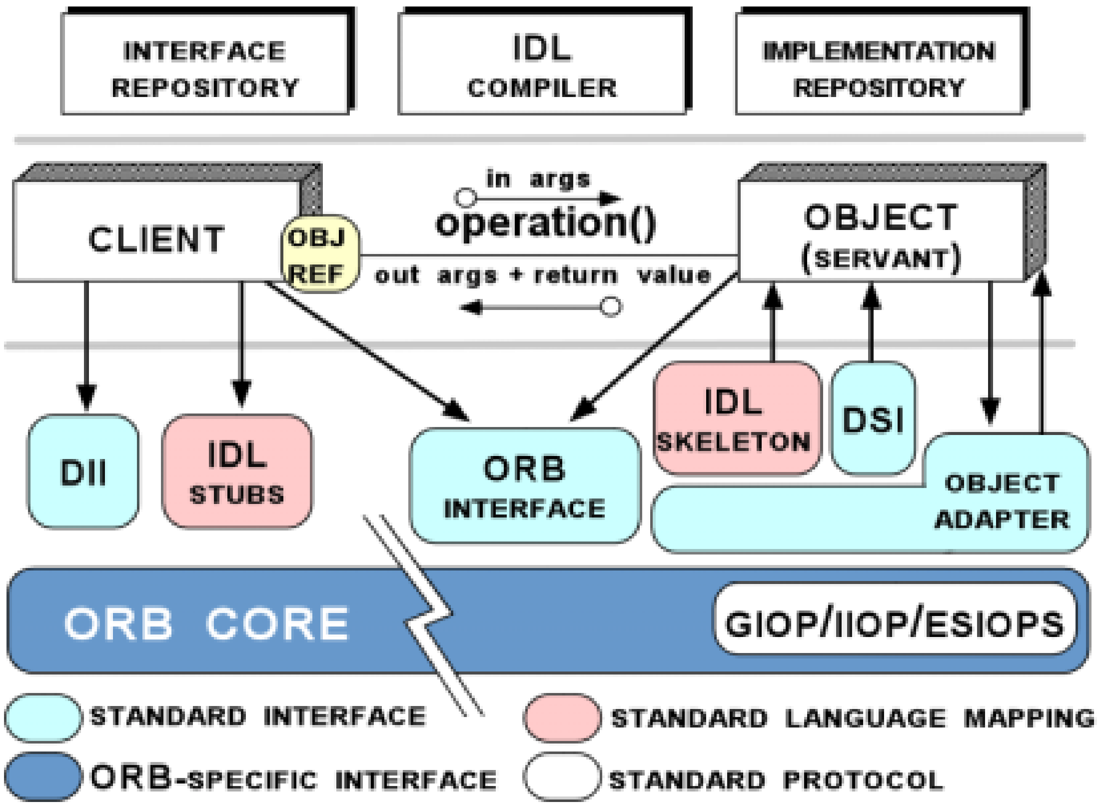




We now have more robust software & more powerful distributed systems

Overview of CORBA 2.x Standard

- CORBA 2.x is *distributed object computing* (DOC) middleware that shields applications from heterogeneous platform *dependencies*
  - *e.g.,* languages, operating systems, networking protocols, hardware



- CORBA 2.x simplifies development of distributed applications by automating/encapsulating
  - Object location
  - Connection & memory mgmt.
  - Parameter (de)marshaling
  - Event & request demultiplexing
  - Error handling & fault tolerance
  - Object/server activation
  - Concurrency
  - Security
- CORBA defines *interfaces & policies*, 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, etc.

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

IDL

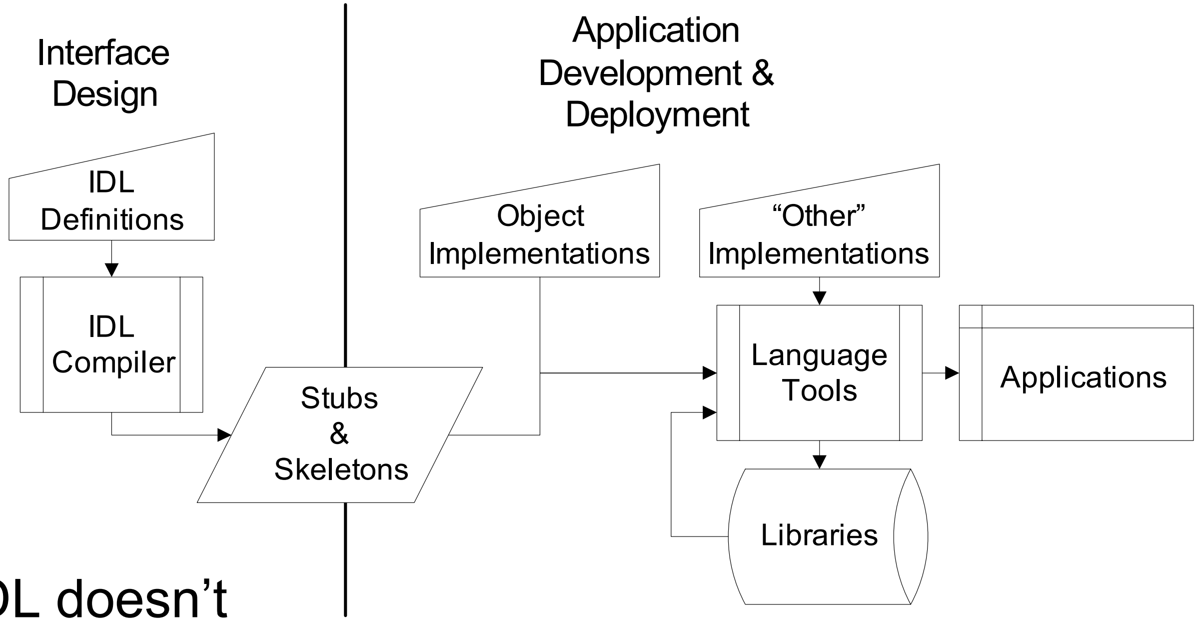
C++

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

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

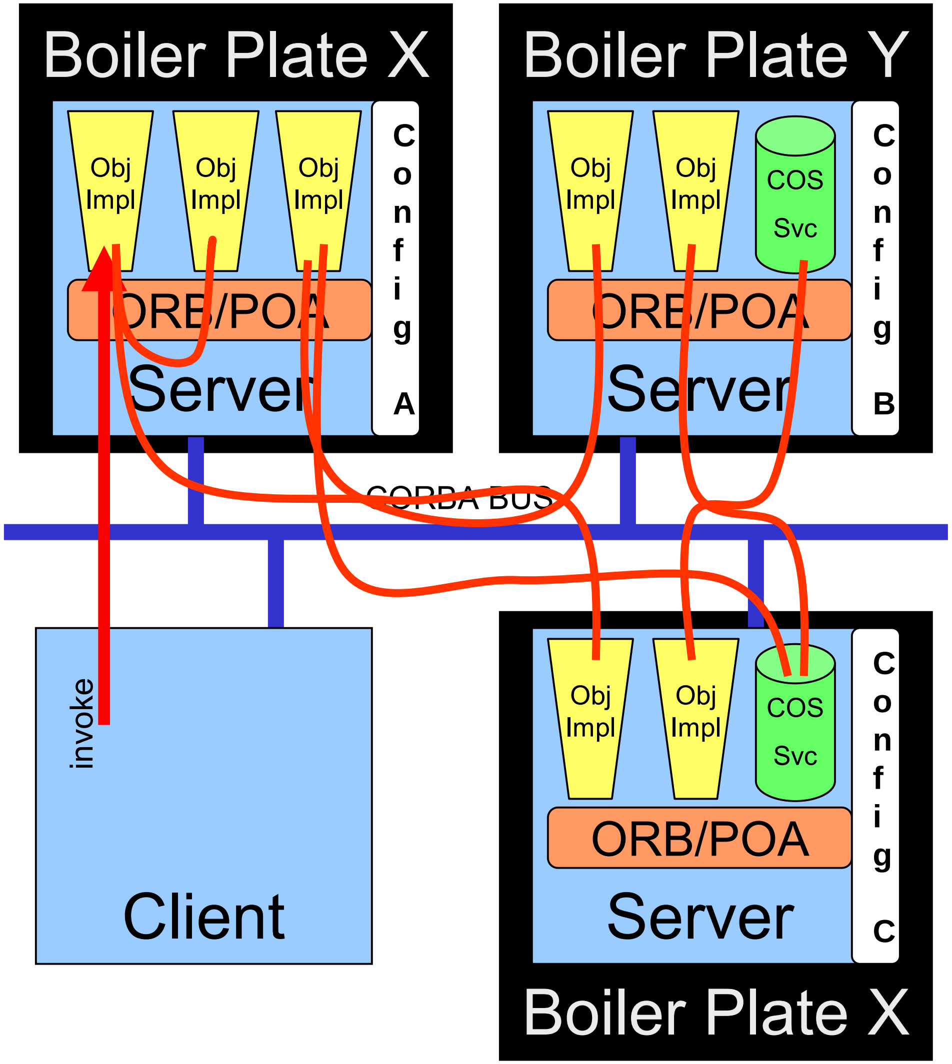
Limitations of OO-based CORBA 2.x Middleware

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 specific service
  - Such "bundling" must be done by developers
- CORBA 2.x doesn't specify how configuration & deployment of objects should be done to create complete applications
  - Proprietary infrastructure & scripts are usually written to facilitate this

Example: Limitations of CORBA 2.x Specification

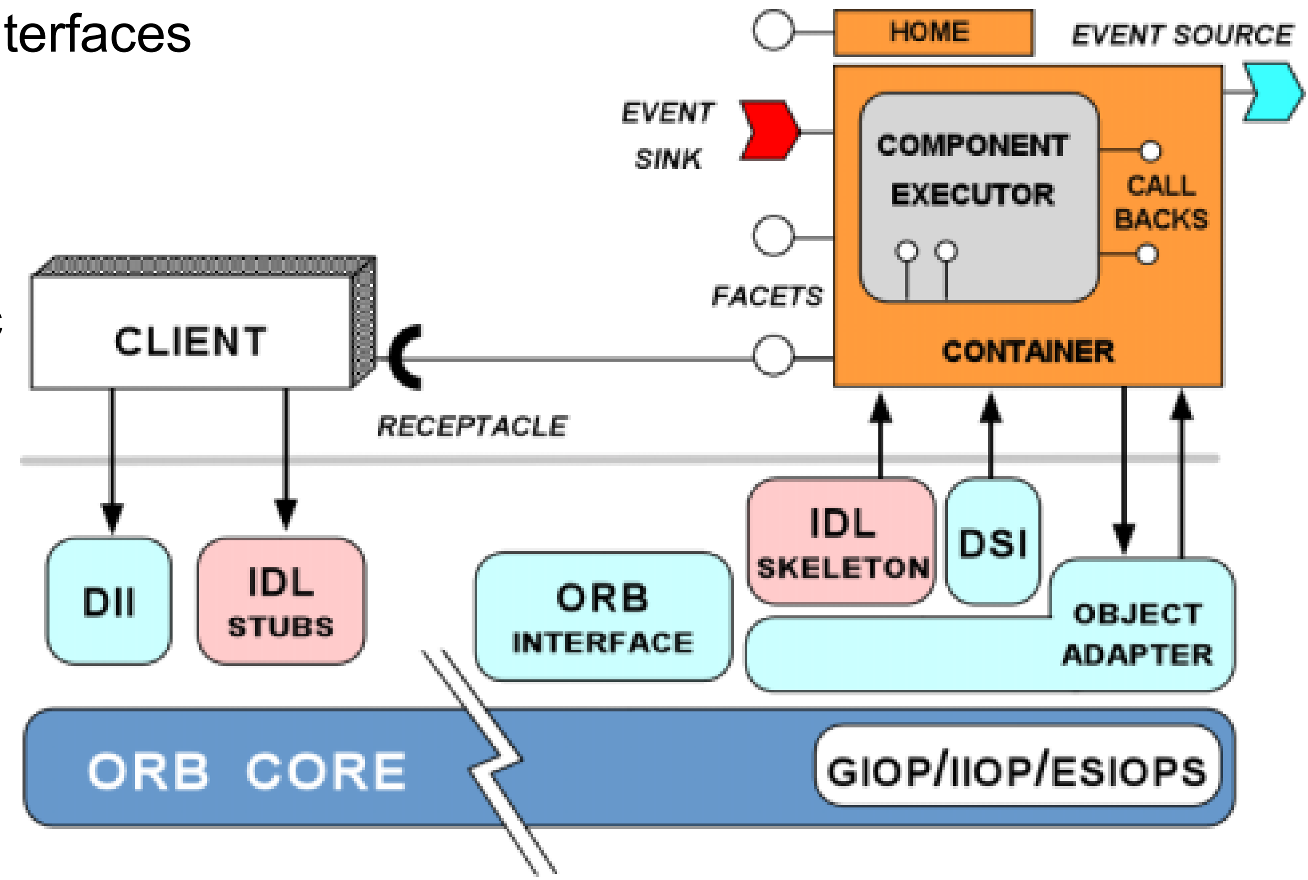


- Requirements of non-trivial DOC applications:
  - Collaboration of multiple objects & services
  - Deployment on diverse platforms
- Limitations – Lack of standards for
  - Server configuration
  - Object/service configuration
  - Application configuration
  - Object/service deployment
- Consequence: tight couplings at various layers
  - Brittle, non-scalable implementation
  - Hard to adapt & maintain
  - Increased time-to-market

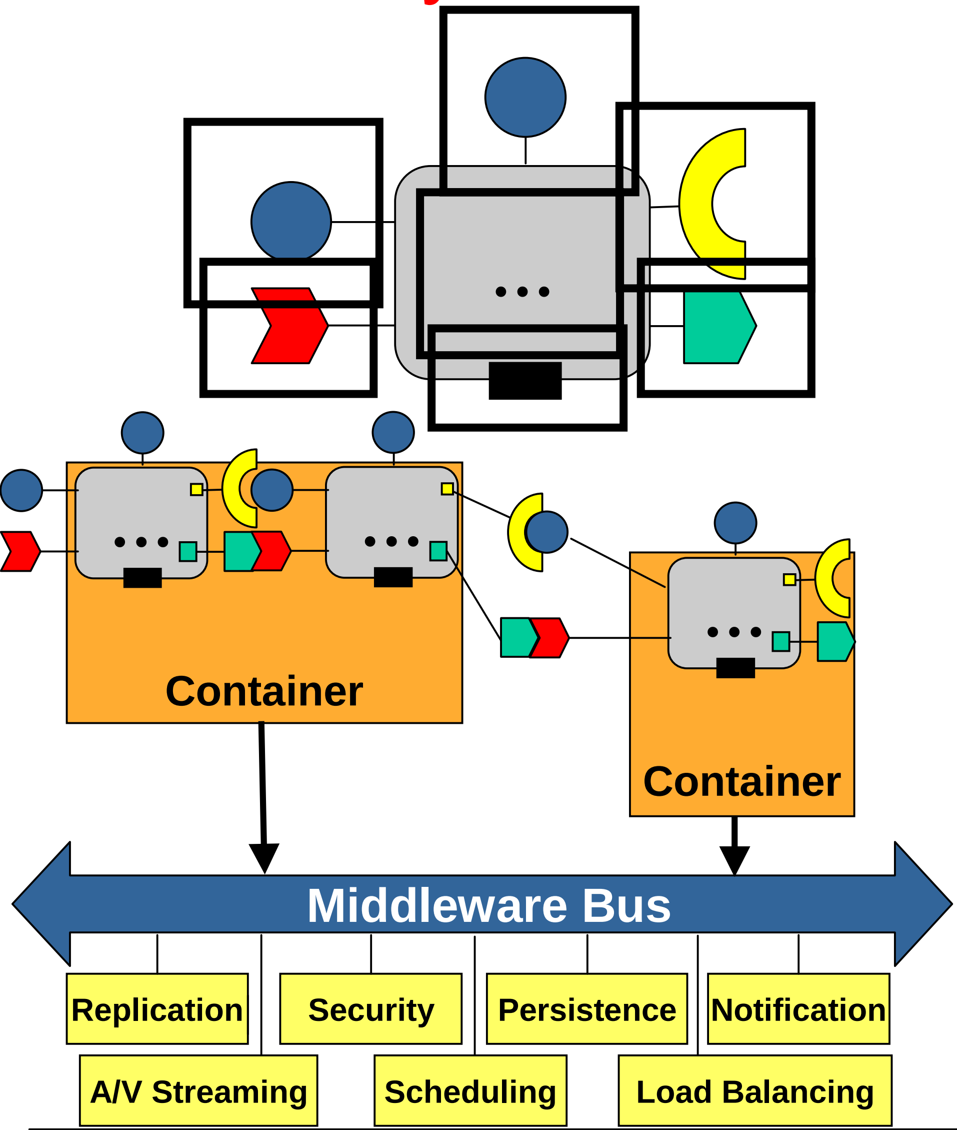
Solution: Component Middleware

Component middleware capabilities:

- 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



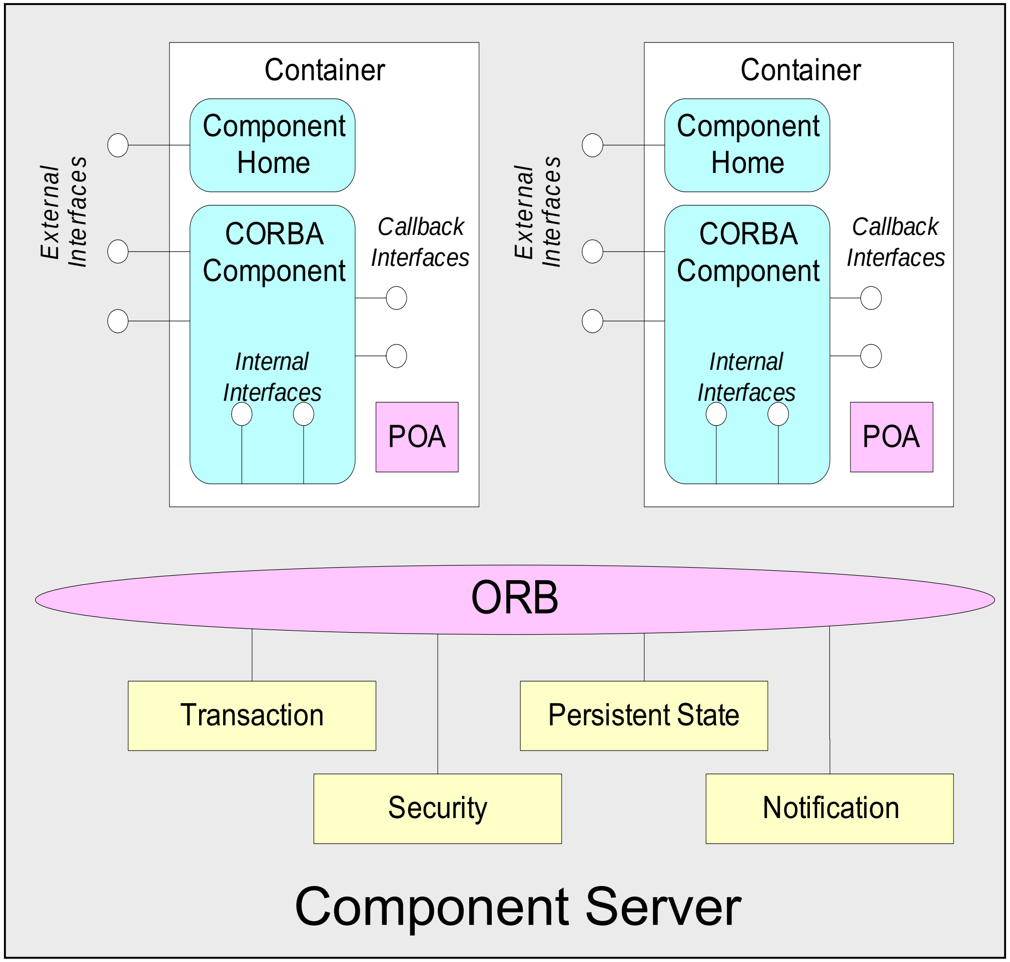
Birdseye View of Component Middleware



- *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*** and
  - Reuse ***common middleware services***

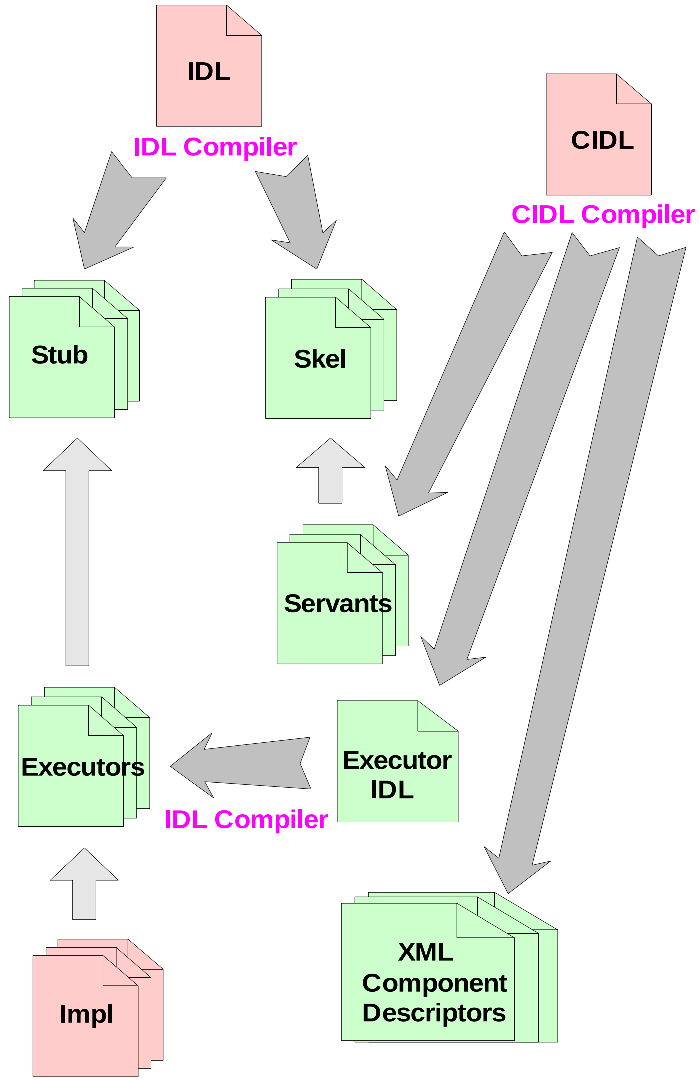
Overview of the CORBA Component Model (CCM)

Capabilities of the CORBA Component Model (CCM)



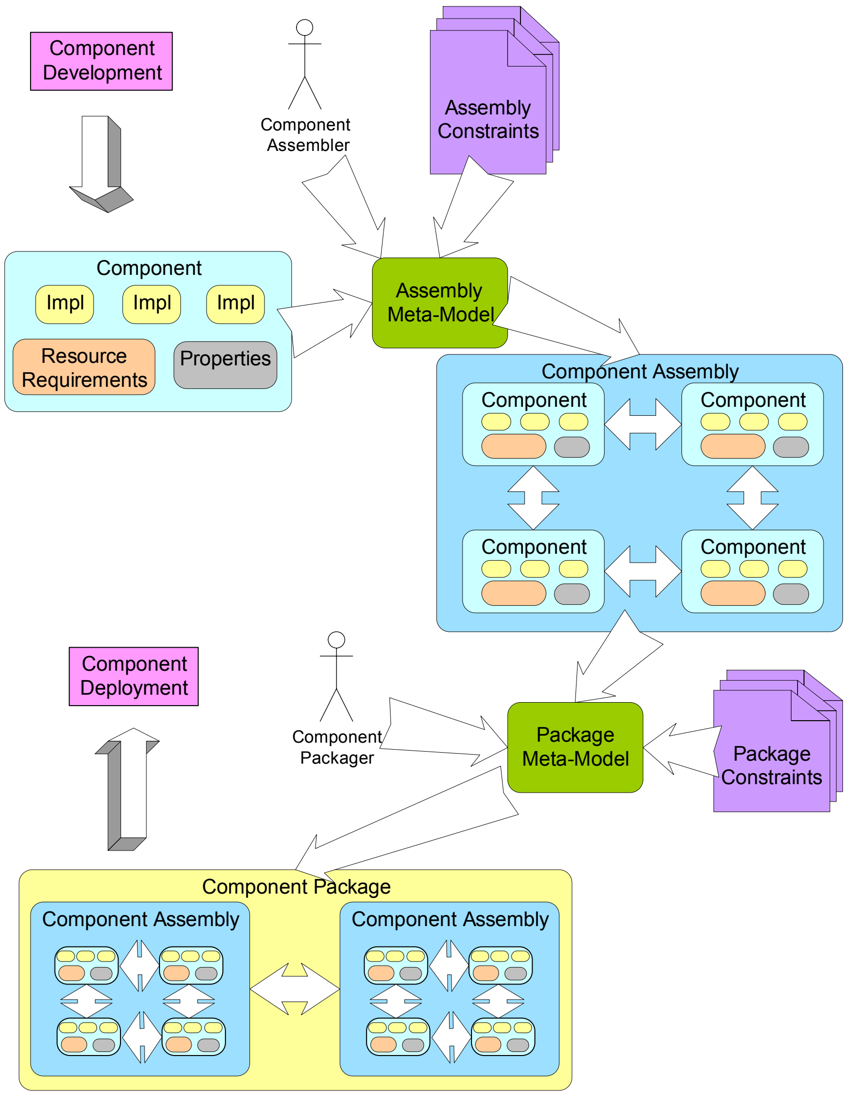
- **Component Server**
  - A generic server process for hosting containers & components/homes
- **Component Implementation Framework (CIF)**
  - Automates the implementation of many component features
- **Component configuration tools**
  - Collect implementation & configuration information into deployable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

Capabilities of the CORBA Component Model (CCM)



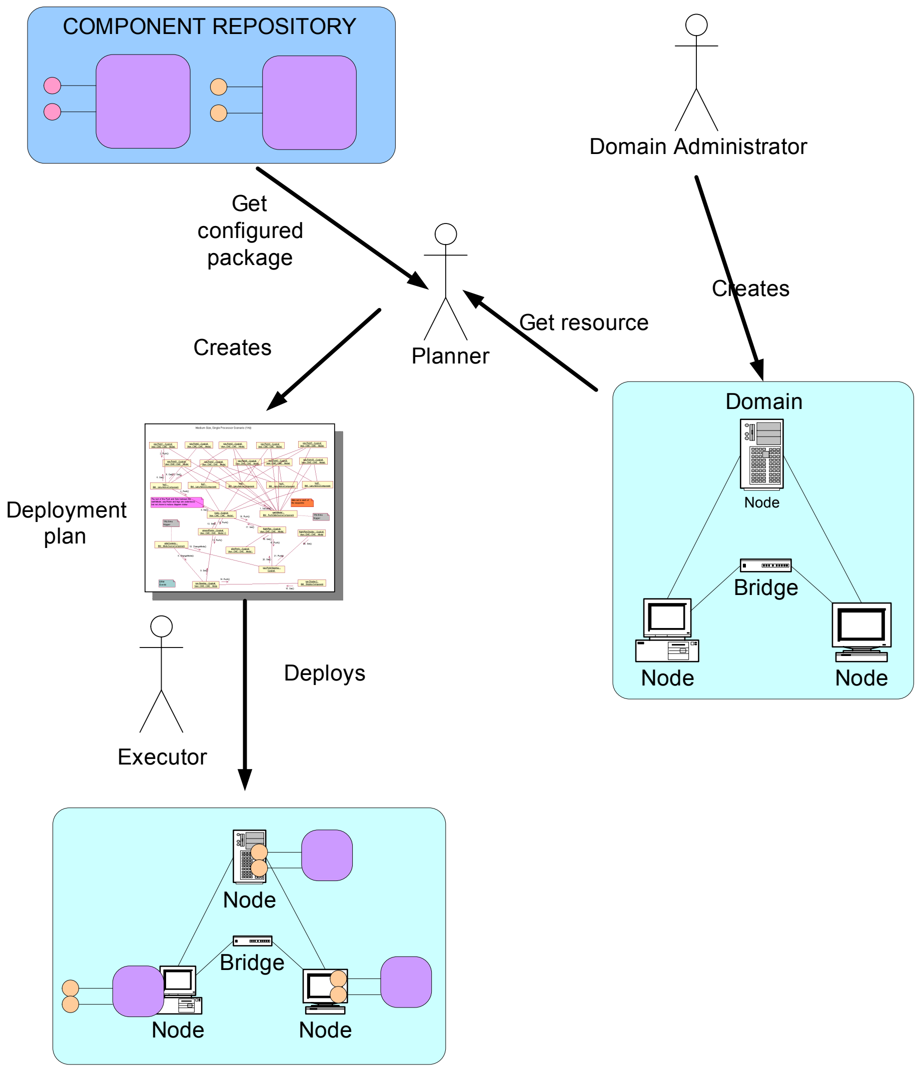
- Component Server
  - A generic server process for hosting containers & components/homes
- **Component Implementation Framework (CIF)**
  - Automates the implementation of many component features
- Component configuration tools
  - Collect implementation & configuration information into deployable assemblies
- 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 & components/homes
- Component Implementation Framework (CIF)
  - Automates the implementation of many component features
- **Component configuration tools**
  - Collect 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 & components/homes
- **Component Implementation Framework (CIF)**
  - Automates the implementation of many component features
- **Component configuration tools**
  - Collect 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	http://www.cpi.com/ejccm/
K2	iCMG	No	C++	http://www.icmgworld.com/products.asp
MicoCCM	FPX	Yes	C++	http://www.fpx.de/MicoCCM/
OpenCCM	ObjectWeb	Yes	Java	http://openccm.objectweb.org/
QoS Enabled Distributed Object (Qedo)	Fokus	Yes	C++	http://qedo.berlios.de/news.php4?lang=eng
StarCCM	Source Forge	Yes	C++	http://sourceforge.net/projects/starccm/

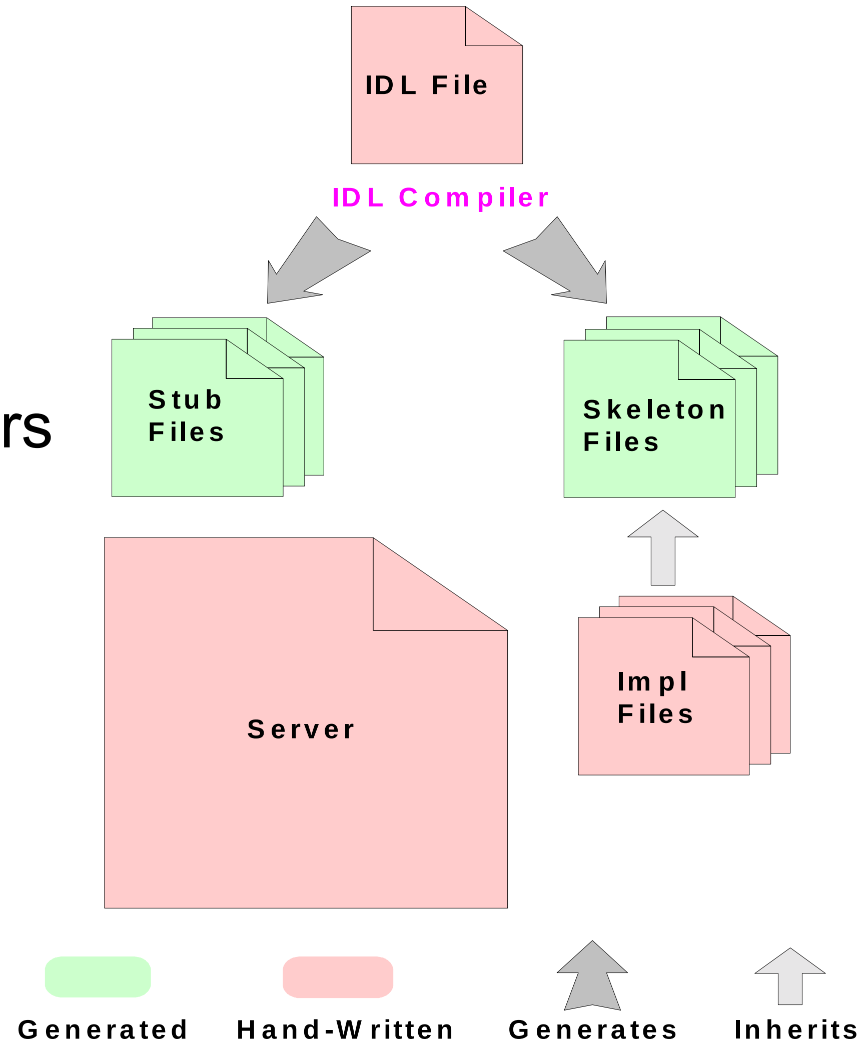
CCM Compared to EJB, COM, & .NET

- Like SUN Microsystems's Enterprise Java Beans (EJB)
  - CORBA components created & managed by homes
  - Run in containers managing system services transparently
  - Hosted by application component servers
  - But can be written in more than Java
- Like Microsoft's Component Object Model (COM)
  - Have several input & output interfaces
  - Both point-to-point sync/async operations & publish/subscribe events
  - Navigation & introspection capabilities
  - But 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 than just Microsoft Windows

Comparing Application Development with CORBA 2.x vs. CCM

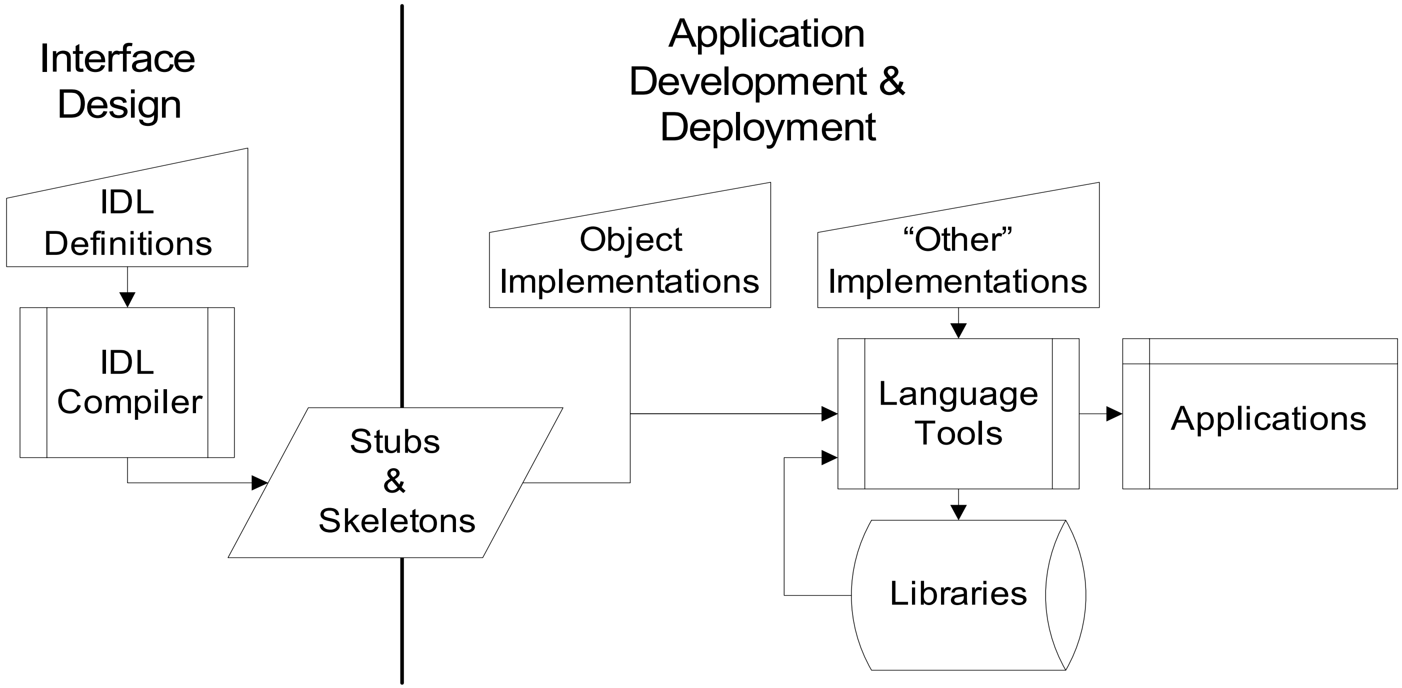
CORBA 2.x User Roles

- Object interface designers
- Server developers
- Client application developers

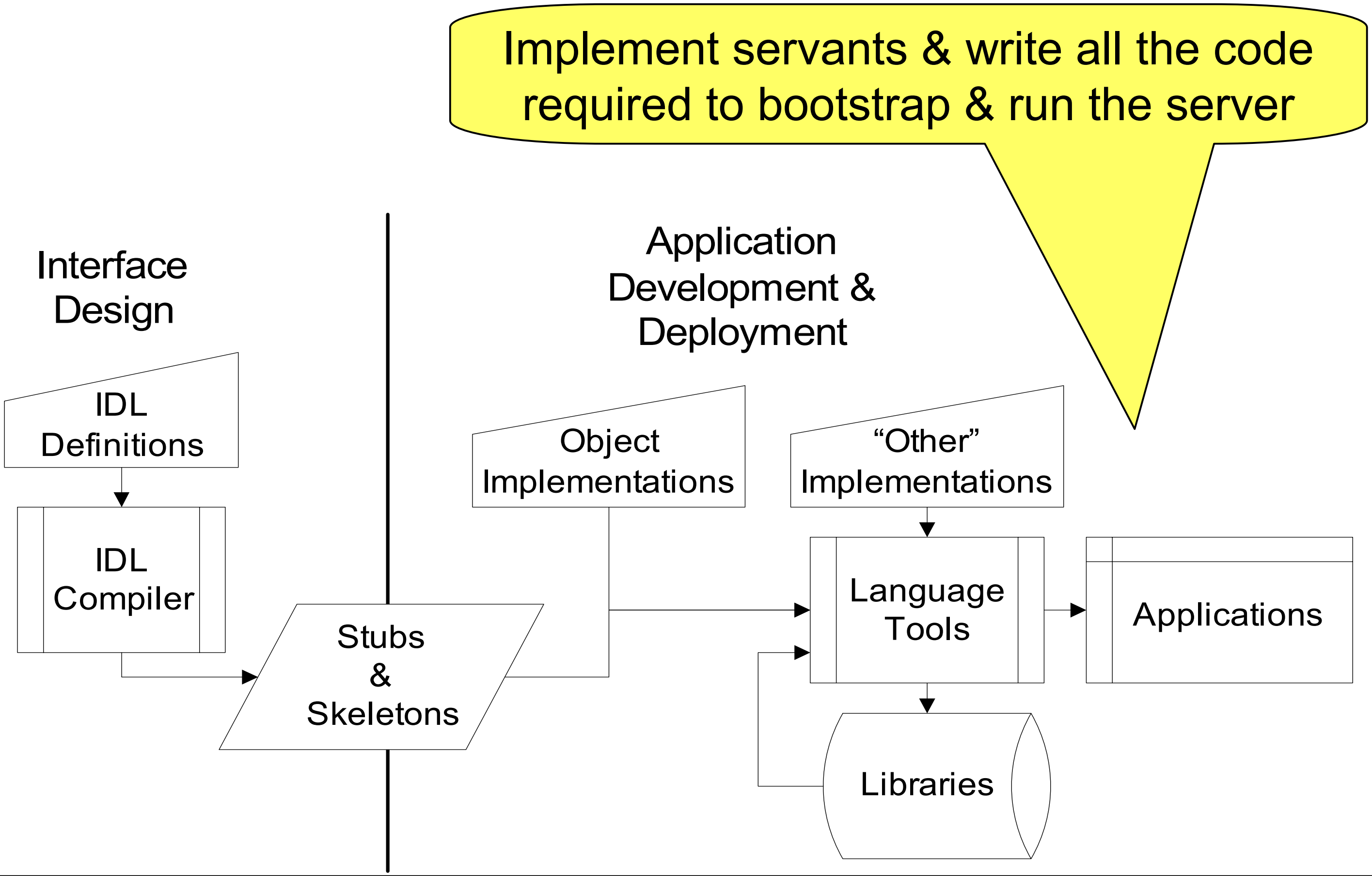


CORBA 2.x Application Development Lifecycle

Specification of IDL interfaces of objects



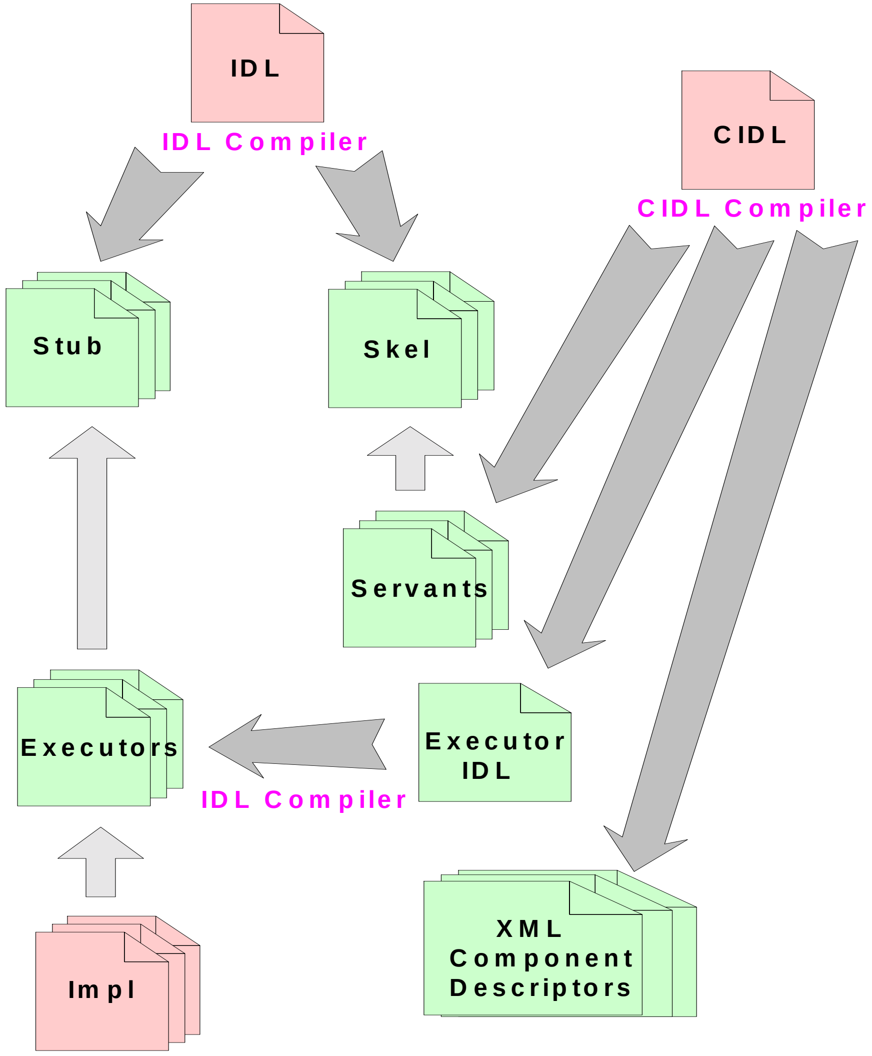
CORBA 2.x Application Development Lifecycle



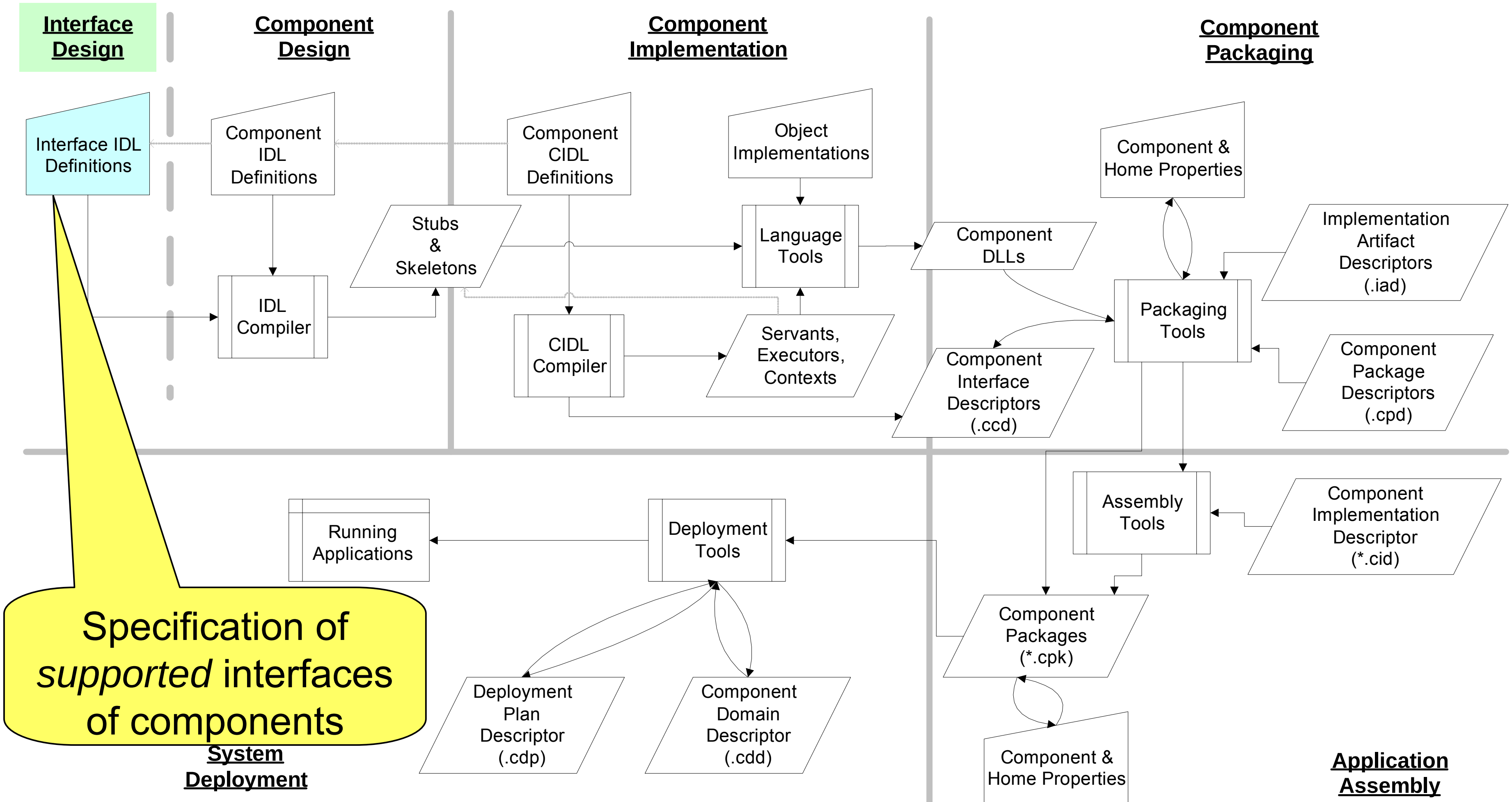
CORBA 2.x supports programming by development (engineering) rather than programming by assembly (manufacturing)

CCM User Roles

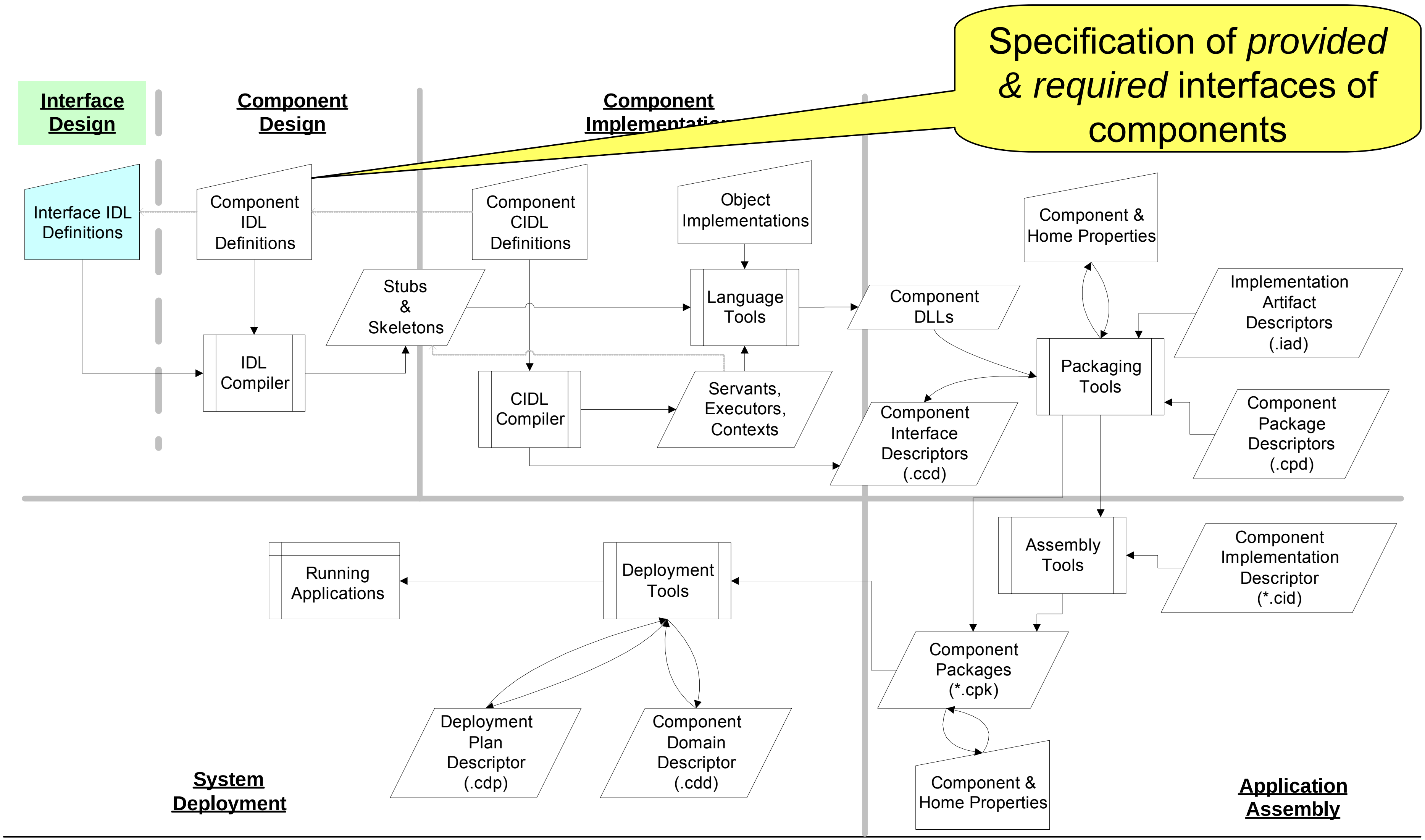
- Component designers
- Component clients
- Composition designers
- Component implementers
- Component packagers
- Component deployers
- Component end-users



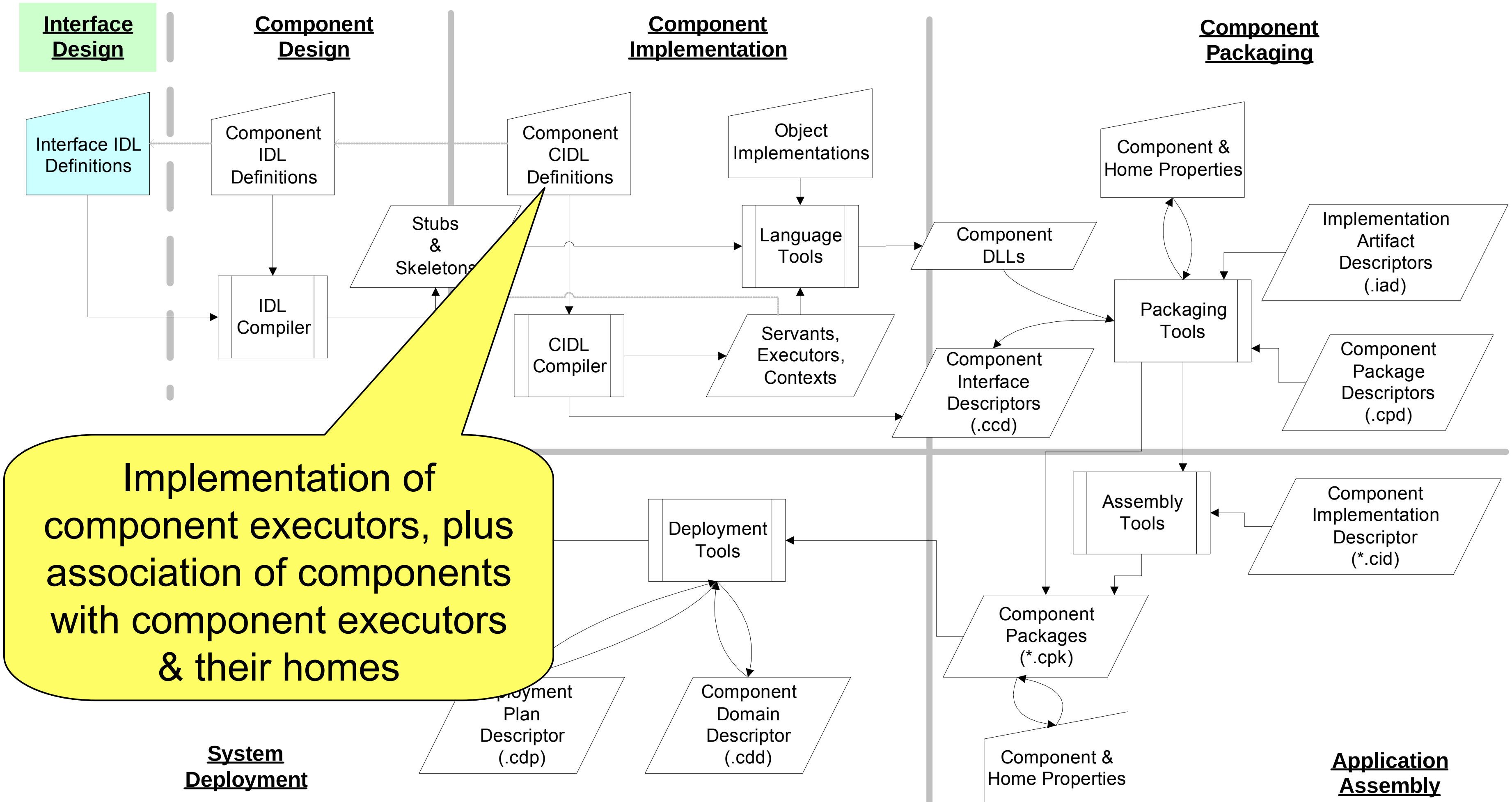
CCM Application Development Lifecycle



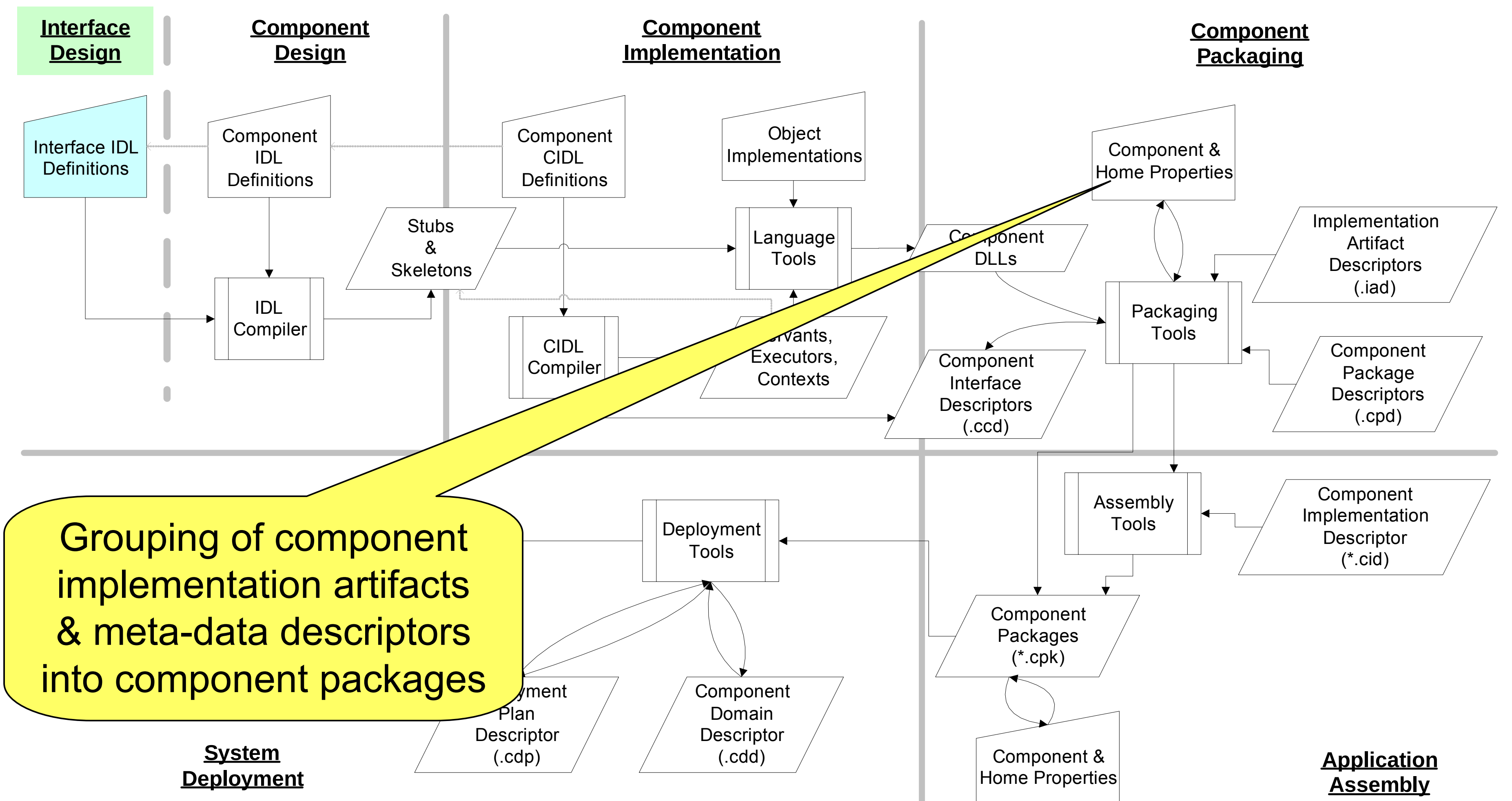
CCM Application Development Lifecycle



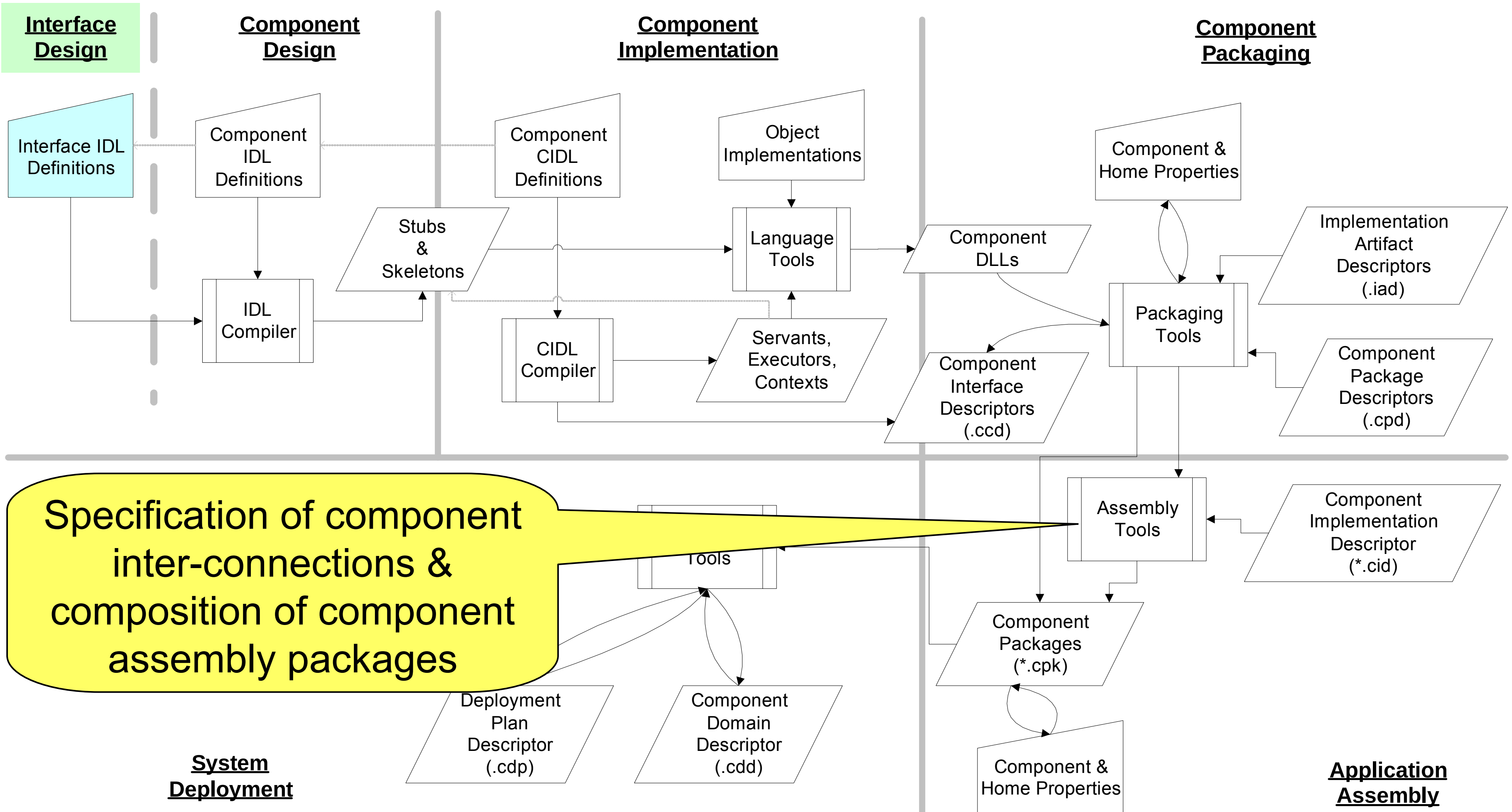
CCM Application Development Lifecycle



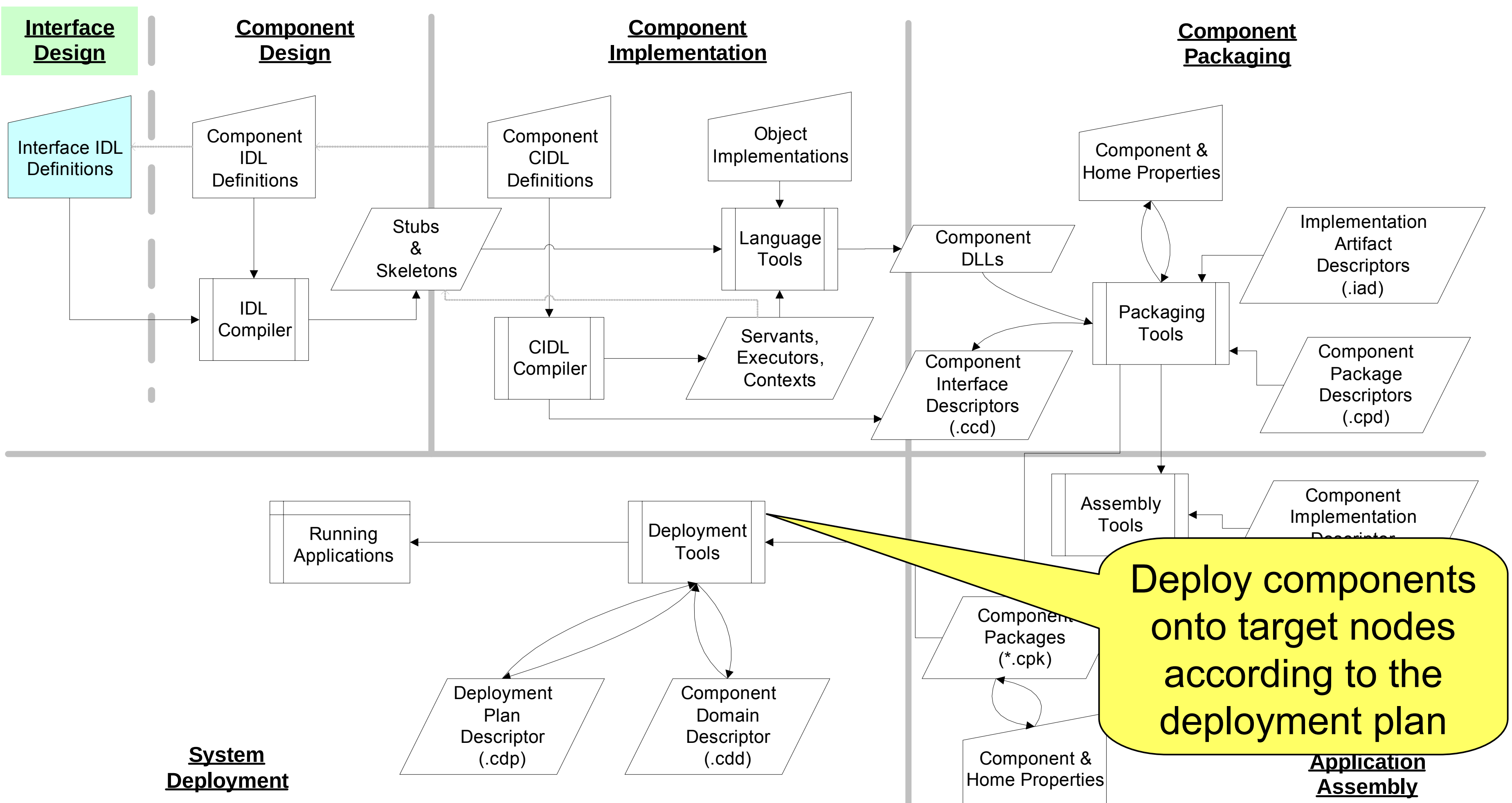
CCM Application Development Lifecycle



CCM Application Development Lifecycle



CCM Application Development Lifecycle



CORBA Component Model (CCM) Features

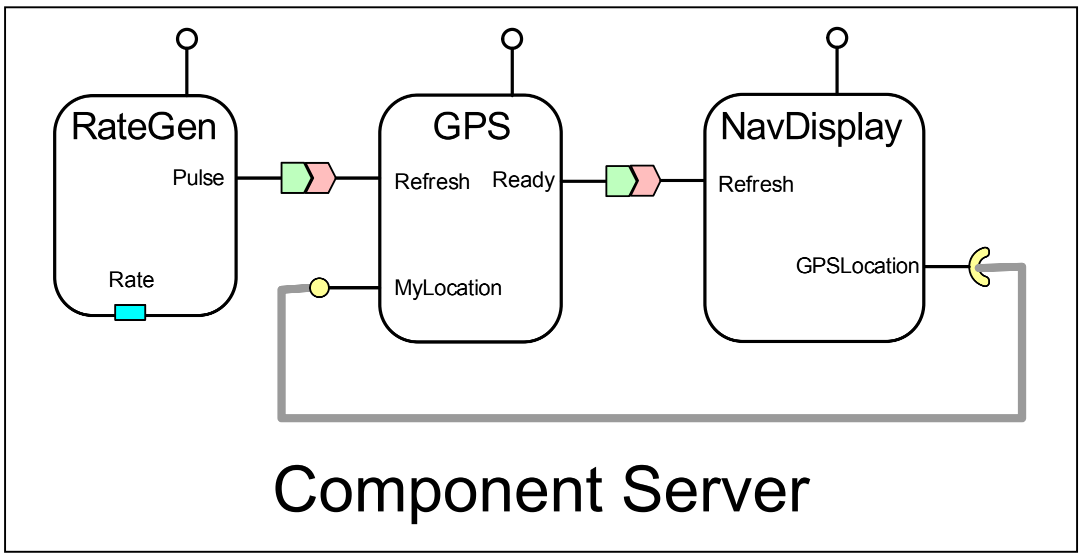
Example CCM DRE Application

Avionics example used throughout tutorial as typical DRE application

Rate Generator

Positioning Sensor

Display Device

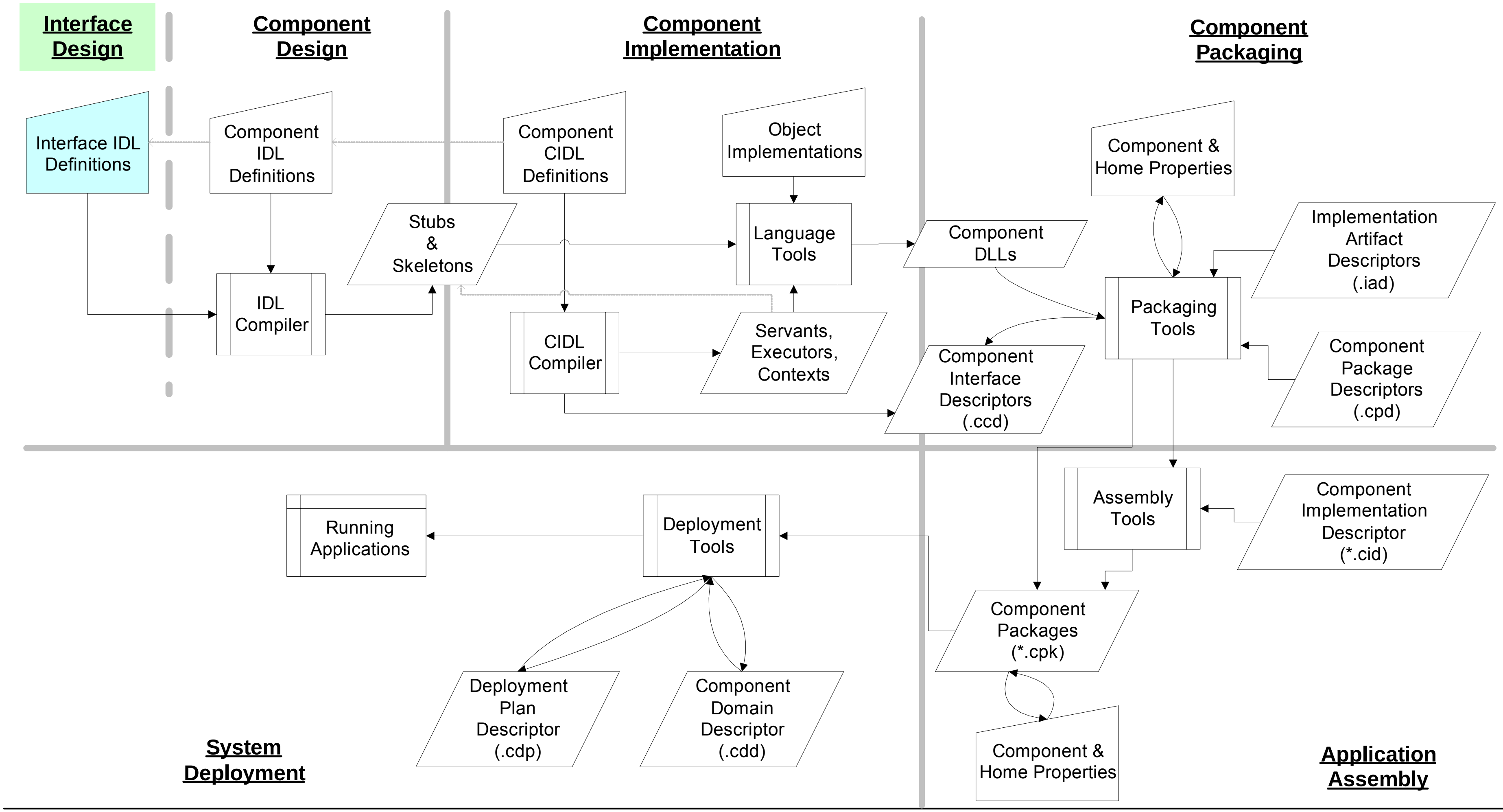


$CIAO_ROOT/examples/OEP/Display/

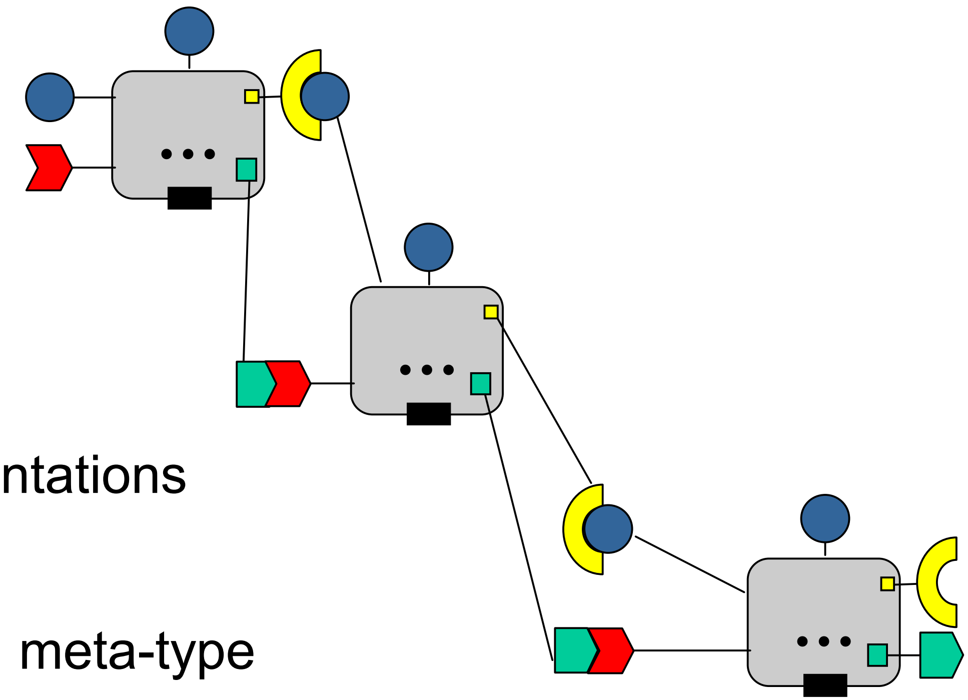
- ***Rate Generator***
  - Sends periodic **`Pulse`** events to consumers
- ***Positioning Sensor***
  - Receives **`Refresh`** events from suppliers
  - Refreshes cached coordinates available thru **`MyLocation`** facet
  - Notifies subscribers via **`Ready`** events
- ***Display Device***
  - Receives **`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 & event sources



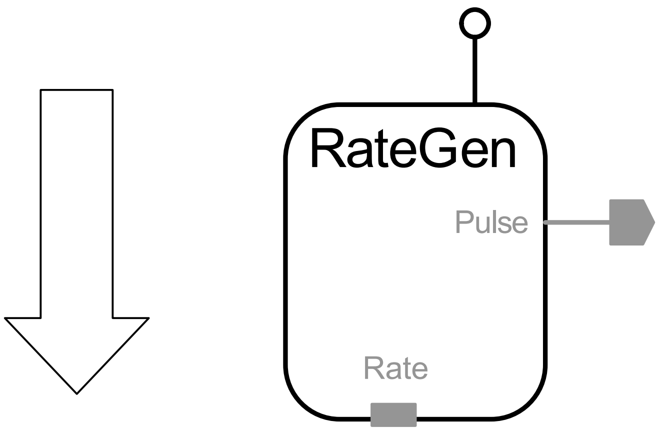
Unit of Business Logic & Composition in CCM



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

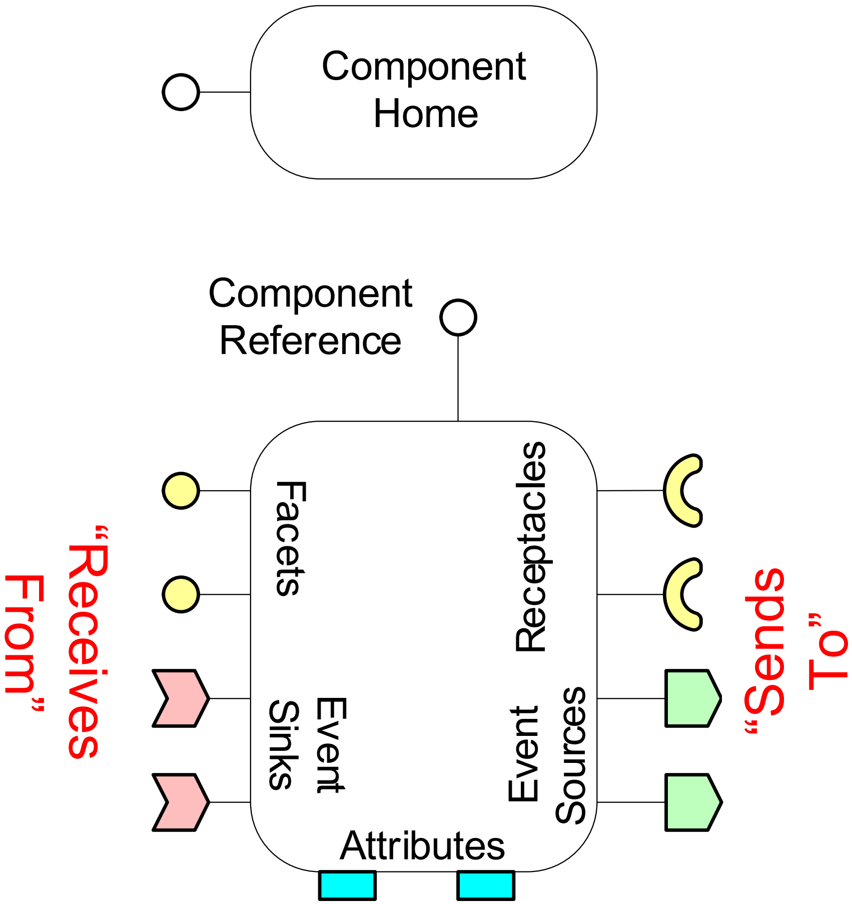



```
// Equivalent IDL 2
interface RateGen :
  Components::CCMObject,
  rate_control {};
```

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

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

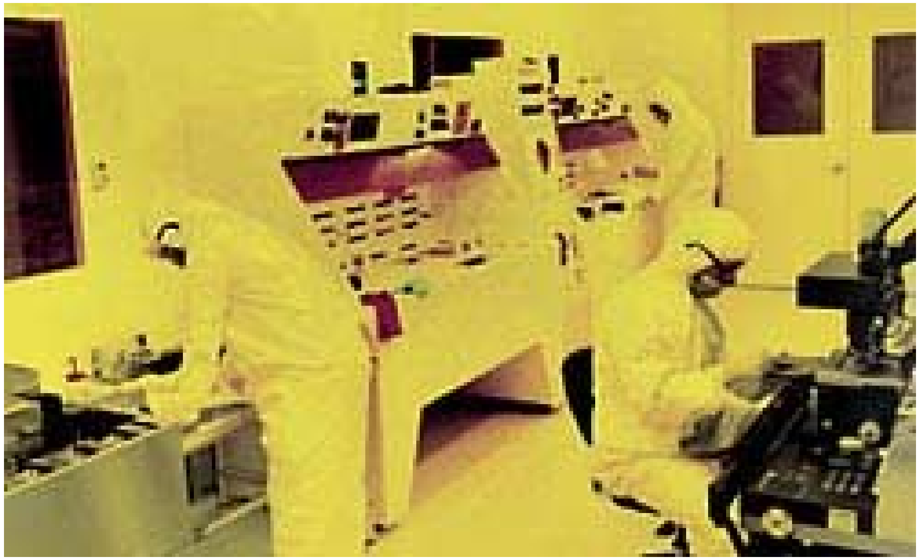
CORBA Component Ports



- 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

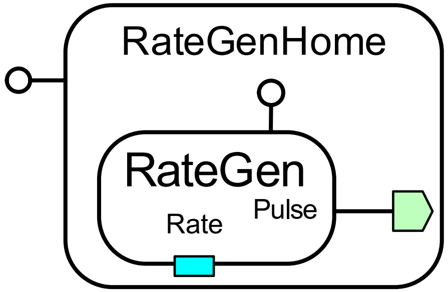


- CCM Solution
  - Integrate lifecycle service into component definitions
  - Use different component *home's* to provide different lifecycle managing strategies
    - Based on the "Factory/Finder" pattern

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

A Quick CCM Client Example

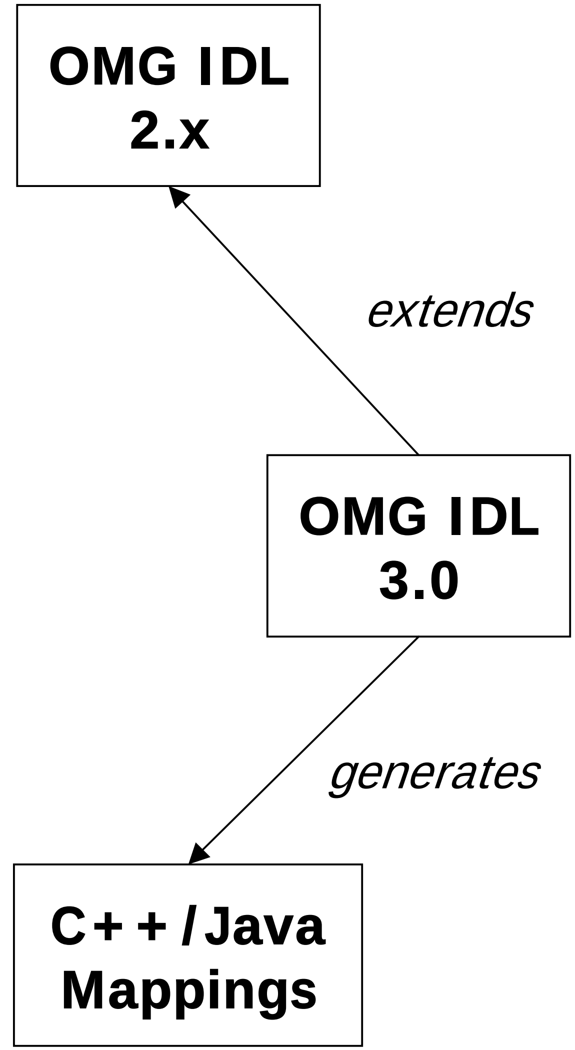
Component & Home for **HelloWorld**

```
interface Hello
{
  void sayHello (in string username);
};


component HelloWorld supports Hello {};


home HelloHome manages HelloWorld {};
```

- IDL 3 definitions for
  - Component: **`HelloWorld`**
  - Managing home: **`HelloHome`**
- Example in **`$CIAO_ROOT/docs/tutorial/Hello/`**

The Client OMG IDL Mapping

- 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
- Permits no change in client programming language mapping
  - Clients still use their favorite IDL-oriented tools, such as CORBA stub generators, etc.
- Clients need not be "component-aware"
  - They 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 ("HelloHome");
12   HelloHome_var hh = HelloHome::_narrow (o);
13   HelloWorld_var hw = hh->create ();
14   hw->sayHello (“Simon”);
15   hw->remove ();
16   return 0;
17 }
```

```
$ ./hello-client
Hello World!  -- from Simon.
```

- Lines 4-10: Perform standard ORB bootstrapping
- Lines 11-12: Obtain object reference to home
- Line 13: 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

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 an 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 to 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 inherit from **`Components::CCMHome`**
  - One for implicit operations generated
  - One inheriting from both previous interfaces

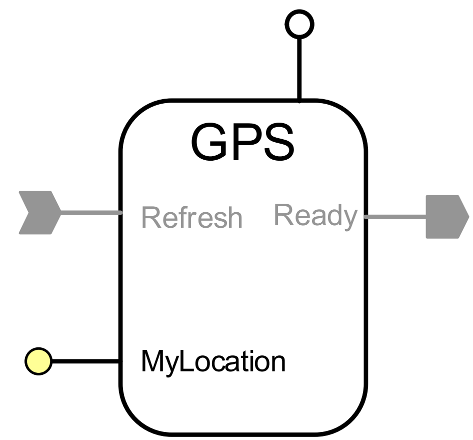
Components Can Offer Different Views

- 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, aka *provided* interfaces, which 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 ();
  …
};
```



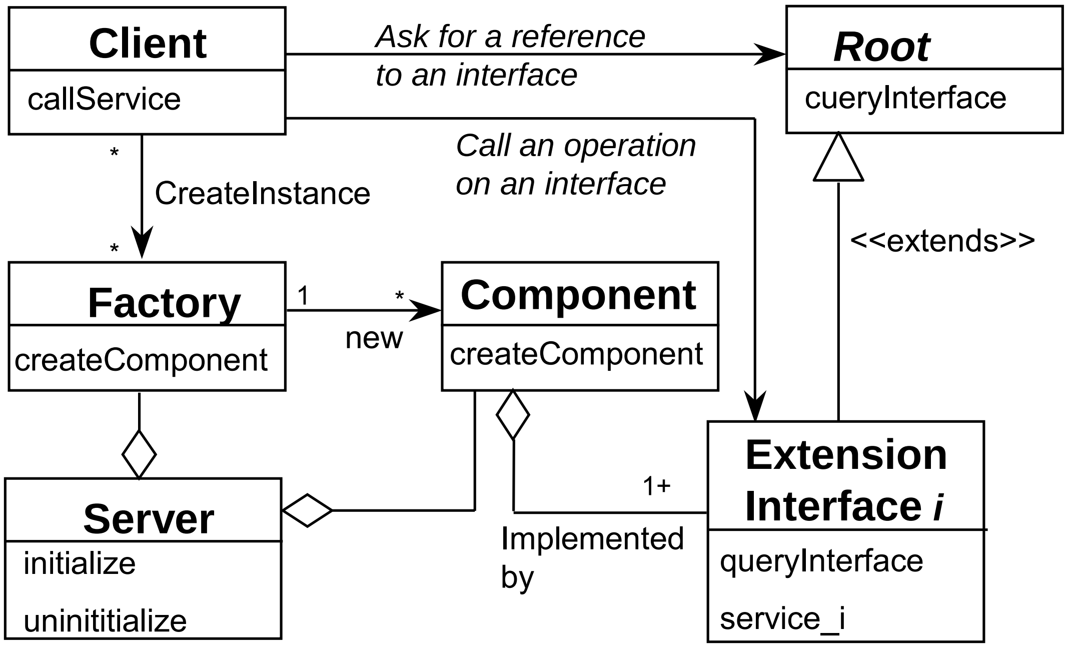
- Component facets:
  - Define *provided* operation interfaces
  - Specified with **`provides`** keyword
  - Represent the component itself, not a separate thing contained by the component
  - Have independent object references
  - Can be used to implement *Extension Interface* pattern

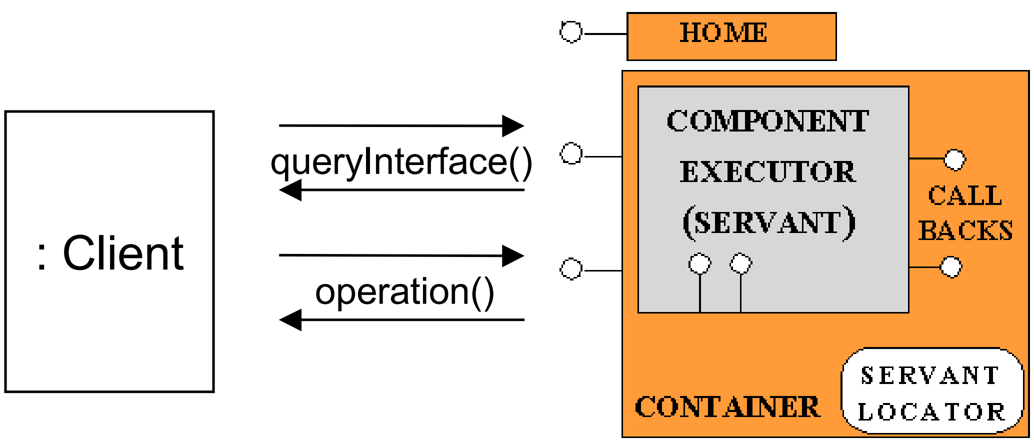
Extension Interface Pattern

The *Extension Interface* design pattern (P2) allows multiple interfaces to be exported by a component to prevent

- breaking of client code &
- bloating of interfaces

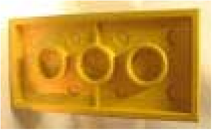
when developers extend or modify component functionality

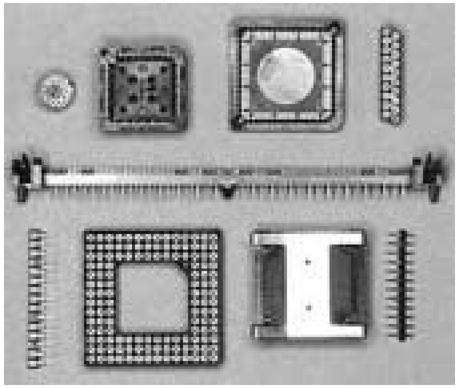


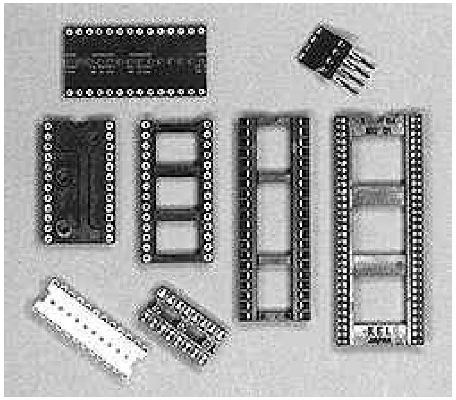


Using Other Components

- Context
  - Components need to collaborate with several different types of components/applications
  - These collaborating components/applications may provide different types of interface
- Problems with CORBA 2.x
  - No standard way to specify dependencies on other interfaces
  - No standard way to connect an interface to a component
- CCM Solution
  - Define receptacles, aka *required* interfaces, which are distinct named connection points for potential connectivity
    - Represents the “bottom of the Lego”





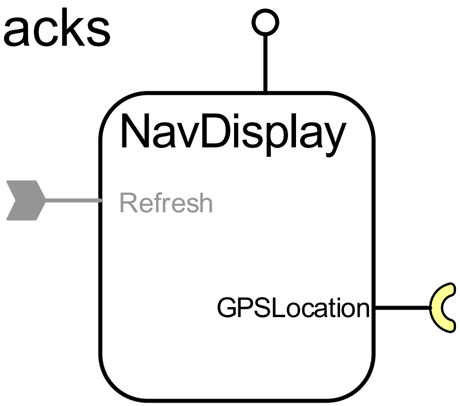


Component Receptacles

```
// 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 ();
  …
};
```

- Component receptacles
  - Specify a way to connect one or more *required* interfaces to this component
  - Specified with **uses** keyword
  - Connections are done statically via configuration & deployment tools during initialization stage or assembly stage
  - Dynamically managed at runtime to offer interactions with clients or other components via callbacks



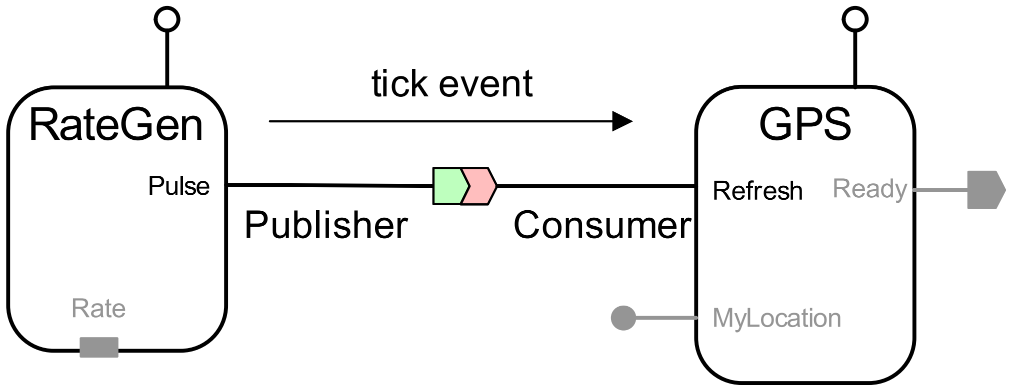
Event Passing

- Context
  - Components may also communicate using anonymous publishing/subscribing message passing mechanism
- Problems with CORBA 2.x
  - Non-trivial to extend existing interfaces to support event passing
  - Standard CORBA Event Service is non-typed → no type-checking connecting publishers-consumers
  - No standard way to specify an object’s capability to generate & process events
- CCM Solution
  - Standard **`eventtype`** & **`eventtype`** consumer interface
  - Event sources & event sinks (“push mode” only)

Component Events

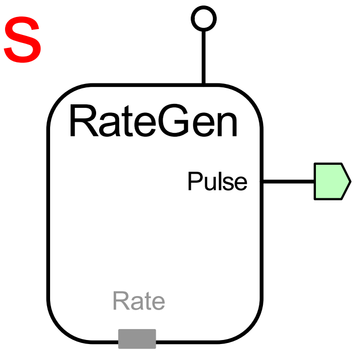
```
// IDL 3
eventtype tick
{
  public rateHz Rate;
};
```

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



- Events are IDL **`valuetypes`**
- Defined with the new IDL 3 **`eventtype`** keyword

Component Event Sources




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

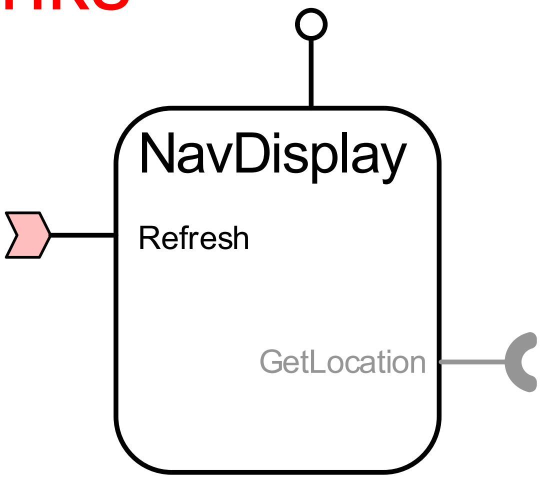
```
// Equivalent IDL 2
interface RateGen :
  Components::CCMObject {
  Components::Cookie
    subscribe_Pulse
    (in tickConsumer c);
  tickConsumer
    unsubscribe_Pulse
    (in Components::Cookie ck);
  …
};
```

- Event sources
  - Named connection points for event production
  - Two kinds: *publisher & emitter*
    - **publishes** = may be multiple consumers
    - **emits** = only one consumer
- Event delivery
  - Consumer subscribes/connects directly
  - CCM container mediates access to CosNotification channels or other event delivery mechanism

Component Event Sinks

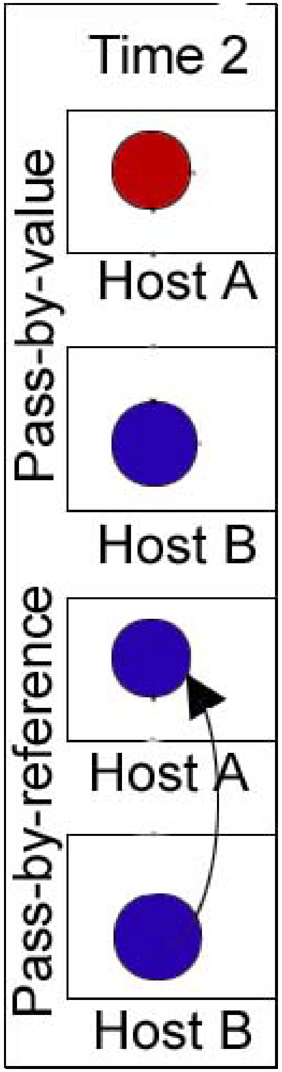
```
// IDL 3
component NavDisplay
{
 …
 consumes tick Refresh;
};
```

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



- Event sinks
  - Named connection points into which events of a specific type may be pushed
  - Event sink can subscribe to multiple event sources
  - No distinction between emitter & publisher

CORBA Valuetypes



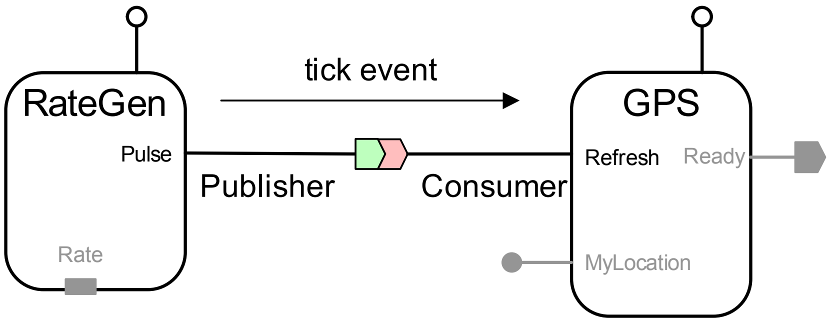
- Context
  - Parameters of IDL operations that are an **`interface`** type always have *pass-by-reference* semantics (even **`in`** parameters)
  - IDL interfaces hide implementations from clients
- Problems
  - Clients cannot instantiate CORBA objects
  - IDL **`structs`** are passed by value, but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

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 be tested reliably for equivalence
- CCM Solution
  - **`Cookie valuetype`**
    - Generated by receptacle or event source implementation
    - Retained by client until needed for **`disconnect()`**

CCM Events




```
valuetype tick :
  Components::EventBase {…};

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

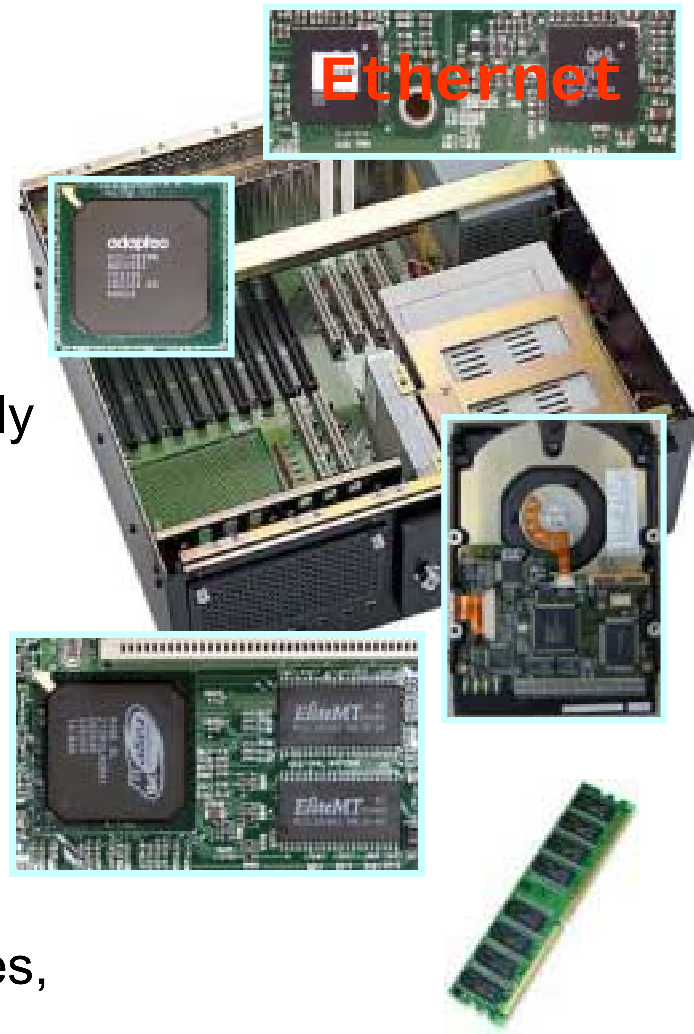
```
class tickConsumer
{
  virtual void push_event
   (Components::EventBase *evt);
  …
};
```

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

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

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

The Need to Configure Components

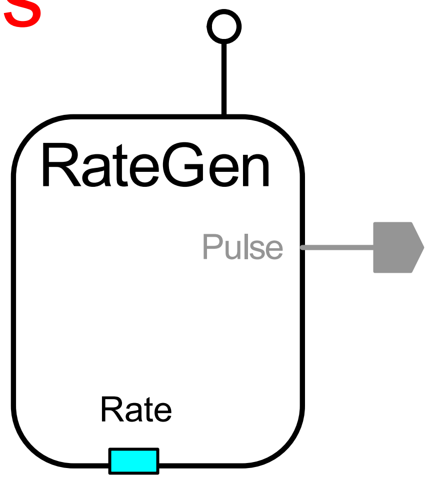
- Context
  - To make component implementations more adaptable, components should be reconfigurable
- Problems
  - Should not commit to a configuration too early
  - No standard way to specify component's configurable knobs
  - Need standard mechanisms to configure components
- CCM Solution
  - Configure components via *attributes* in assembly/deployment environment, by homes, and/or during implementation 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;
};
```

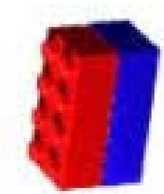


- Named configurable properties
  - Intended for component configuration
    - e.g., optional behaviors, modality, resource hints, etc.
  - Could raise exceptions (new CCM capability)
  - Exposed through accessors & mutators

Configuring & Connecting Components

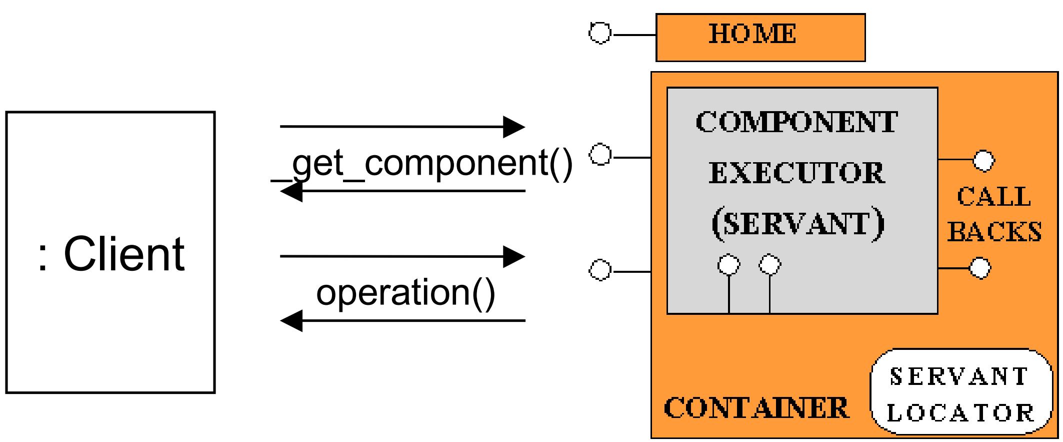
- Context
  - Components need to be configured & connected together to form applications

- Problems
  - Components can have multiple ports with different types & names
  - Non-scalable to write code manually to connect a specific set of components
- CCM Solution
  - Provide introspection interface to discover component capability
  - Provide generic port operations to compose/configure components



CCM Navigation & Introspection

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

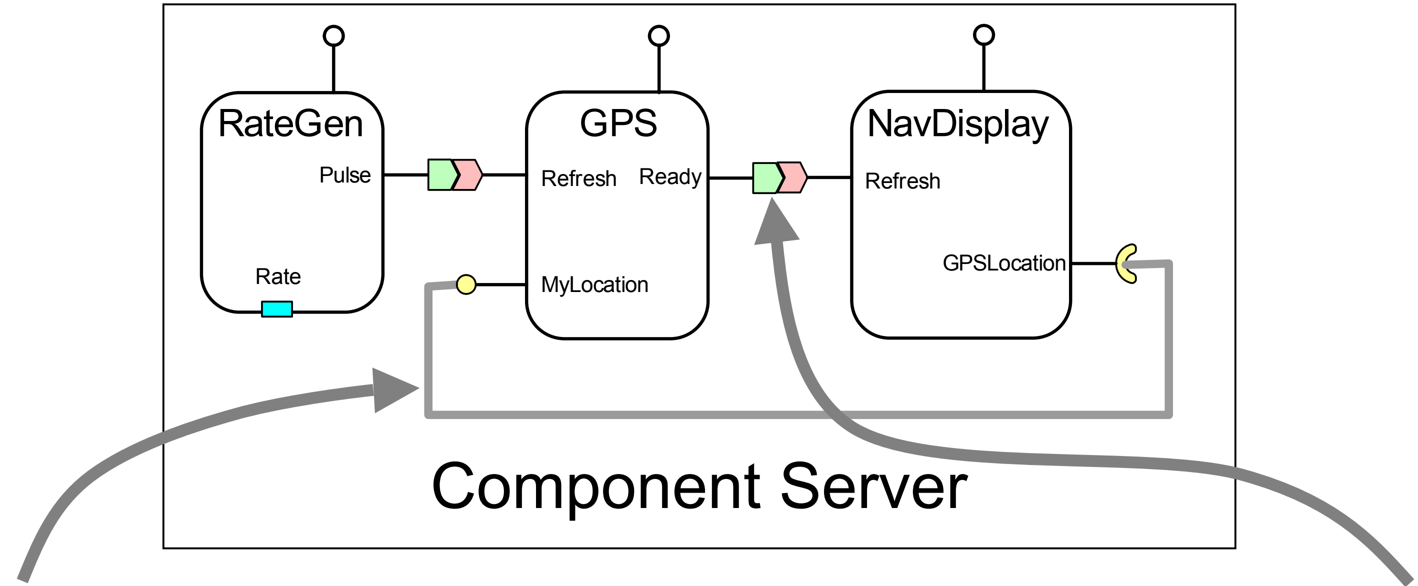


Generic Port Operations

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`**
  - Apply the Extension Interface pattern
  - Used by configuration & deployment tools
  - Lightweight CCM spec doesn't include equivalent IDL 2 operations

Example of Connecting Components



- Facet → Receptacle

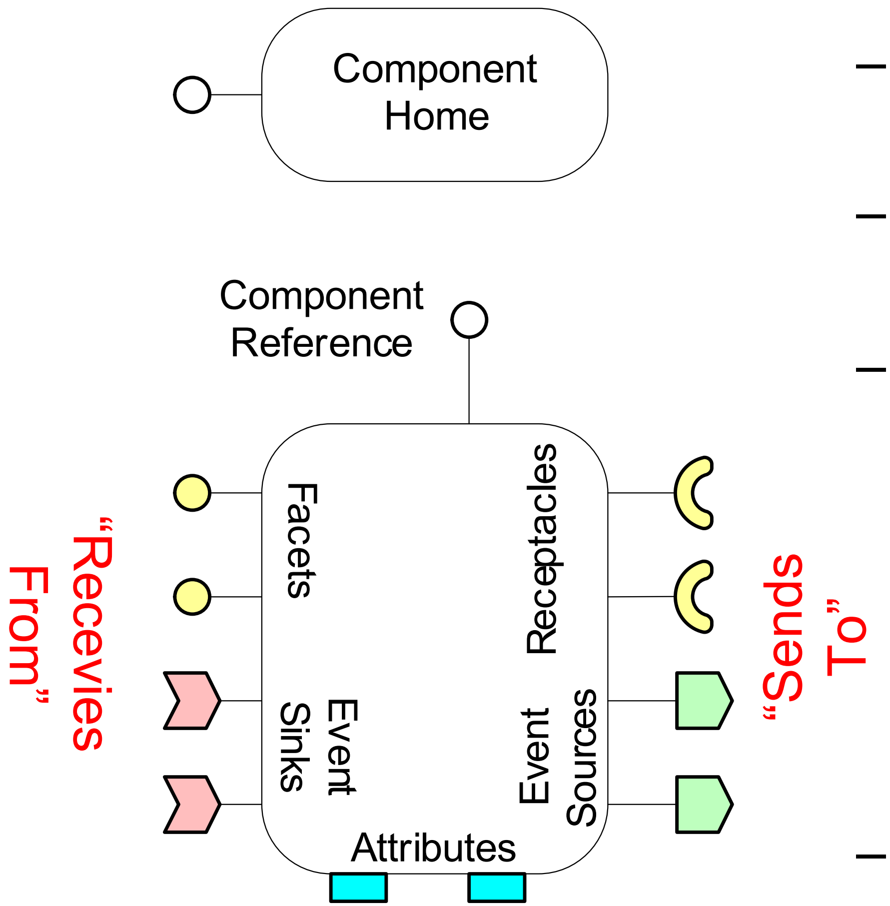
```
objref = GPS->provide
  ("MyLocation");
NavDisplay->connect
  ("GPSLocation", objref);
```

- Event Source → Event Sink

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

CCM components are usually connected via deployment & configuration tools

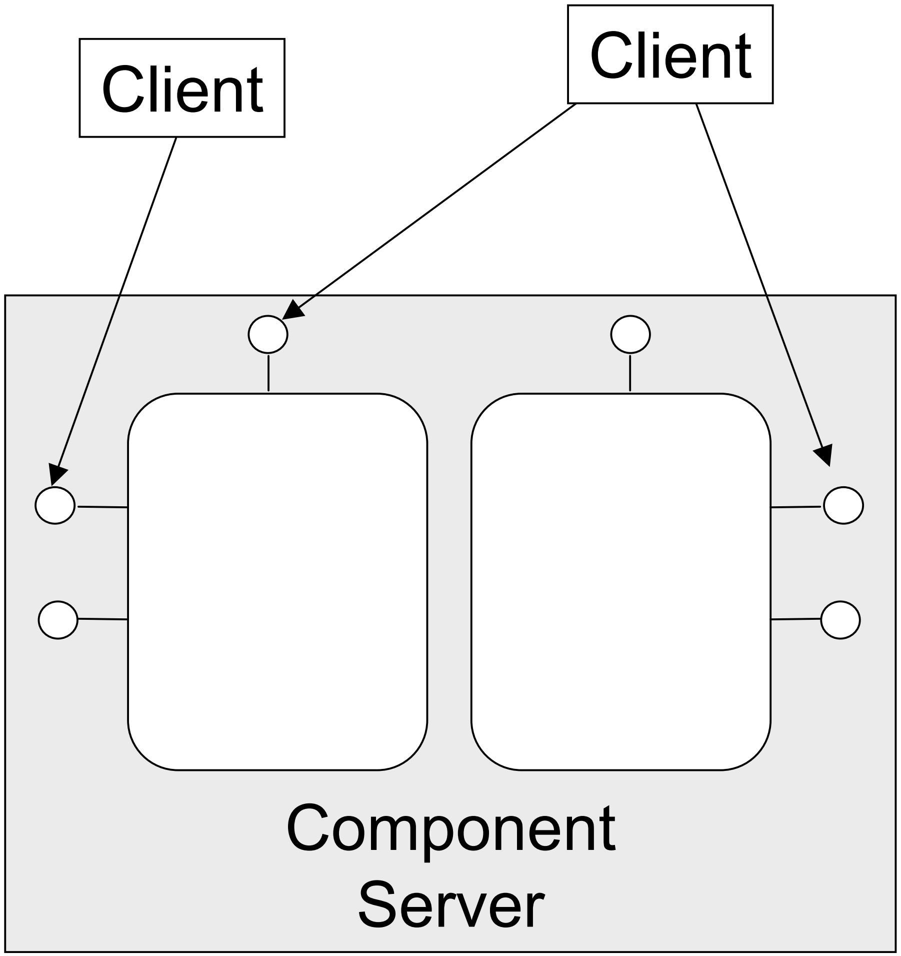
Recap – CCM Component Features



- 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
- Maps to "Equivalent IDL 2 Interfaces"

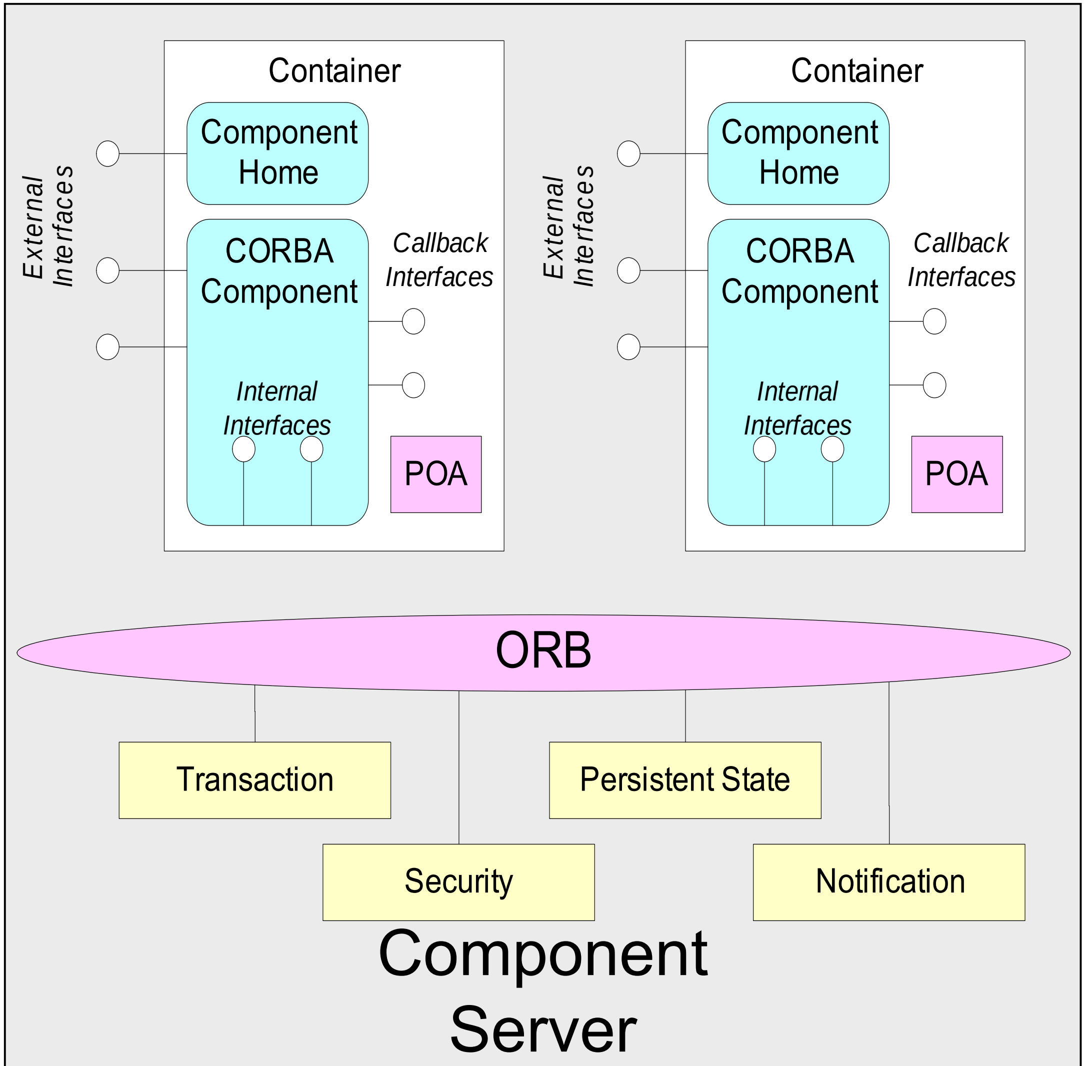
CCM Component Run-time Environment & Containers

CCM Server Features



- CCM focuses largely on component server & application configuration
- Enhance CORBA 2.x by supporting
  - Higher-level abstractions of servant usage models
  - Tool-based configuration & *meta-programming* techniques, e.g.:
    - Reusable run-time environment
    - Drop in & run
    - Transparent to clients

The CCM Container Model



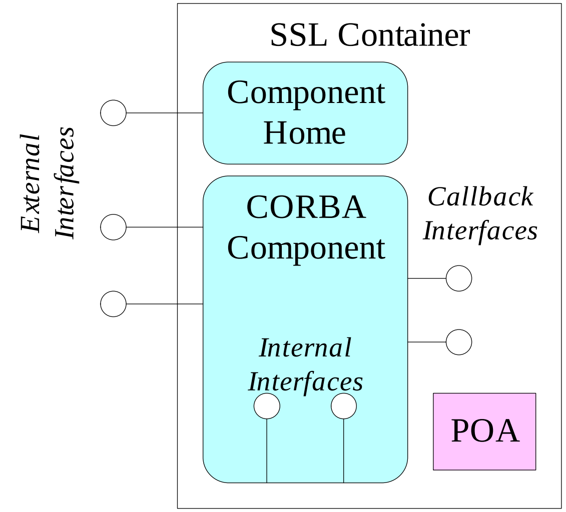
- A framework within component servers
- Built on the Portable Object Adaptor (POA)
  - Automatic activation & deactivation of components
  - Resource usage optimization
- Provides simplified interfaces for CORBA Common Services
  - e.g., security, transactions, persistence, & events
- Uses callbacks for instance management
  - e.g., session states, activation, deactivation, etc.

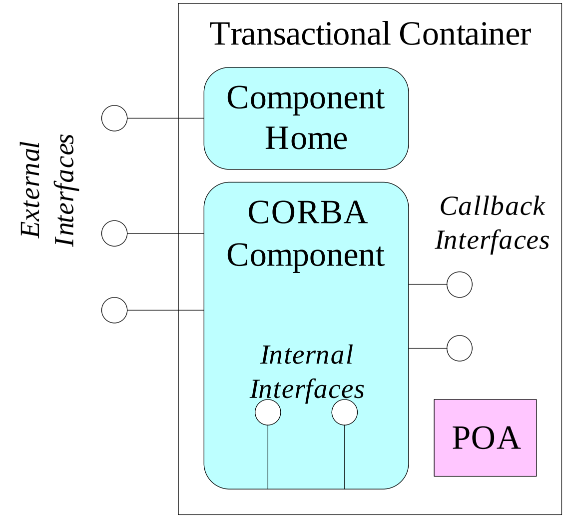
CCM Component/Container Categories

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

Container-managed CORBA Policies

- Goal: decouple run-time configuration from component implementation & configuration
- Specified by component implementers using XML-based meta-data
- Implemented by the container, not the component
  - Uses Interceptor pattern
- CORBA policy declarations defined for:
  - Servant Lifetime
  - Transaction
  - Security
  - Events
  - Persistence





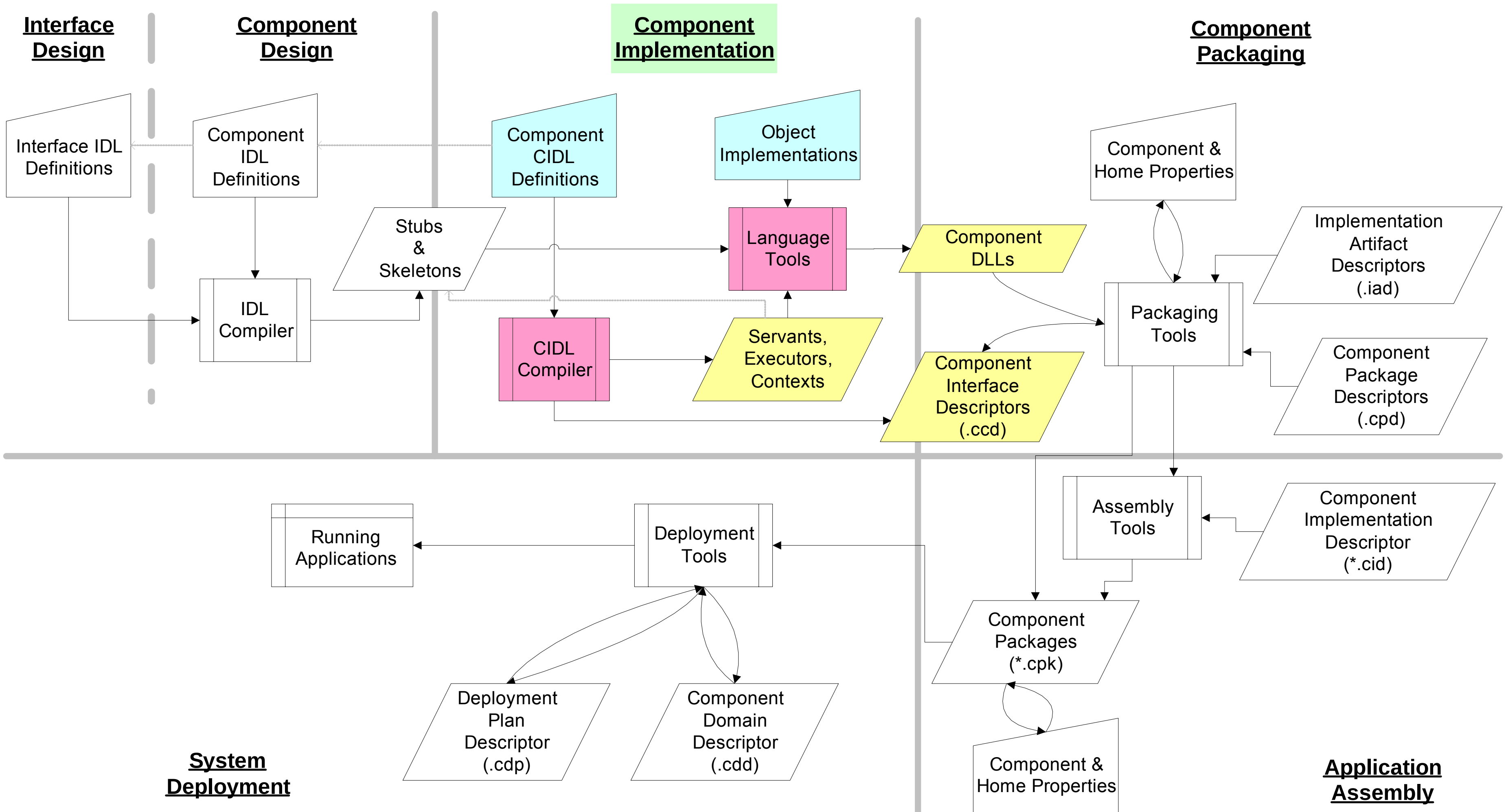
CORBA Implementation Framework (CIF)

&

Component Implementation Definition Language (CIDL)

Component Implementation Stage

Goal: Implement component interfaces & associate them with homes



Requirements for Implementing Components

IDL 3

Component & home definitions

IDL 3 compiler

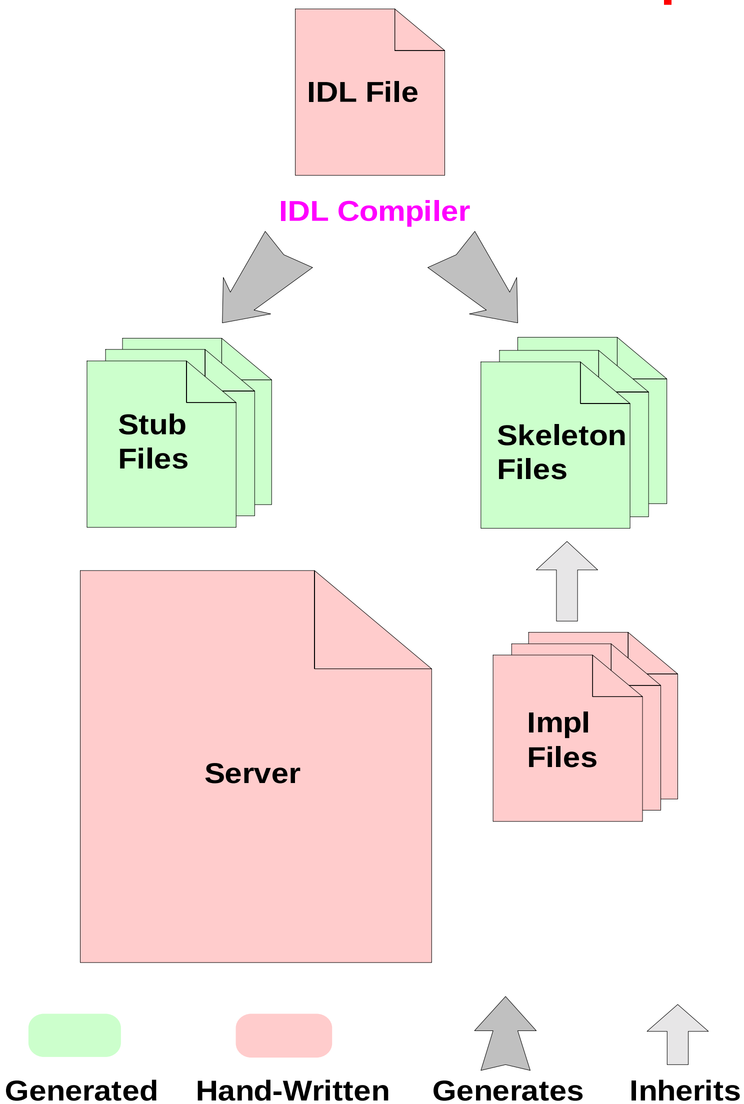
C++/Java/etc.

Component & Home Servants

- Navigation interface operations
- Receptacles interface operations
- Events interface operations
- **`CCMObject`** interface operations
- **`CCMHome`** interface operations
- Implied equivalent IDL 2 port operations
- Application-related operations
  - in facets, supported interfaces, event consumers

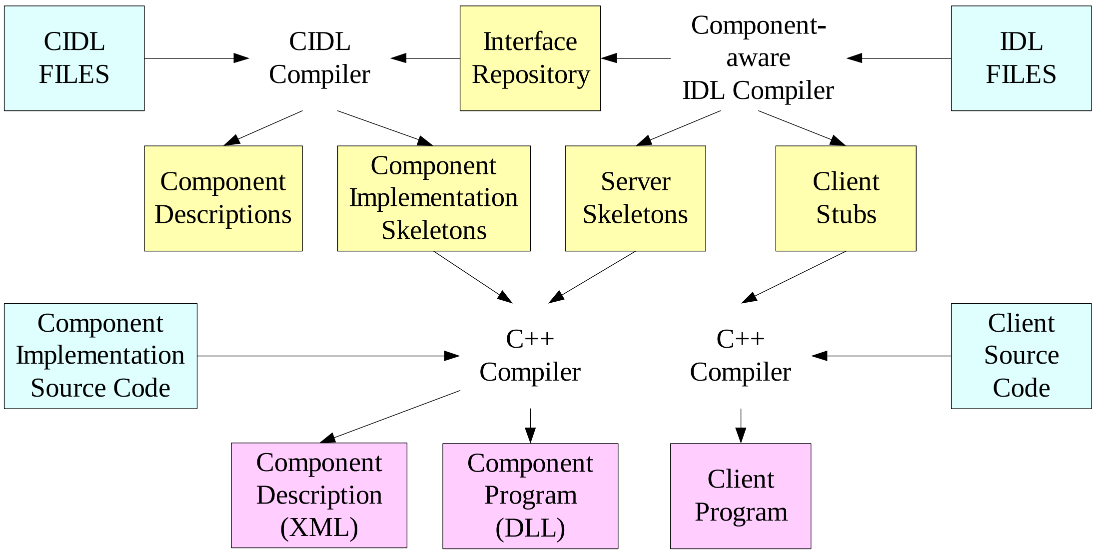
- Component implementations need to support introspection, navigation, & manage connections
- Different implementations may have different run-time requirements
- Different run-time requirements use different container interfaces

Difficulties with Implementing CORBA 2.x Objects



- Problems
  - Generic lifecycle & initialization server code must be handwritten, e.g.:
    - Server initialization & event loop code
    - Any 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 and manage object lifecycles
- Consequences are *ad hoc* design, code bloat, limited reuse

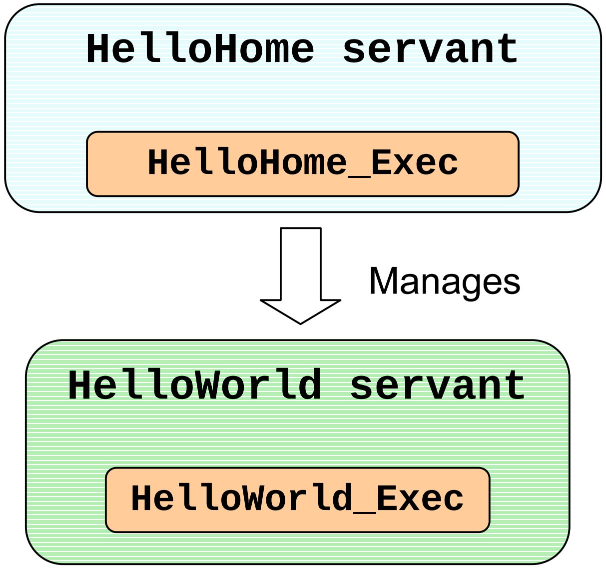
Component Implementation Framework (CIF)



- Defines programming model rules & tools for developing component implementations
  – i.e., specifies how components should be implemented via *executors*
- Facilitates component implementation
  – "only" business logic should be implemented, *not* activation, identification, port management, introspection, etc
- Automates much of the component implementation "glue"

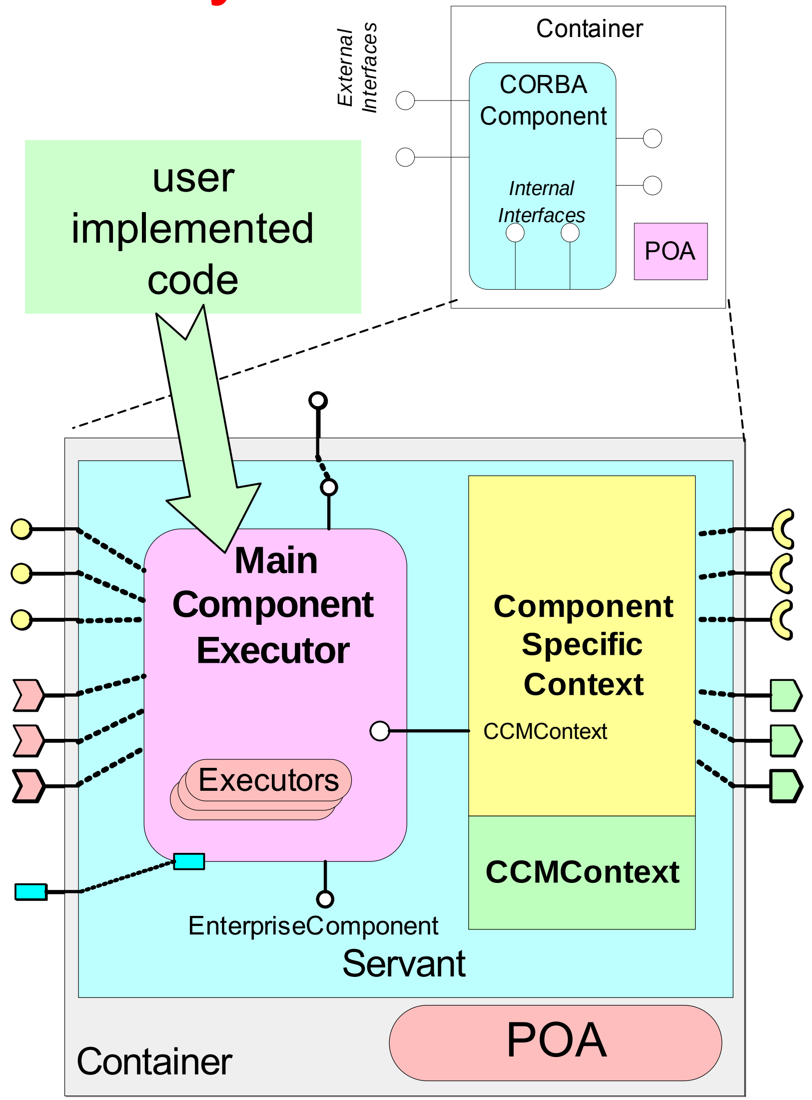
CCM Executors & Home Executors

- Programming artifacts implementing a component's or component home's behavior
  - Local CORBA objects with interfaces defined by the local server-side OMG IDL mapping
- Component executors can be *monolithic*
  - All component attributes, supported interfaces, facet operations, and event sinks implemented by one class
- Component executors can also be *segmented*
  - Component features split into several classes
  - Implements **`ExecutorLocator`** interface
- Home executors are always monolithic

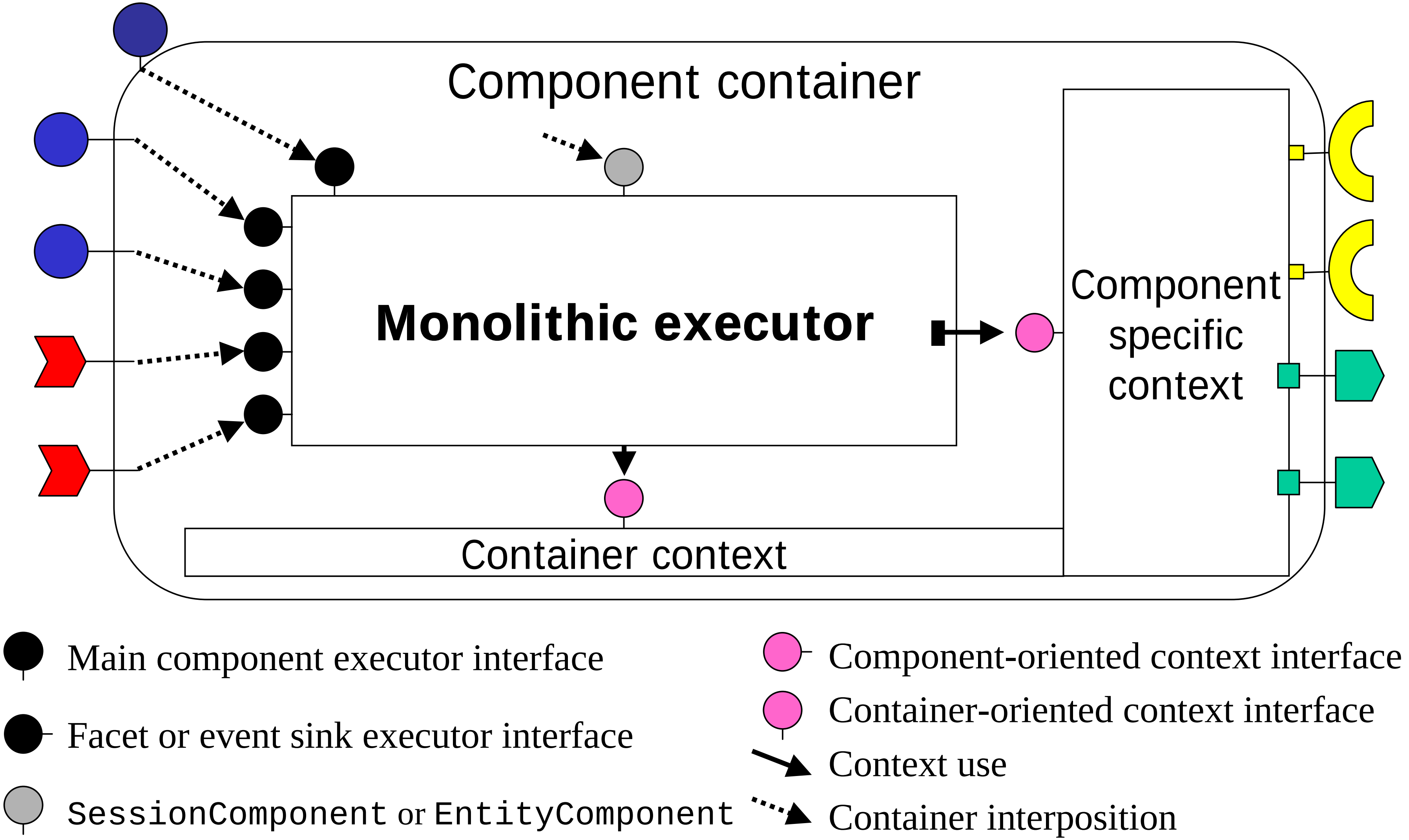


Executors Are Hosted by Container

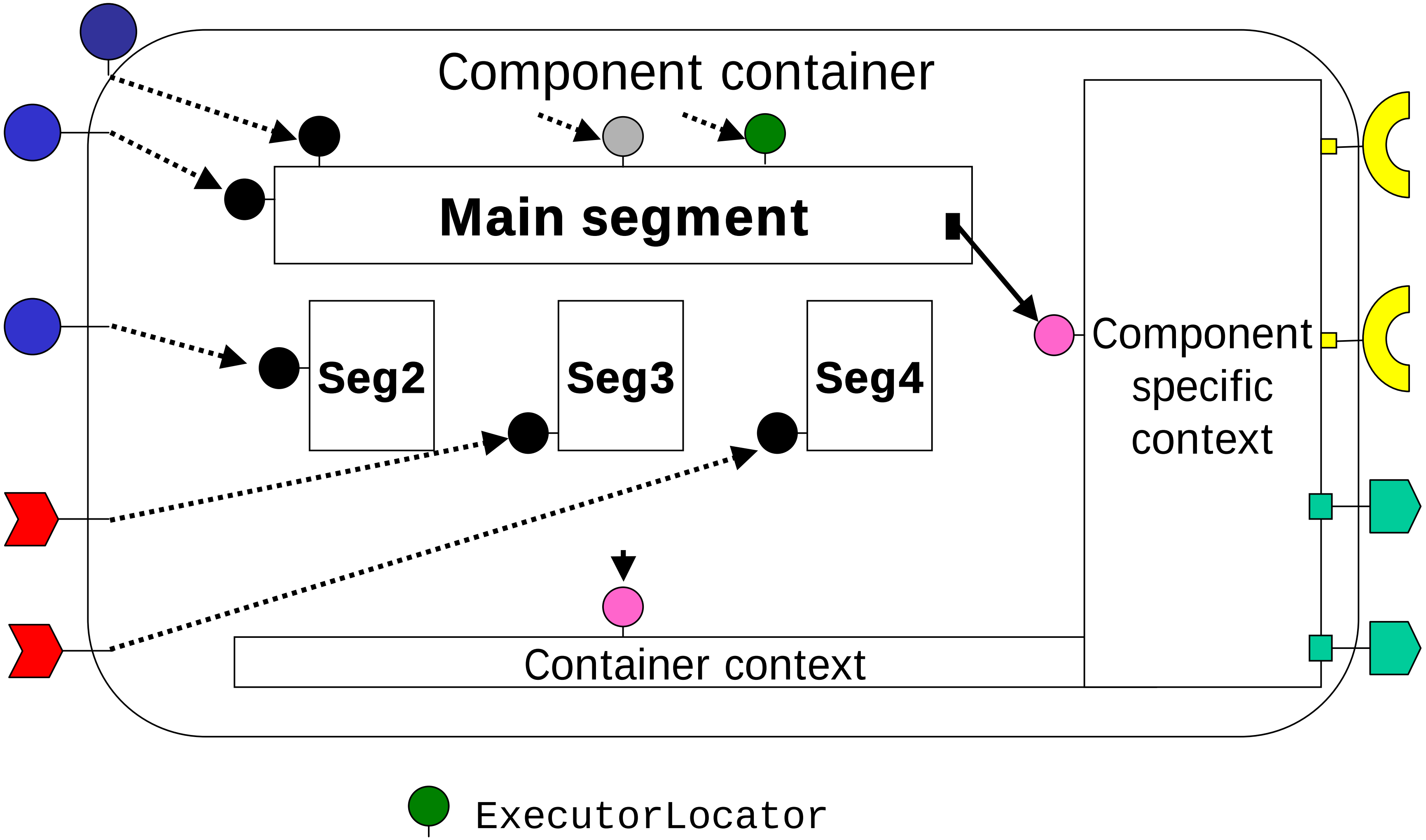
- Container intercepts invocations on executors for managing activation, security, transactions, persistency, etc.
- Component executors must implement a local callback lifecycle interface used by the container
  - `SessionComponent` for transient components
  - `EntityComponent` for persistent components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



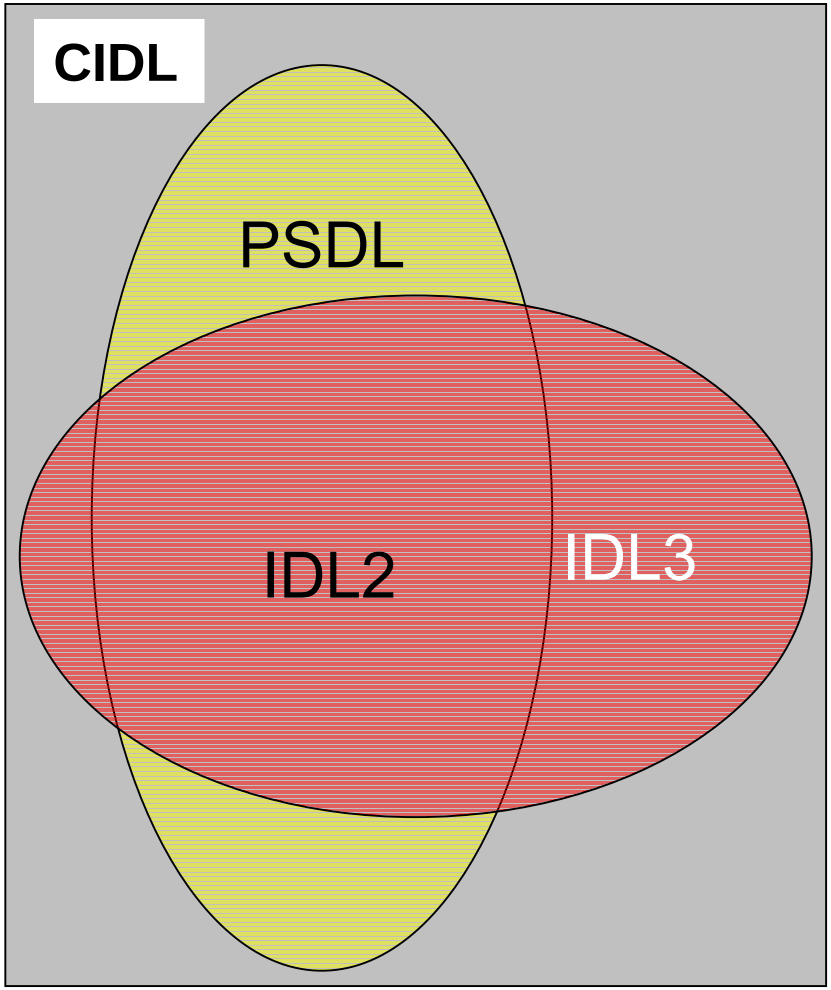
A Monolithic Component Executor



A Segmented Component Executor

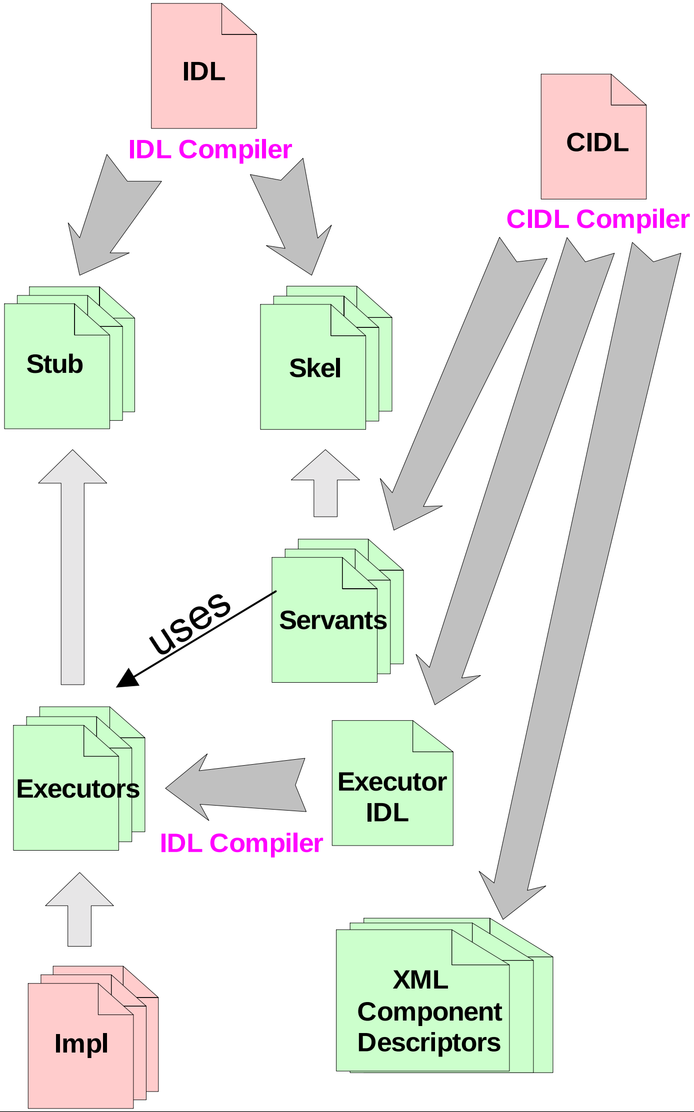


Overview of Component Implementation Definition Language (CIDL)



- Describes component composition
  - Aggregate entity that describes all the artifacts required to implement a component & its home
- Manages component persistence state
  - With OMG Persistent State Definition Language (PSDL)
  - Links storage types to segmented executors
- Generates executor skeletons providing
  - Segmentation of component executors
  - Default implementations of callback operations
  - Component's state persistency

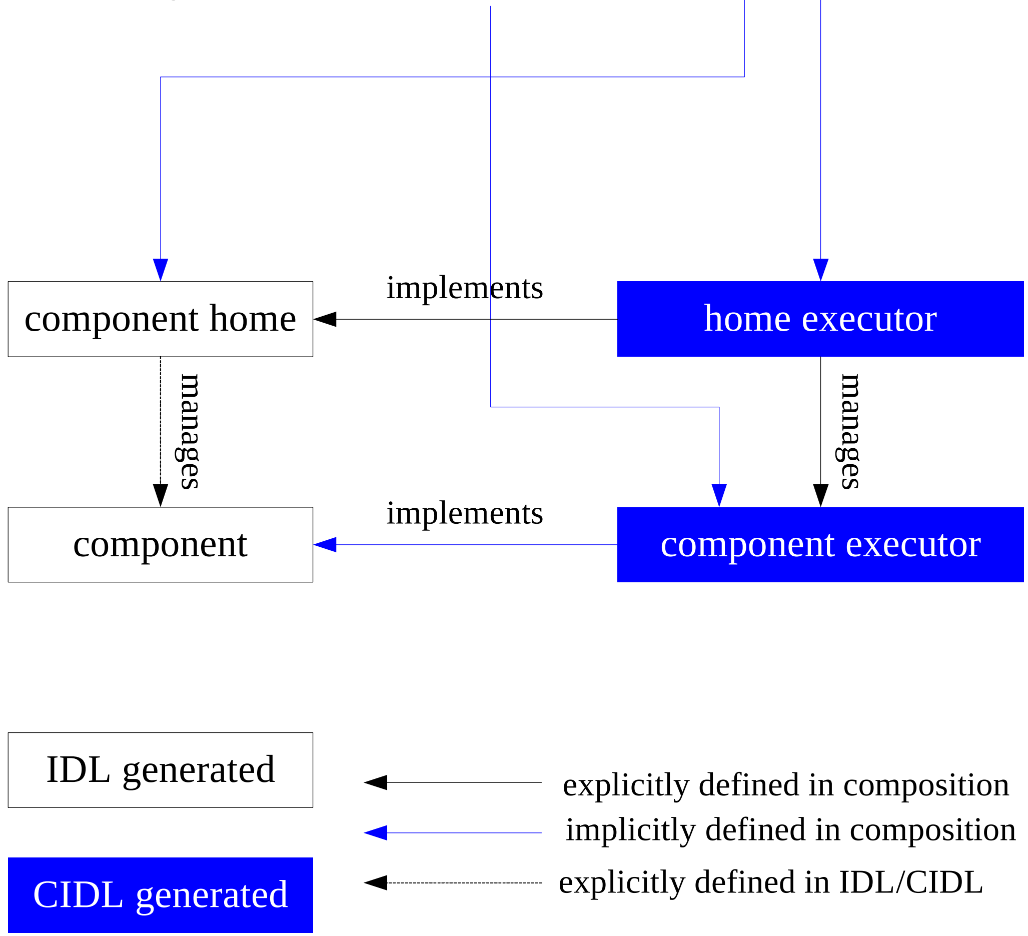
Facilitating Component Implementation via CIDL



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

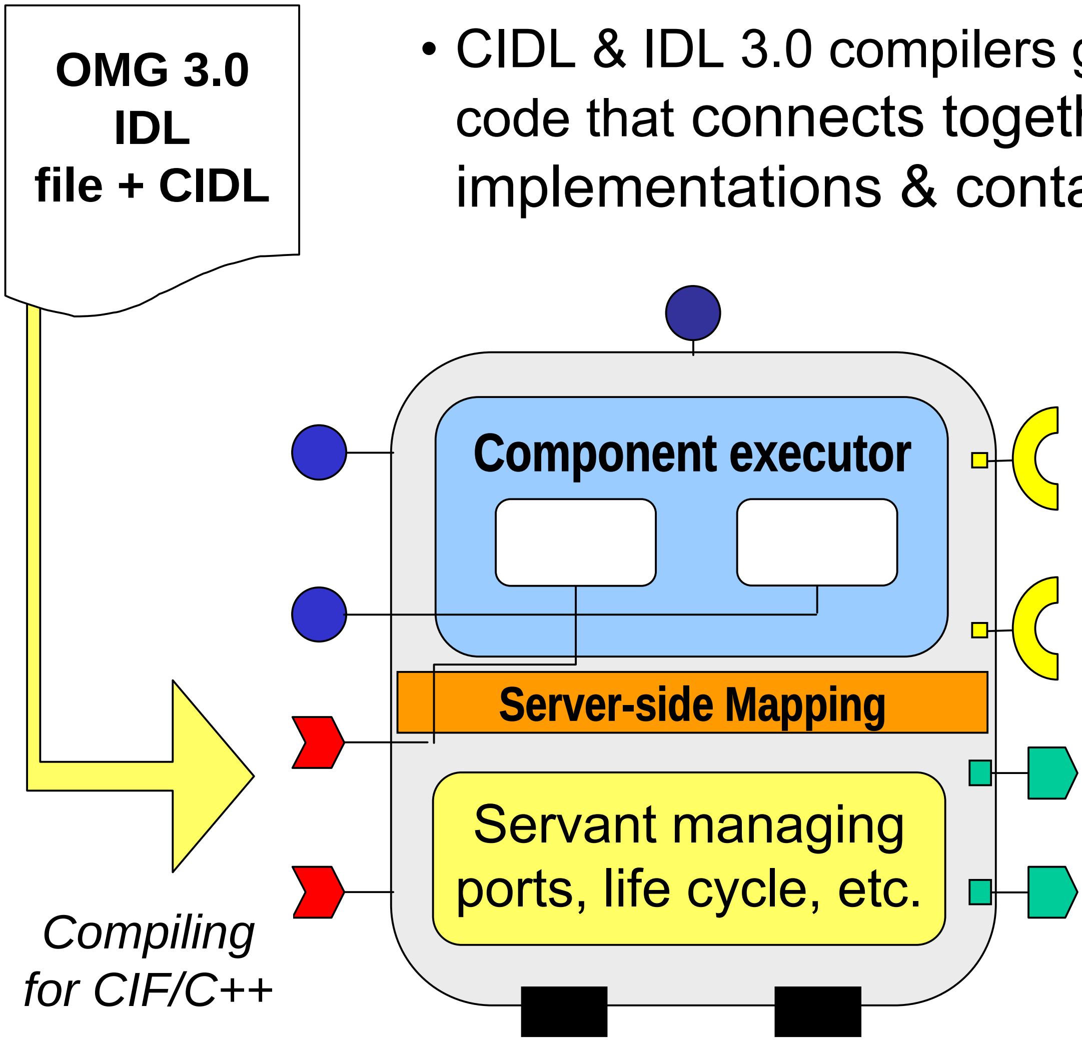
Facilitating Component Composition via CIDL

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



- Component lifecycle category
  - Service, session, process, entity
- Composition name specifies home executor skeleton to generate
- Component home type implemented
  - Implicitly the component type implemented
- Name of main executor skeleton to generate

Connecting Components & Containers with CIDL



- CIDL & IDL 3.0 compilers generate infrastructure "glue" code that connects together component implementations & container that hosts them
- Infrastructure code in container intercepts invocations on executors
  - e.g., can be used to manage activation, security, transactions, persistency, & so on
- CIF defines "executor mappings"

CCM Component Application Examples

Steps for Developing CCM Applications

- **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 and exceptions
- **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
- **Use IDL 3.x features to manage the lifecycle of the component types**, e.g., use the new CCM keyword **home** to define factories that create & destroy component instances
- **Implement your components**, e.g., using C++ or Java & the Component Implementation Definition Language (CIDL), which generates the component implementation executors and associated metadata
- **Assemble your components**, e.g., group related components together & characterize their metadata that describes the components present in the assembly
- **Deploy your components and run your application**, e.g., move the component assemblies to the appropriate nodes in the distributed system & invoke operations on components to perform the application logic

Summary of Server OMG IDL Mapping Rules

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

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

Simple **HelloWorld** Component & Executors

```
// hello.idl

interface Hello {
  void sayHello (in string name);
};
component HelloWorld supports Hello
{};
home HelloHome manages HelloWorld
{};
```

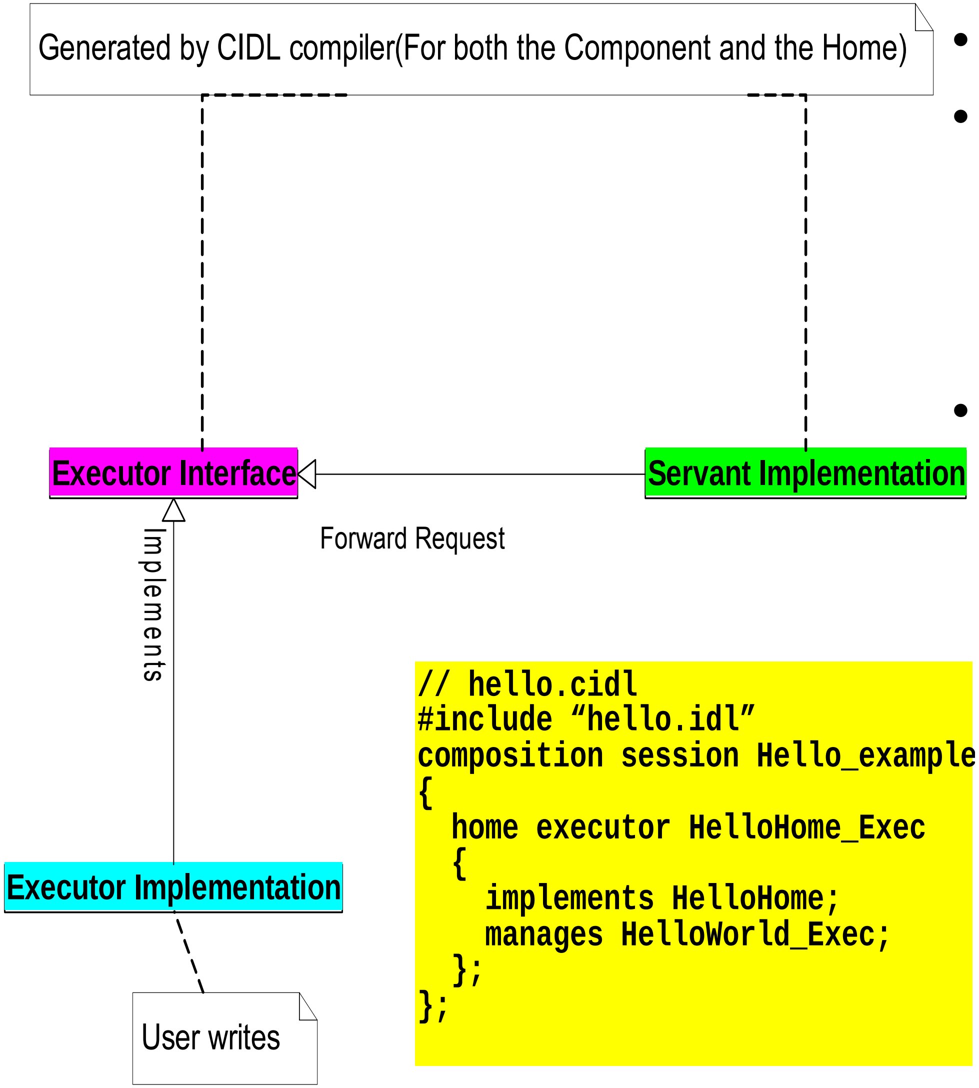
```
class HelloWorld_Impl
  : public virtual CCM_HelloWorld,
    public virtual CORBA::LocalObject {
public:
  HelloWorld_Impl () {}
  ~HelloWorld_Impl () {}
  void sayHello (const char *name) {
    cout << "Hello World! -- from "
         << name << endl;
  }
  // … _add_ref() and _remove_ref()
};
```

```
class HelloHome_Impl
  : public virtual CCM_HelloHome,
    public virtual CORBA::LocalObject
{
public:
  HelloWorldHome_Impl () {}
  ~HelloWorldHome_Impl () {}

  Components::EnterpriseComponent_ptr
  create ()
  {
    return new HelloWorld_Impl ();
  }
  // … _add_ref() and _remove_ref()
};
```

- **`HelloWorld_Impl`** executor implements behavior of **`HelloWorld`** component
- **`HelloHome_Impl`** executor implements lifecycle management strategy of **`HelloWorld`** component

$CIAO_ROOT/docs/tutorial/Hello/

Simple **Hello** CIDL File (1/2)

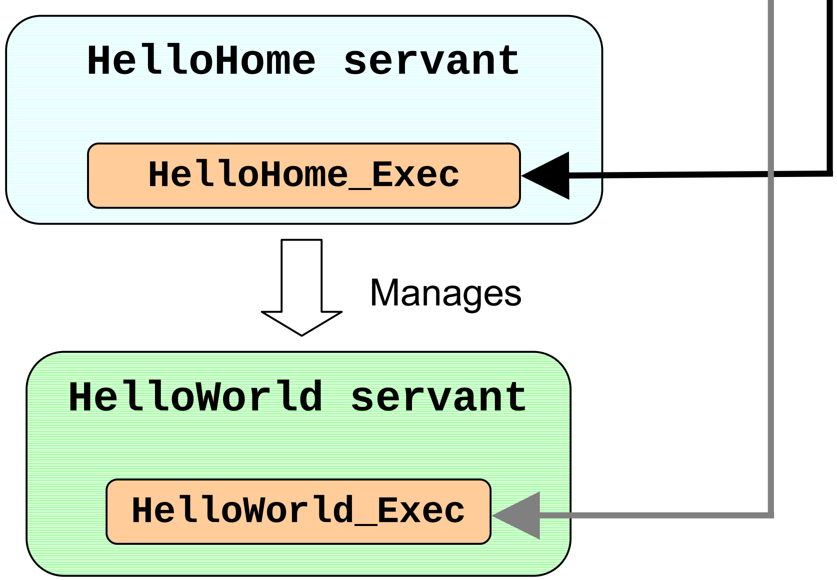



```
// hello.cidl
#include “hello.idl”
composition session Hello_example
{
  home executor HelloHome_Exec
  {
    implements HelloHome;
    manages HelloWorld_Exec;
  };
};
```

- Passed to CIDL compiler
- Triggers code generation for
  - **HelloHome**
  - managed component (**HelloWorld**)
- CIDL compiler generates
  - **HelloHome** servant
  - **HelloWorld** servant
    - attribute operations
    - port operations
    - supported interface operations
    - container callback operations
    - navigation operation overrides
  - IDL executor mapping
  - XML descriptor with meta-data

Simple **`Hello`** CIDL File (2/2)

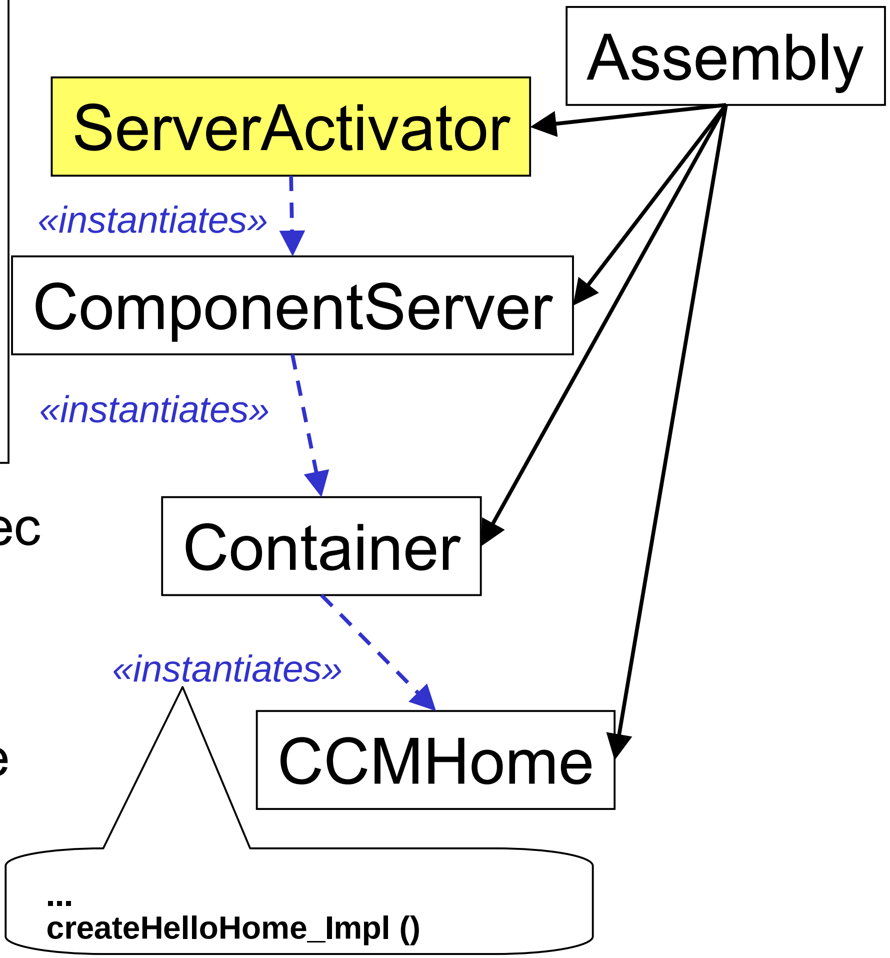
```
// hello.cidl
#include “hello.idl”
composition session Hello_example
{
    home executor HelloHome_Exec
    {
        implements HelloHome;
        manages HelloWorld_Exec;
    }
};
```



- An executor is where a component/home is implemented
  - The component/home's servant will forward a client's business logic request on the component to its corresponding executor
- User should write the implementation of the following **`*_Exec`** interfaces generated by the CIDL compiler:
  - **`HelloHome_Exec`**
  - **`HelloWorld_Exec`**
- In the example, we call these classes
  - **`HelloHome_Impl`**
  - **`HelloWorld_Impl`**

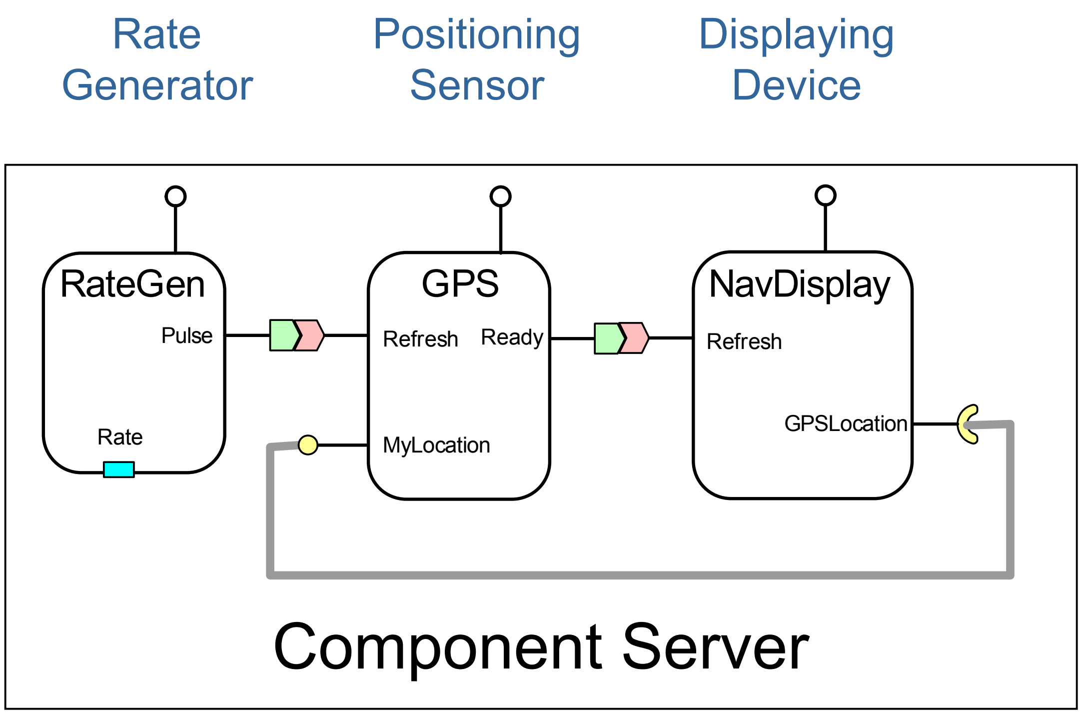
Component Entry Point Example

```
extern "C" {

Components::HomeExecutorBase_ptr
createHelloHome_Impl (void)
{
  return new HelloHome_Impl;
}

}
```



- The signature is defined by the CCM spec
- Container calls this method to create a home executor
- extern “C” required to prevent C++ name mangling, so function name can be resolved in DLL
- User or modeling tool should generate the XML file that contains this information

Implementing Port Mechanisms

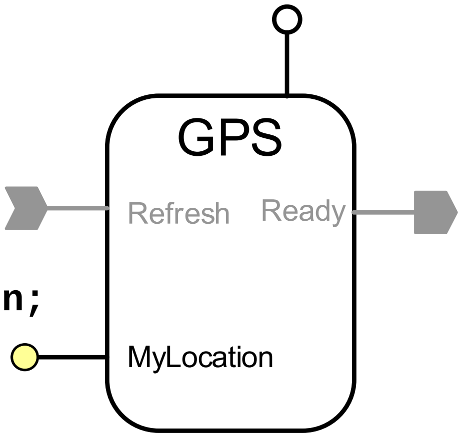


- Component developers must implement
  - Entities that are invoked by its clients
    - Facets
    - Event sinks
  - Entitles that the component invokes
    - Receptacles
    - Event sources

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

Implementing Facets

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



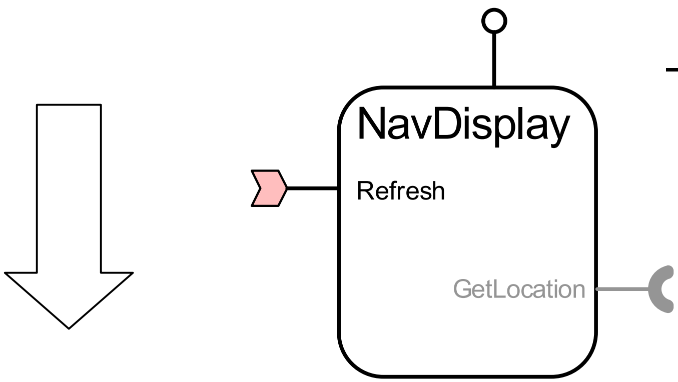

```
// Equivalent IDL 2
interface GPS :
  Components::CCMObject
{
  position
    provide_MyLocation ();
  …
};
```

```
local interface GPS_Executor :
  CCM_GPS,
  CCM_position,
  Components::SessionComponent
{
  // Monolithic Executor Mapping
}
```

```
class GPS_Executor_Impl :
  public virtual GPS_Executor,
  public virtual CORBA::LocalObject
{
public:
  …
  virtual CCM_position_ptr
  get_MyLocation ()
  { return this; }

  virtual CORBA::Long
  get_pos ()
  { return cached_current_location_; }
  …
};
```

Component Event Sinks

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

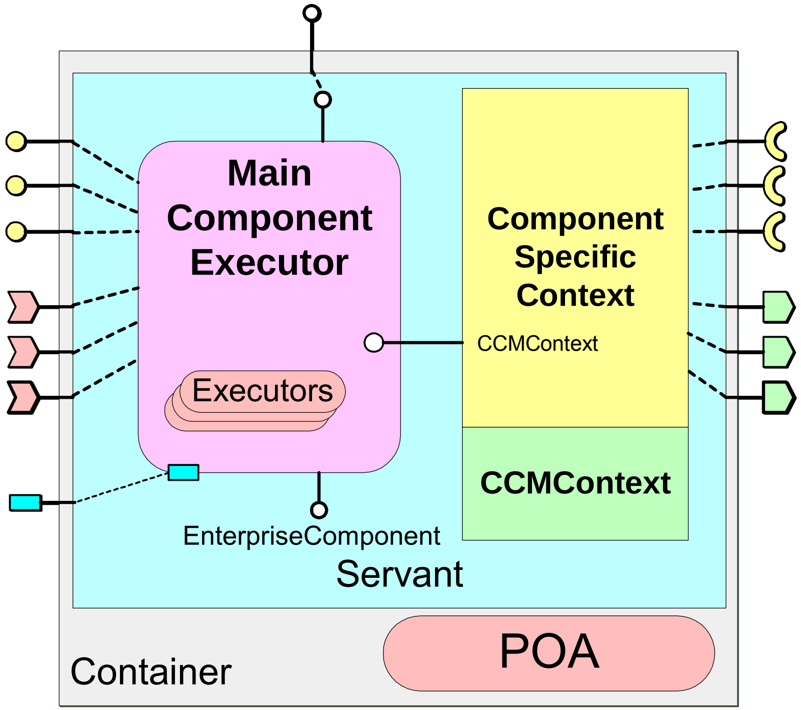



```
// Equivalent IDL 2
interface NavDisplay :
  Components::CCMObject
{
  …
  tickConsumer get_consumer_Refresh();
  …
};
```

- Event sinks
  - Clients can acquire consumer interfaces, similar to facets
  - CIDL generates event consumer servants
  - Executor mapping defines push operations directly

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

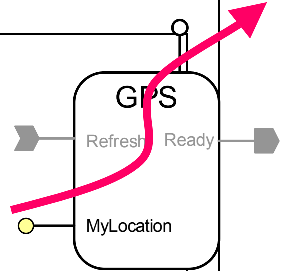
Initializing Component-specific Context



- Component-specific context manages connections & subscriptions
- Container passes component its context via either
  - `set_session_context()`
  - `set_entity_context()`

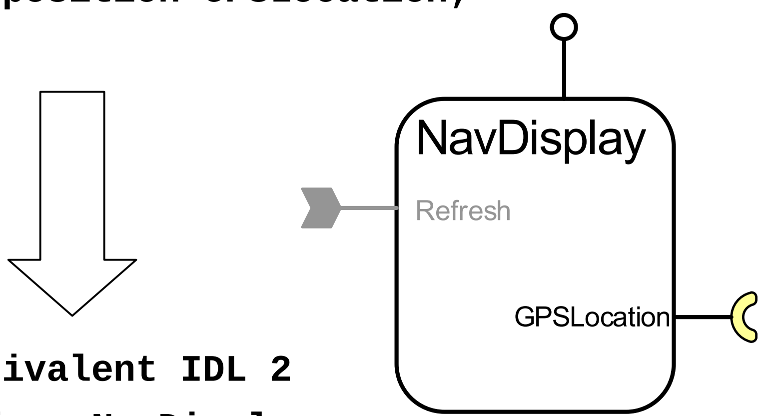
```
class GPS_Executor_Impl :
  public virtual GPS_Executor,
  public virtual CORBA::LocalObject
{
private:
  CCM_GPS_Context_var context_;
public:
  …
  void set_session_context
    (Components::SessionContext_ptr c)
  {
    this->context_ =
      CCM_GPS_Context::narrow (c);
  }
  …
};
```

```
local interface GPS_Executor :
  CCM_GPS,
  CCM_position,
  Components::SessionComponent
{
}
```



Using 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 ();
 …
};
```

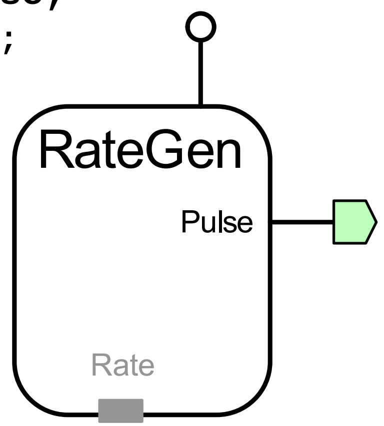
- Component-specific context manages receptacle connections
- Executor acquires the connected reference from the context

```
class NavDisplay_Executor_Impl :
  public virtual CCM_NavDisplay,
  public virtual CORBA::LocalObject
{
public:
  …
  virtual void refresh_reading (void)
  {
    position_var cur =
      this->context_->
        get_connection_GPSLocation ();
    long coord = cur->get_pos ();
    …
  }
  …
};
```

Pushing Events from a 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);
  …
};
```

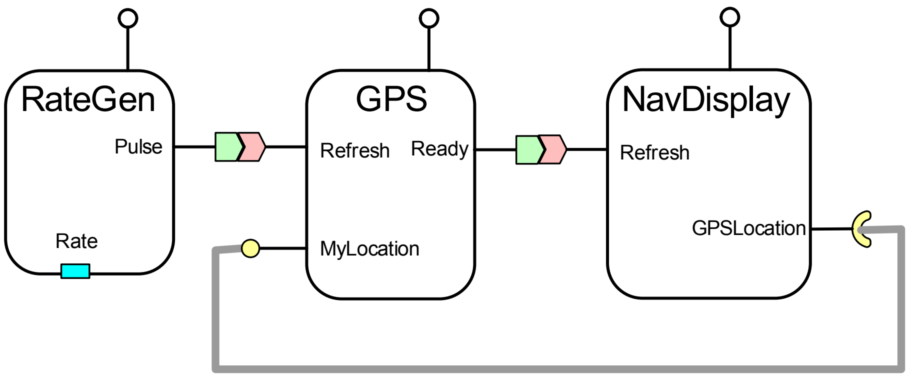


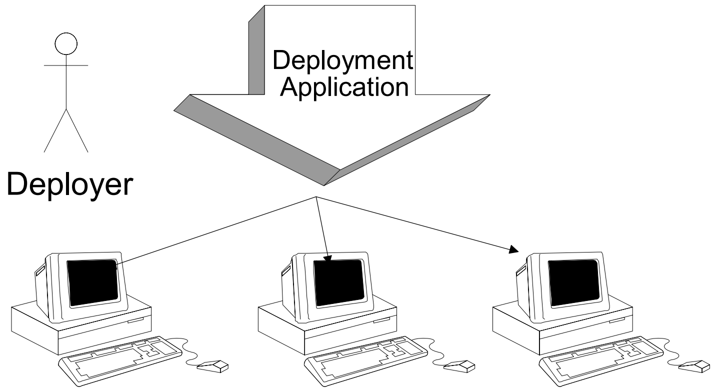
- Component-specific context manages consumer subscriptions (for publishers) & connections (for emitters)
- Component-specific context also provides the event pushing operations & relays events to consumers

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

Component Deployment & Configuration

Motivation for Deployment & Configuration



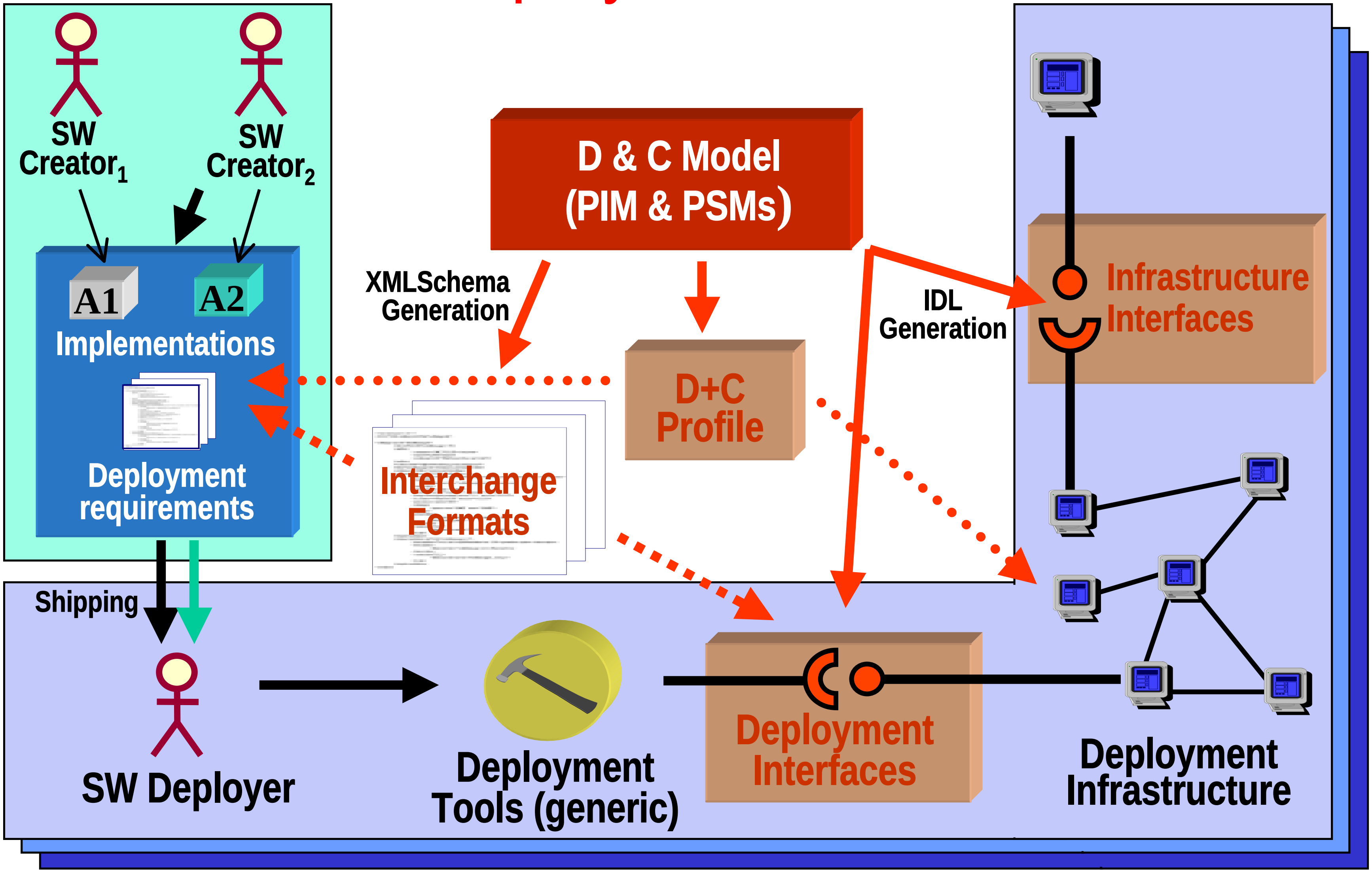


- Goals
  - Ease component reuse
  - Build complex applications by assembling existing components
  - Deploy component-based application into heterogeneous domain
- Separation of concerns
  - Component development
  - Application assembly
  - Application deployment
  - Application configuration
  - Server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

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 (e.g., on serial bus or USB)
    - e.g., Navigation display component needs … a display
      - not as trivial as it may sound!
- Computers, networks are heterogeneous
  - Not all computers can execute all component implementations
- The above is true for each and every component of an application
  - Each component has different requirements

Deployment Idea

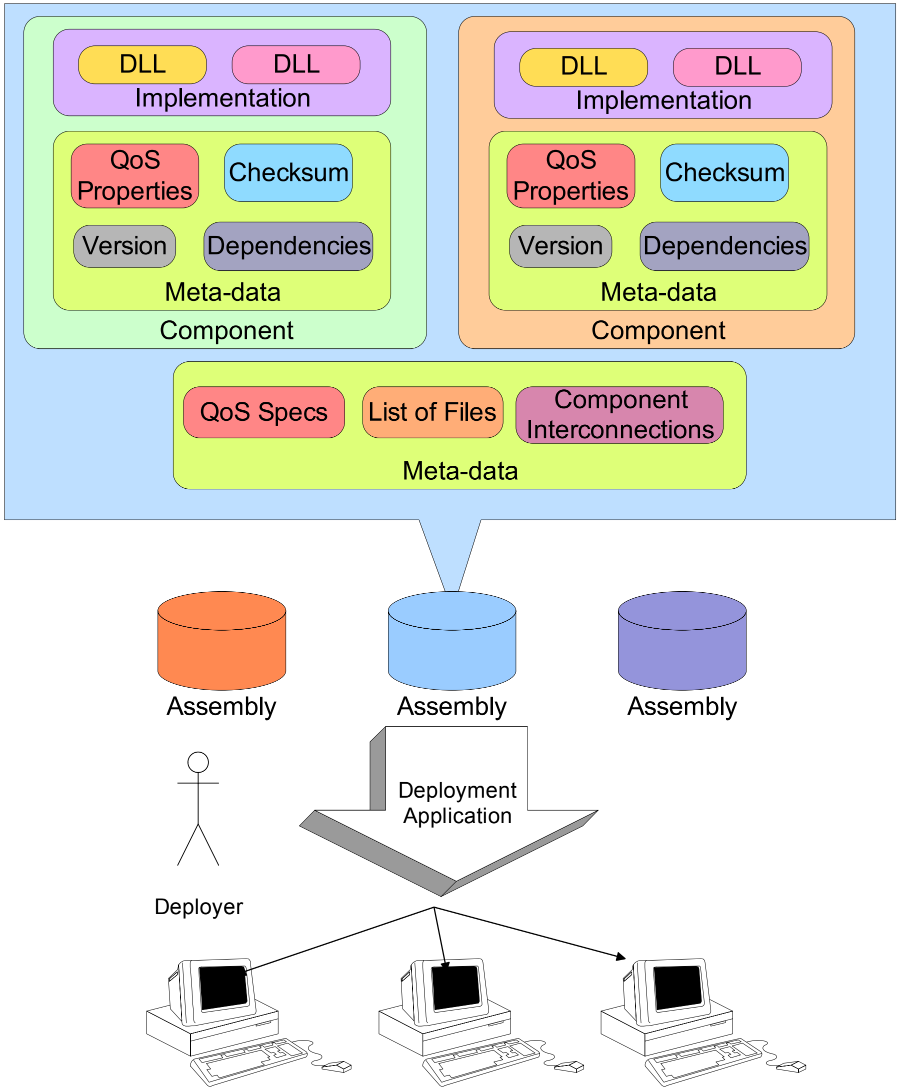


Deployment Solution

- Well-defined Exchange Format
  - Defines what a software vendor delivers
  - Requires “off-line” data format, to be stored in files
- Well-defined Interfaces
  - Infrastructure to install, configure and deploy software
  - Requires “on-line” data format, to be passed to and from interfaces
- Well-defined software meta-data Model
  - Annotate software and hardware with interoperable, vendor-independent, deployment-relevant information
  - Generate “on-line” and “off-line” data formats from model

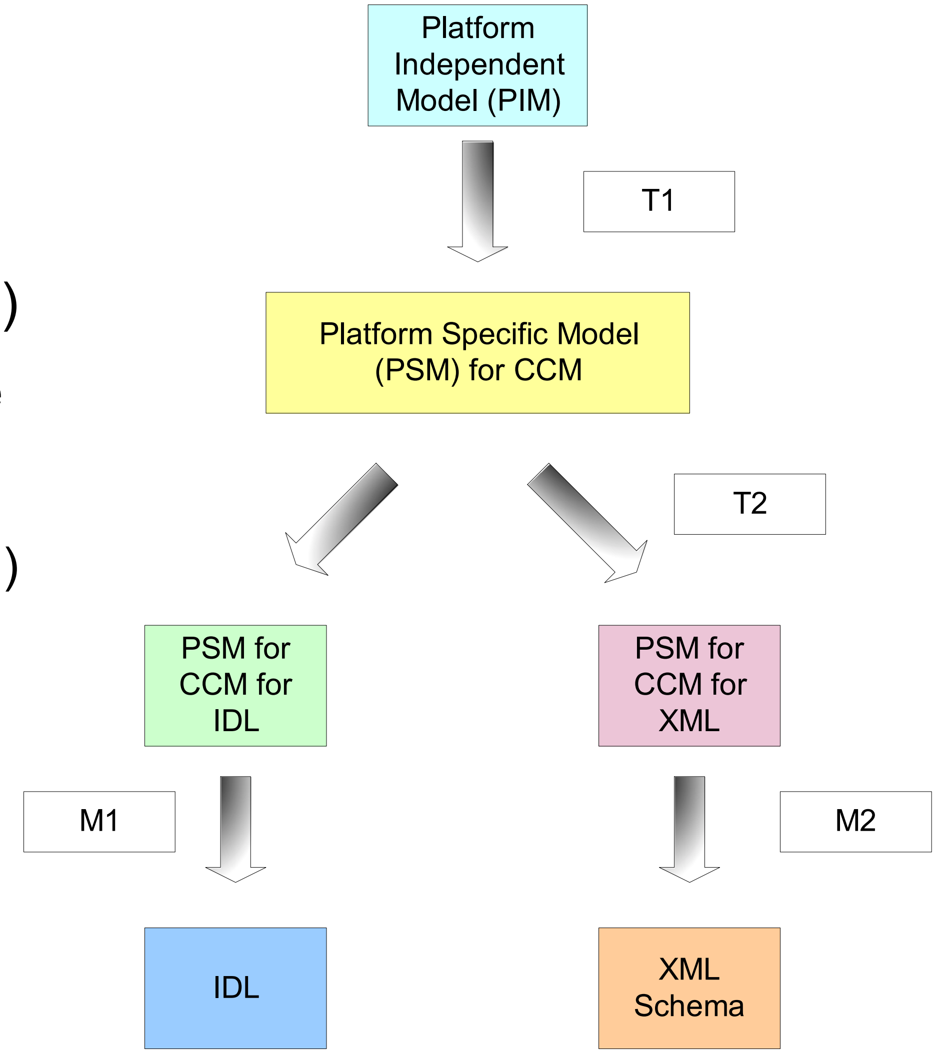
CCM Deployment & Configuration (D+C) Spec

- "D+C" spec was adopted by OMG last year
- Intended to replace *Packaging & Deployment* chapter of CCM specification
- Supports …
  - Hierarchical assemblies
  - Resource management
  - QoS characteristics
  - Automated deployment
  - Vendor-independent deployment infrastructure



D+C & the MDA Approach

- Platform-independent model
  - Defines "deployment" model
  - Independent of CORBA and CCM
- Refined into CCM-specific model (T1)
- Uses standard mappings to generate
  - IDL (for "on-line" data)
    - using UML Profile for CORBA (M1)
  - XML Schema (for "off-line" data)
    - using XMI (M2)
- Intermediate transformation T2
  - Transforms PSM for CCM into suitable input for M1 and M2

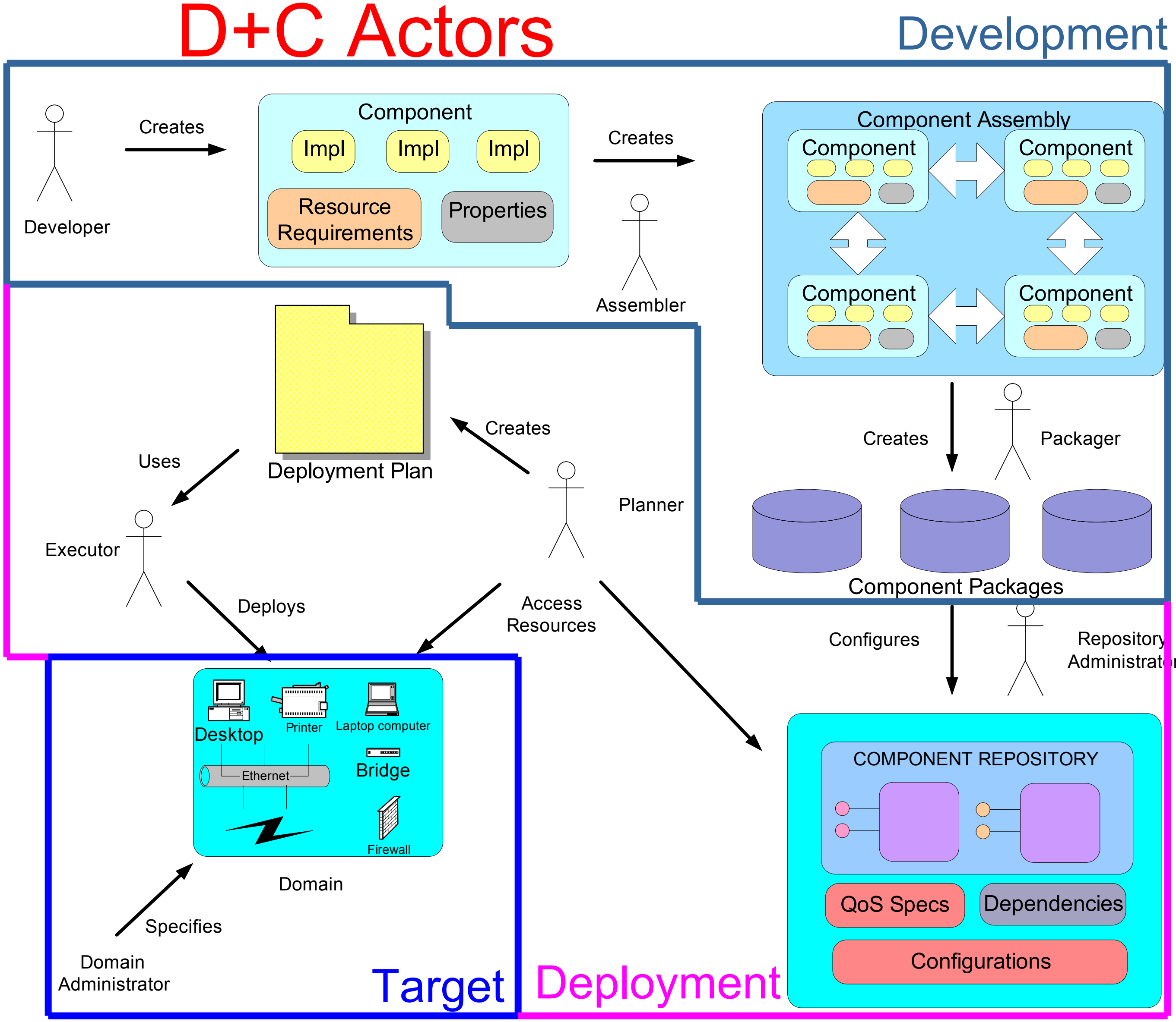


D+C Metadata Model Slices

- **Component Model**
  - Metadata to describe component-based applications
  - "Repository Manager" interface for installing, maintaining and retrieving Component Packages
- **Target Model**
  - Metadata to describe available resources
  - "Target Manager" interface for accessing and tracking resources
- **Execution Model**
  - Metadata to describe "Deployment Plan"
  - "Execution Manager" interface to execute applications according to plan

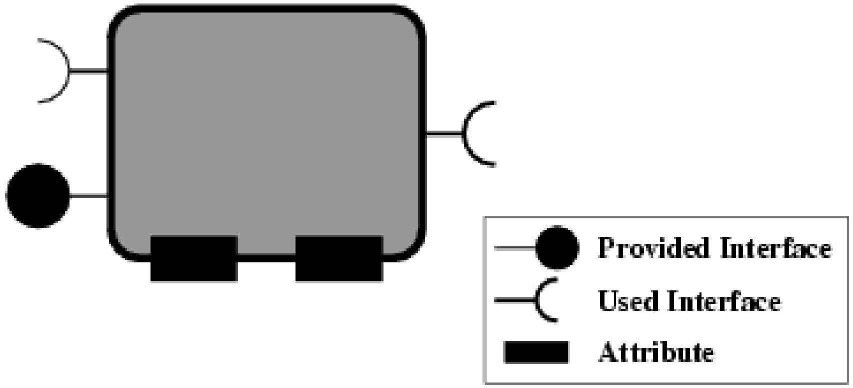
D+C Actors

- Different Stages
  - *Development*
    - Developer
    - Assembler
    - Packager
  - *Target*
    - Domain Administrator
  - *Deployment*
    - Repository Administrator
    - Planner
    - Executor
- Actors are abstract
  - Usually human + software tool



Component-based Software: Component

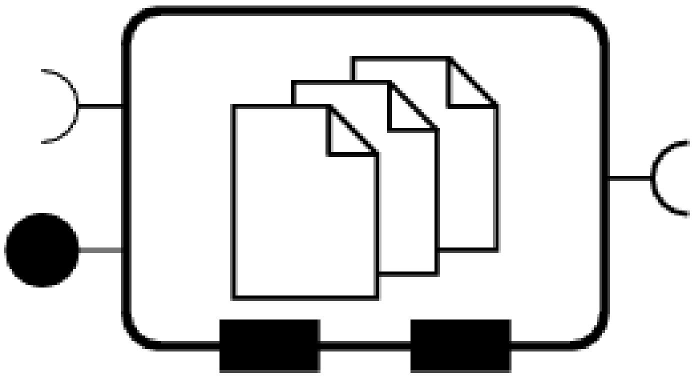
- **Component**
  - Modular
  - Encapsulates its contents
  - Replaceable "black box", conformance defined by interface compatibility



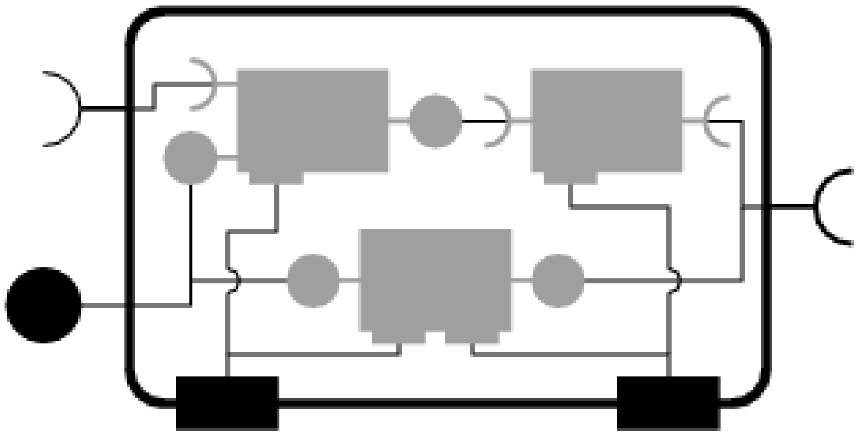
- **Component Interface**
  - "Ports": provided interfaces, used (required) interfaces
  - Attributes
- **Component Implementation**
  - Either "Monolithic" or
  - "Assembly-based"

Monolithic Implementation

- **Monolithic Implementation**
  - Executable piece of software
    - One or more "implementation artifacts" (e.g., .exe, .so, .class)
    - Zero or more supporting artifacts (e.g., config files)
  - May have hardware or software requirements
    - Specific CPU (e.g., x86)
    - Specific OS (e.g., Linux)
    - Hardware devices (e.g., GPS sensor)



Assembly-based Implementation

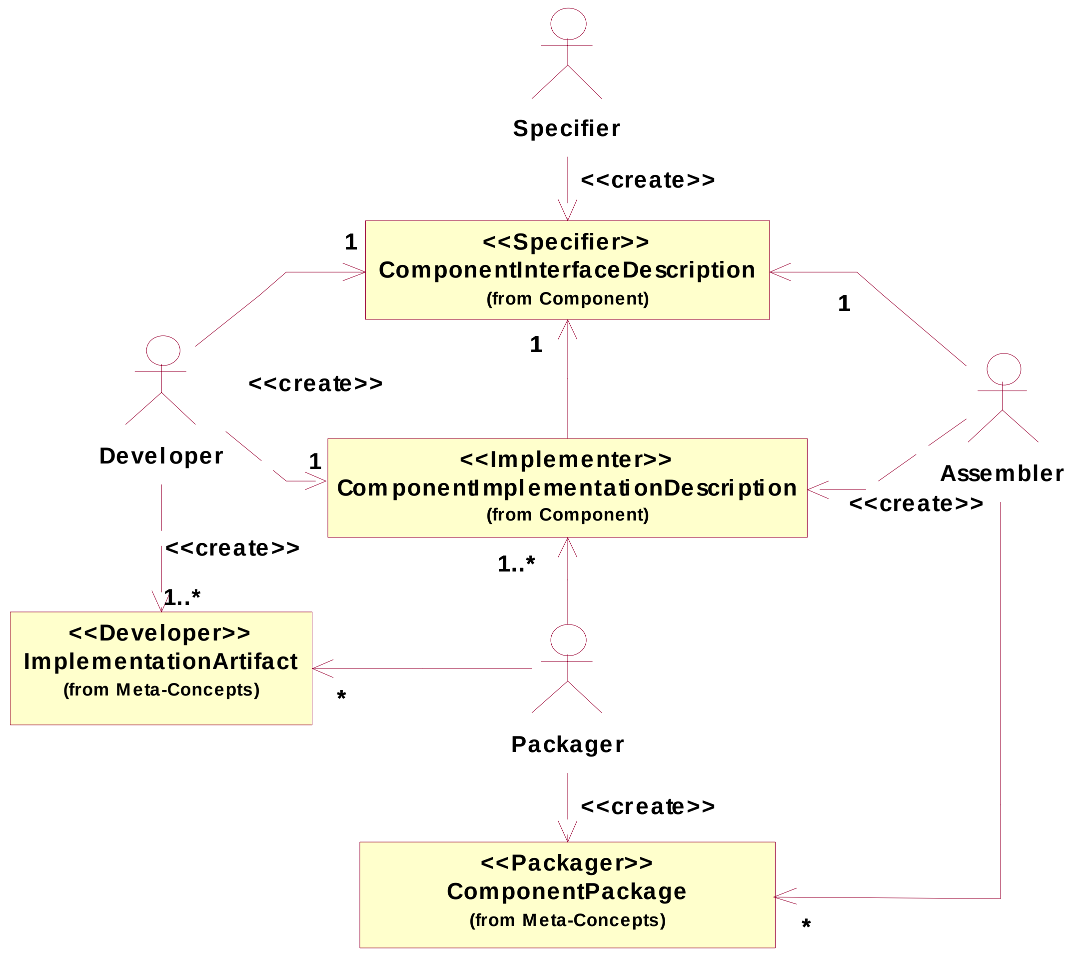


- Set of interconnected subcomponents
- Hardware and software independent
  - Reuses subcomponents as “black boxes”, independent of their implementation
- Implements a specific component interface
  - Ports & attributes are “mapped” to subcomponents
- Assemblies are fully reusable
  - Can be “standalone” applications, or reusable components
  - Can be used in an encompassing assembly

Component Package

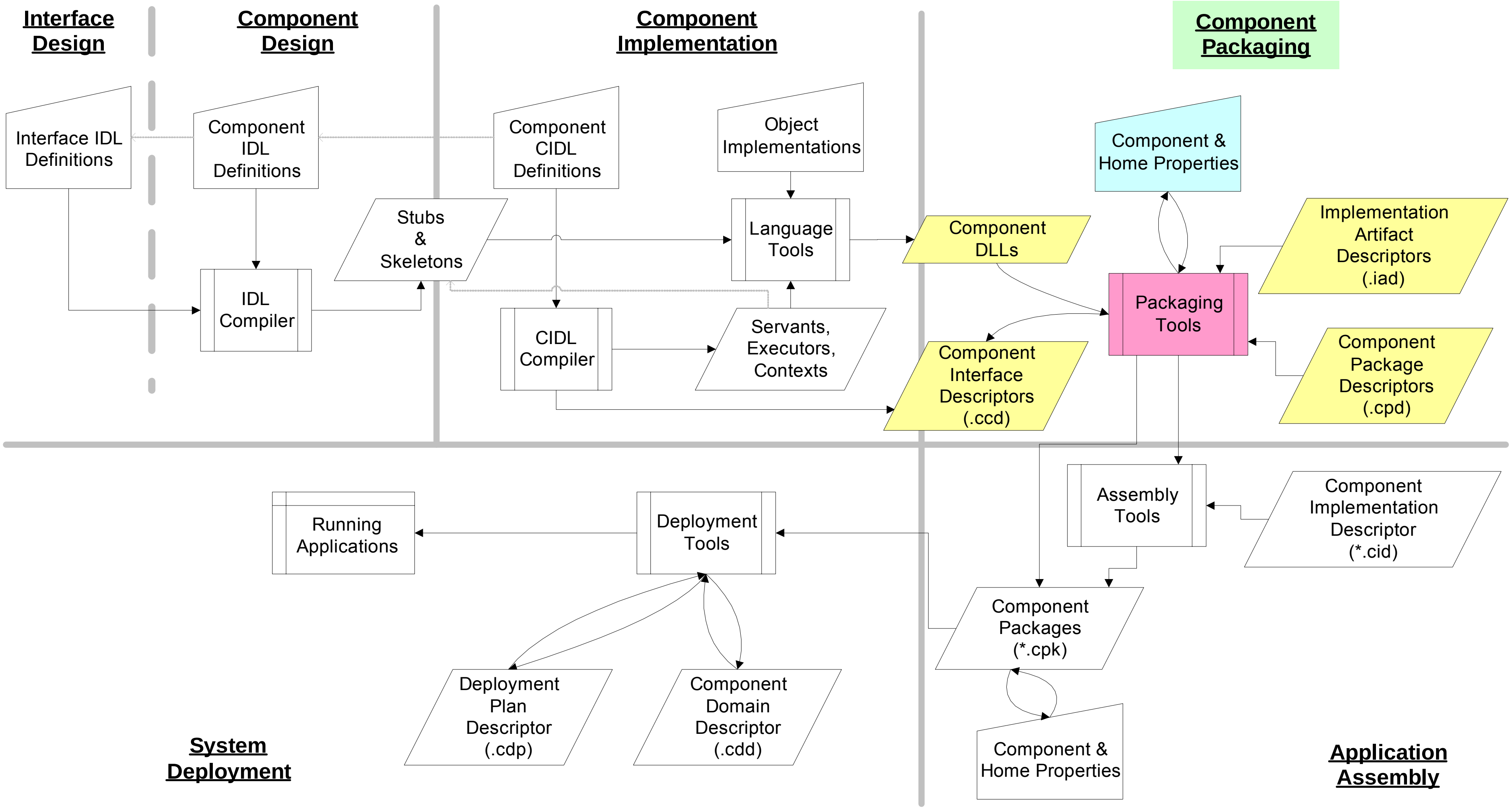
- **Component Package**
  - A set of alternative, replaceable implementations of the same component interface
    - e.g., Implementations for Win32, Linux, Java
  - May be a mix of monolithic and assembly-based implementations
    - e.g., a parallel, scalable implementation for Mercury multicomputer, or a single, monolithic Java component
  - Implementations may have different “Quality of Service”
    - e.g., latency, resolution
  - “Best” implementation is chosen at deployment time
    - Based on available hardware and QoS requirements

Development Actors



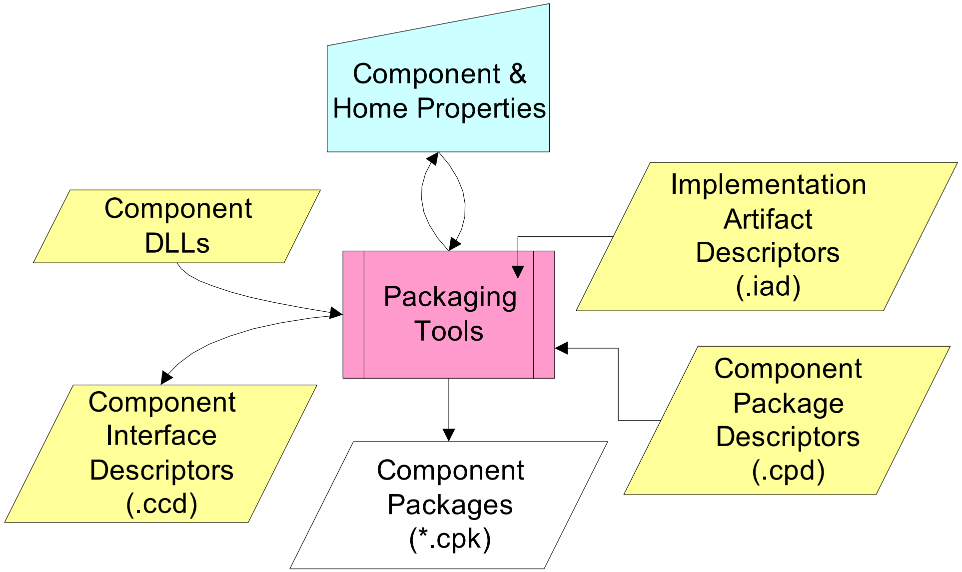
Component Packaging

Goal: Associate a component implementation with its meta-data



Component Packaging Tools

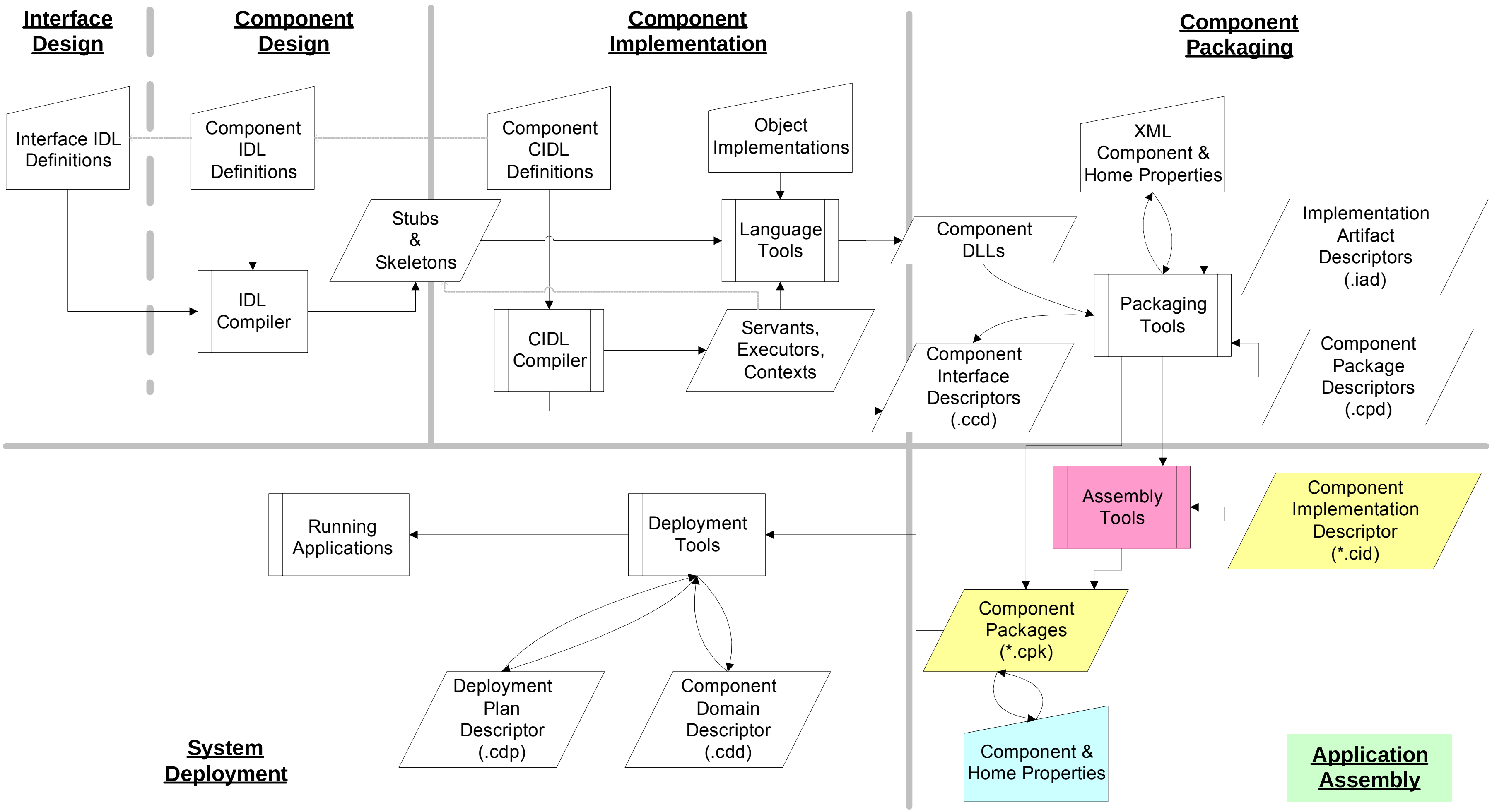
- Goals
  - Extract systemic properties into meta-data
  - Configure components, containers, target environment, applications



- CCM component packages bring together
  - Multiple component implementations
  - Component properties
  - Descriptors (XML Files)
- Descriptors provide meta-data that describe contents of a package, dependencies on other components, 3rd party DLLs, & value factories

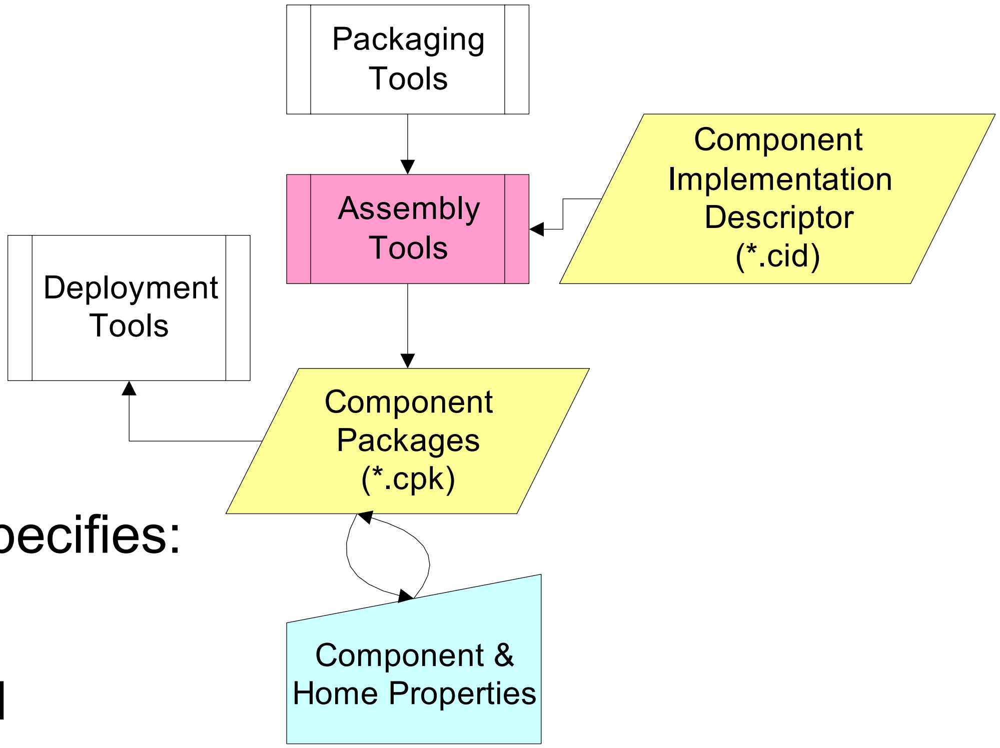
Application Assembly

Goal: Group packages & meta-data by specifying inter-connections

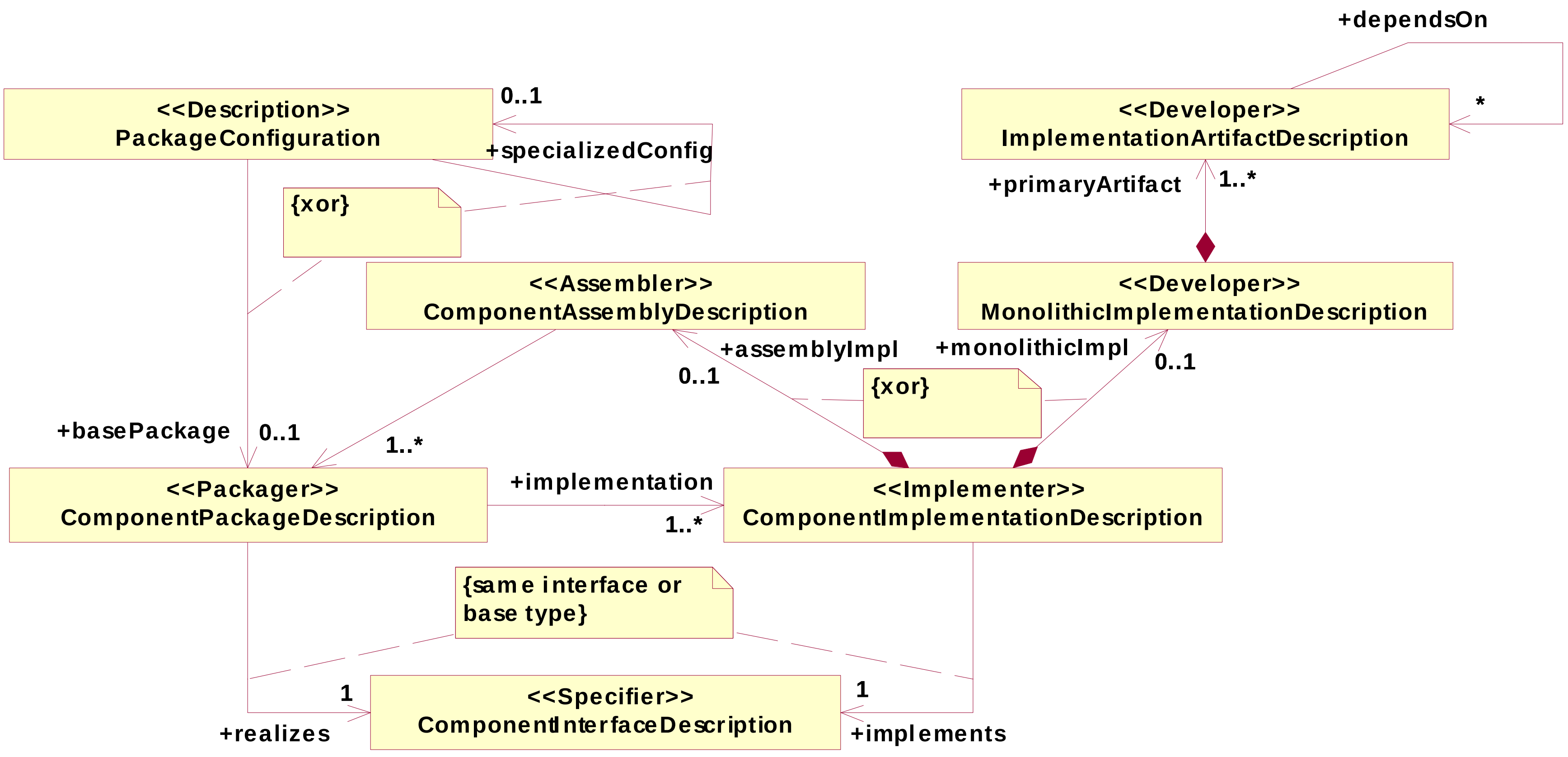


Application Assembly Tools

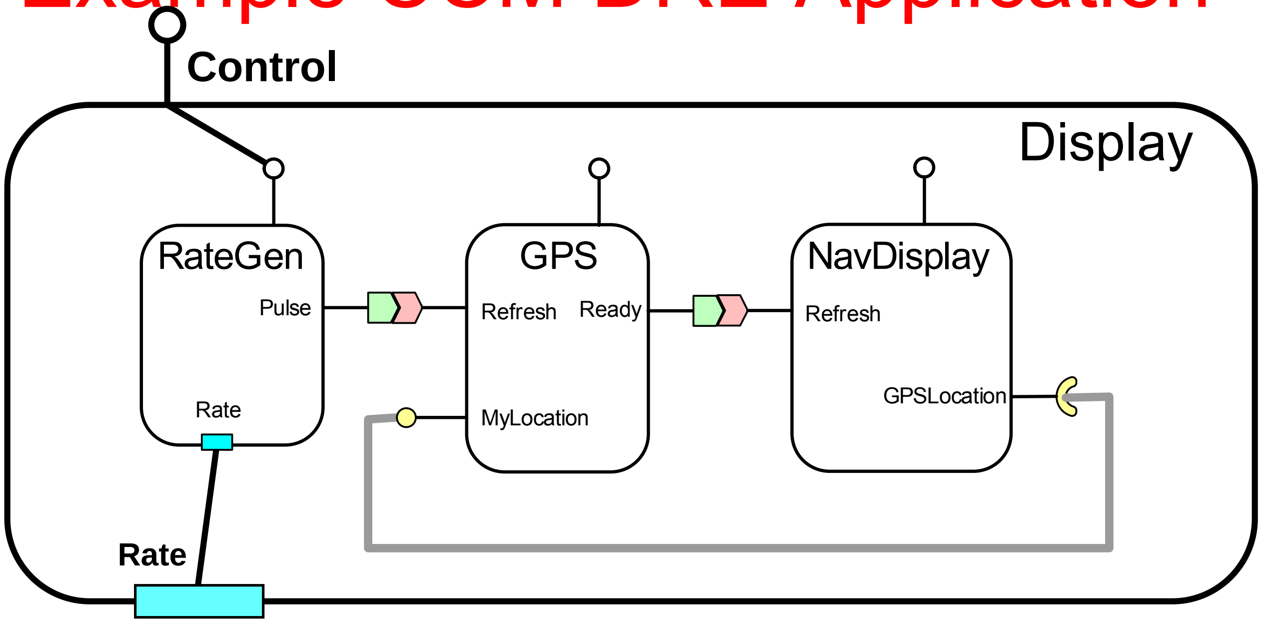
- Goals
  - Compose higher level components from set of sub-components
  - Store composition information as meta-data
  - Provide logical abstraction
- Component assembly description specifies:
  - Sub-component packages
  - Sub-component instantiation and configuration
  - Interconnections
  - Mapping of ports and properties to subcomponents
- "Pure meta-data" construct (no code, hardware-independent)



Component Data Model Overview

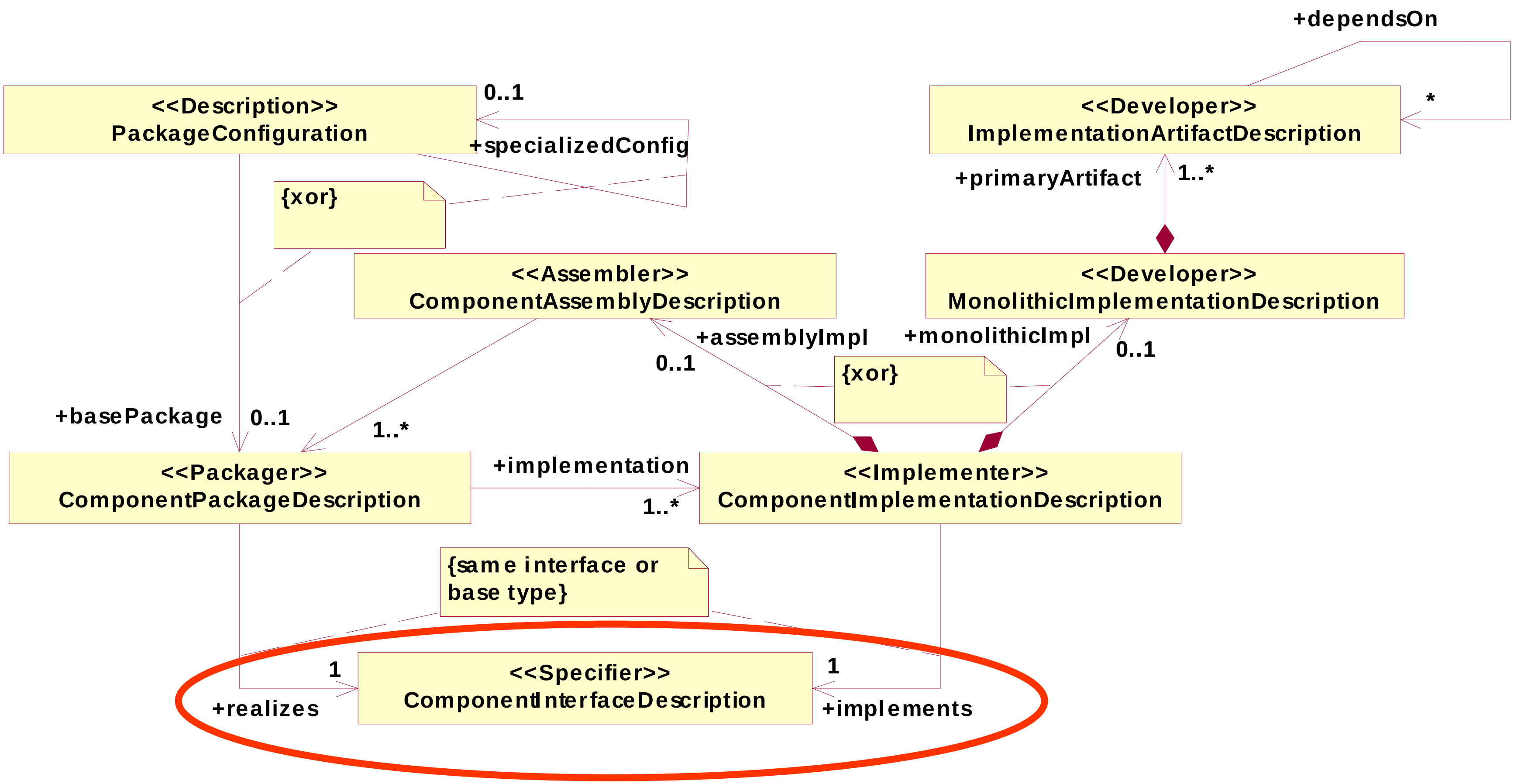


Example CCM DRE Application

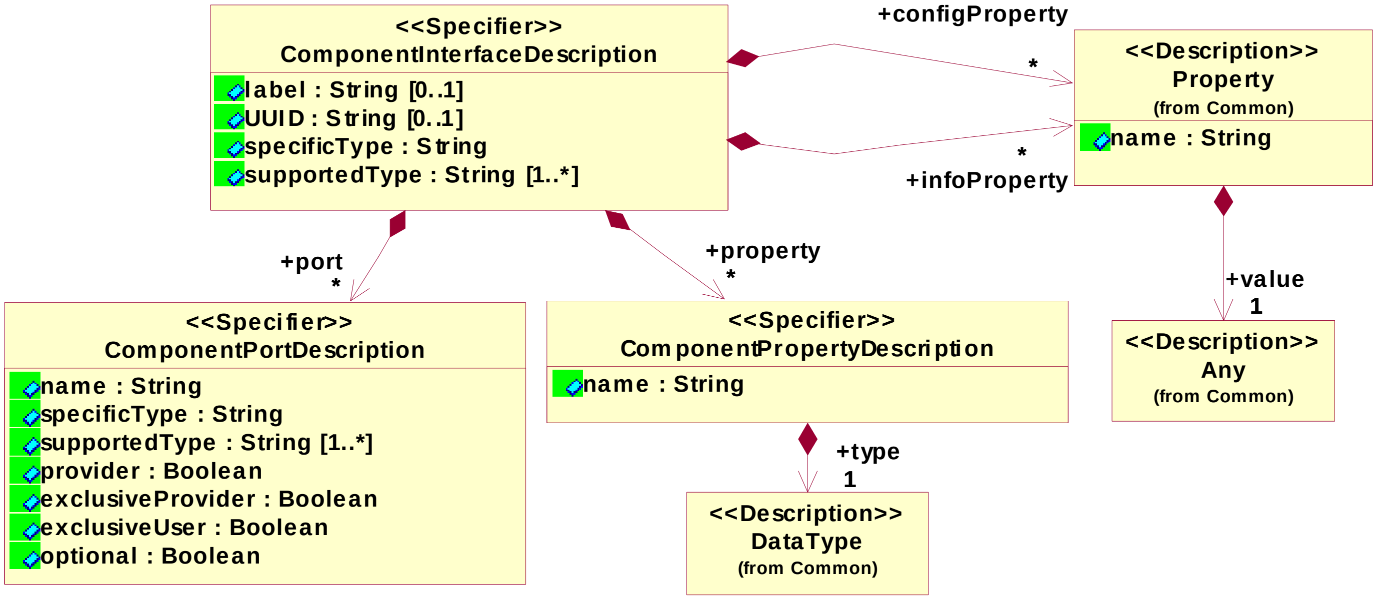


- "Display" component is an assembly of three components
- RateGen, GPS, & NavDisplay implemented monolithically
- GPS component requires "GPS" device
- Two alternative implementations for NavDisplay
  - Text-based & GUI versions

Component Interface Description



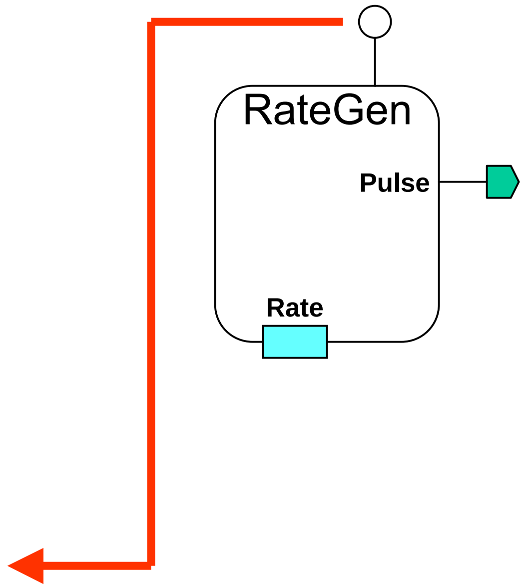
Component Interface Description



- Meta-data to describe a component interface
  - Identifies a component's specific (most-derived) type, and supported (inherited) types
  - Describes a component's ports and properties (attributes)
  - Optionally configures default property values

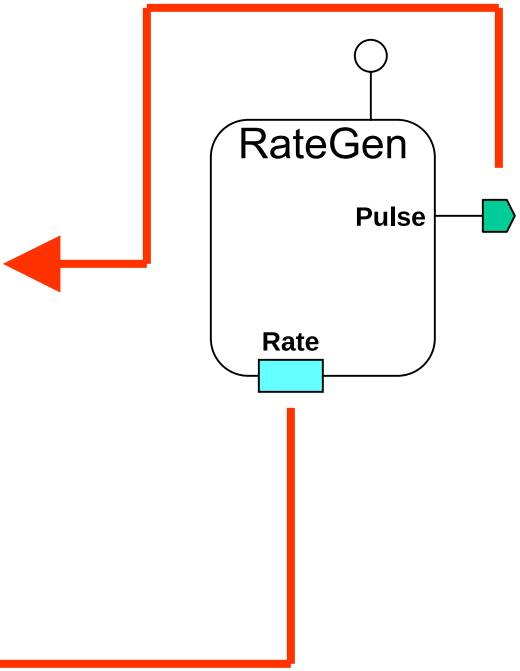
Component Interface Descriptor for RateGen component: RateGen.ccd (1/3)

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Deployment:ComponentInterfaceDescription
    xmlns:Deployment='http://www.omg.org/Deployment'
    xmlns:xmi='http://www.omg.org/XMI'
    >
  <label>Rate Generator</label>
  <specificType>IDL:HUDisplay/RateGen:1.0</specificType>
  <supportedType>IDL:HUDisplay/RateGen:1.0</supportedType>
  <idlFile>RateGen.idl</idlFile>
 <port>
    <name>supports</name>
    <specificType>IDL:HUDisplay/opmode:1.0</specificType>
    <supportedType>IDL:HUDisplay/opmode:1.0</supportedType>
    <provider>true</provider>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>false</exclusiveUser>
    <optional>true</optional>
    <kind>Facet</kind>
  </port>
  [...]
</Deployment:ComponentInterfaceDescription>
```



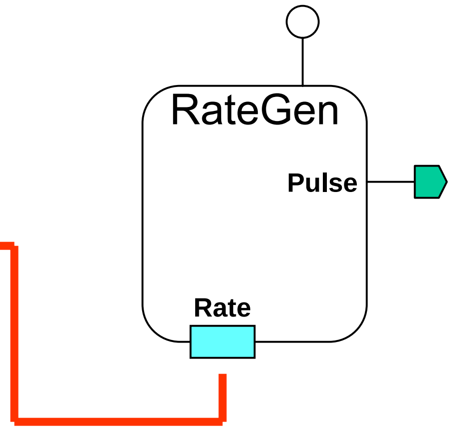
Component Interface Descriptor for RateGen component: RateGen.ccd (2/3)

```
<Deployment:ComponentInterfaceDescription>
  [...]
 <port>
    <name>Pulse</name>
    <specificType>IDL:HUDisplay/tick:1.0</specificType>
    <supportedType>IDL:HUDisplay/tick:1.0</supportedType>
    <provider>false</provider>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>false</exclusiveUser>
    <optional>true</optional>
    <kind>EventPublisher</kind>
  </port>
  <property>
    <name>Rate</name>
    <type>
      <kind>tk_long</kind>
    </type>
  </property>
  [...]
</Deployment:ComponentInterfaceDescription>
```



Component Interface Descriptor for RateGen component: RateGen.ccd (3/3)

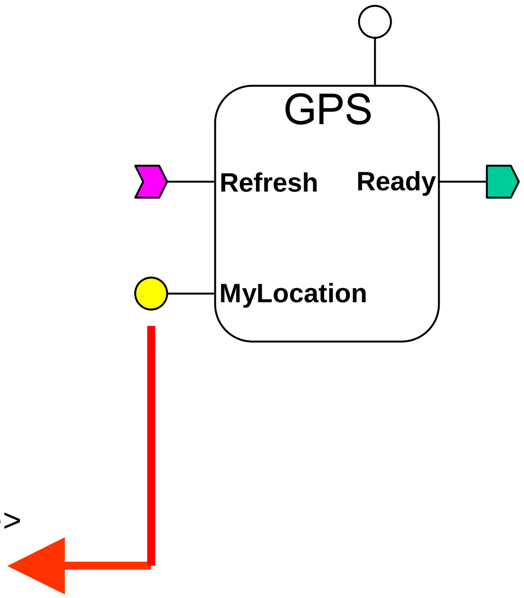
```
<Deployment:ComponentInterfaceDescription>
  [...]
 <configProperty>
    <name>Rate</name>
    <value>
      <type>
        <kind>tk_long</kind>
      </type>
      <value>
        <long>1</long>
      </value>
    </value>
  </configProperty>
</Deployment:ComponentInterfaceDescription>
```



- Default value for Rate property
  – Can be overridden by implementation, package, assembly, user, or at deployment time

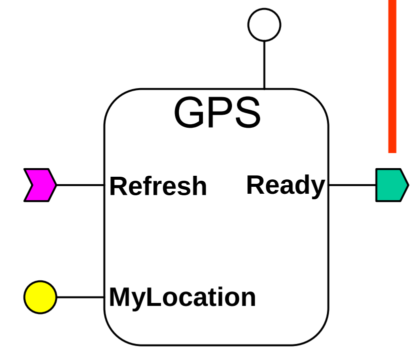
Component Interface Descriptor for GPS component: GPS.ccd (1/2)

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Deployment:ComponentInterfaceDescription
    xmlns:Deployment='http://www.omg.org/Deployment'
    xmlns:xmi='http://www.omg.org/XMI'
    >
 <label>Positioning Sensor</label>
  <specificType>IDL:HUDisplay/GPS:1.0</specificType>
  <supportedType>IDL:HUDisplay/GPS:1.0</supportedType>
  <idlFile>GPS.idl</idlFile>
  <port>
    <name>MyLocation</name>
    <specificType>IDL:HUDisplay/position:1.0</specificType>
    <supportedType>IDL:HUDisplay/position:1.0</supportedType>
    <provider>true</provider>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>false</exclusiveUser>
    <optional>true</optional>
    <kind>Facet</kind>
  </port>
  [...]
</Deployment:ComponentInterfaceDescription>
```



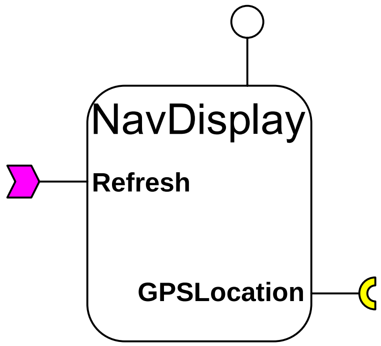
Component Interface Descriptor for GPS component: GPS.ccd (2/2)

```
<Deployment:ComponentInterfaceDescription> [...]
 <port>
    <name>Ready</name>
    <specificType>IDL:HUDisplay/tick:1.0</specificType>
    <supportedType>IDL:HUDisplay/tick:1.0</supportedType>
    <provider>false</provider>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>false</exclusiveUser>
    <optional>true</optional>
    <kind>EventPublisher</kind>
  </port>
  <port>
    <name>Refresh</name>
    <specificType>IDL:HUDisplay/tick:1.0</specificType>
    <supportedType>IDL:HUDisplay/tick:1.0</supportedType>
    <provider>true</provider>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>false</exclusiveUser>
    <optional>false</optional>
    <kind>EventConsumer</kind>
  </port>
</Deployment:ComponentInterfaceDescription>
```



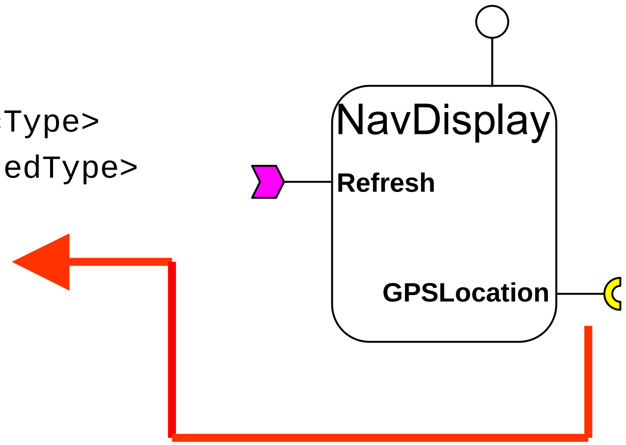
Component Interface Descriptor for NavDisplay component: NavDisplay.ccd (1/2)

```
<Deployment:ComponentInterfaceDescription
    xmlns:Deployment='http://www.omg.org/Deployment'
    xmlns:xmi='http://www.omg.org/XMI'
    >
  <label>Display Device</label>
  <specificType>IDL:HUDisplay/NavDisplay:1.0</specificType>
  <supportedType>IDL:HUDisplay/NavDisplay:1.0</supportedType>
  <idlFile>NavDisplay.idl</idlFile>
  <port>
    <name>Refresh</name>
    <specificType>IDL:HUDisplay/tick:1.0</specificType>
    <supportedType>IDL:HUDisplay/tick:1.0</supportedType>
    <provider>true</provider>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>false</exclusiveUser>
    <optional>false</optional>
    <kind>EventConsumer</kind>
  </port>
 [...]
</Deployment:ComponentInterfaceDescription>
```

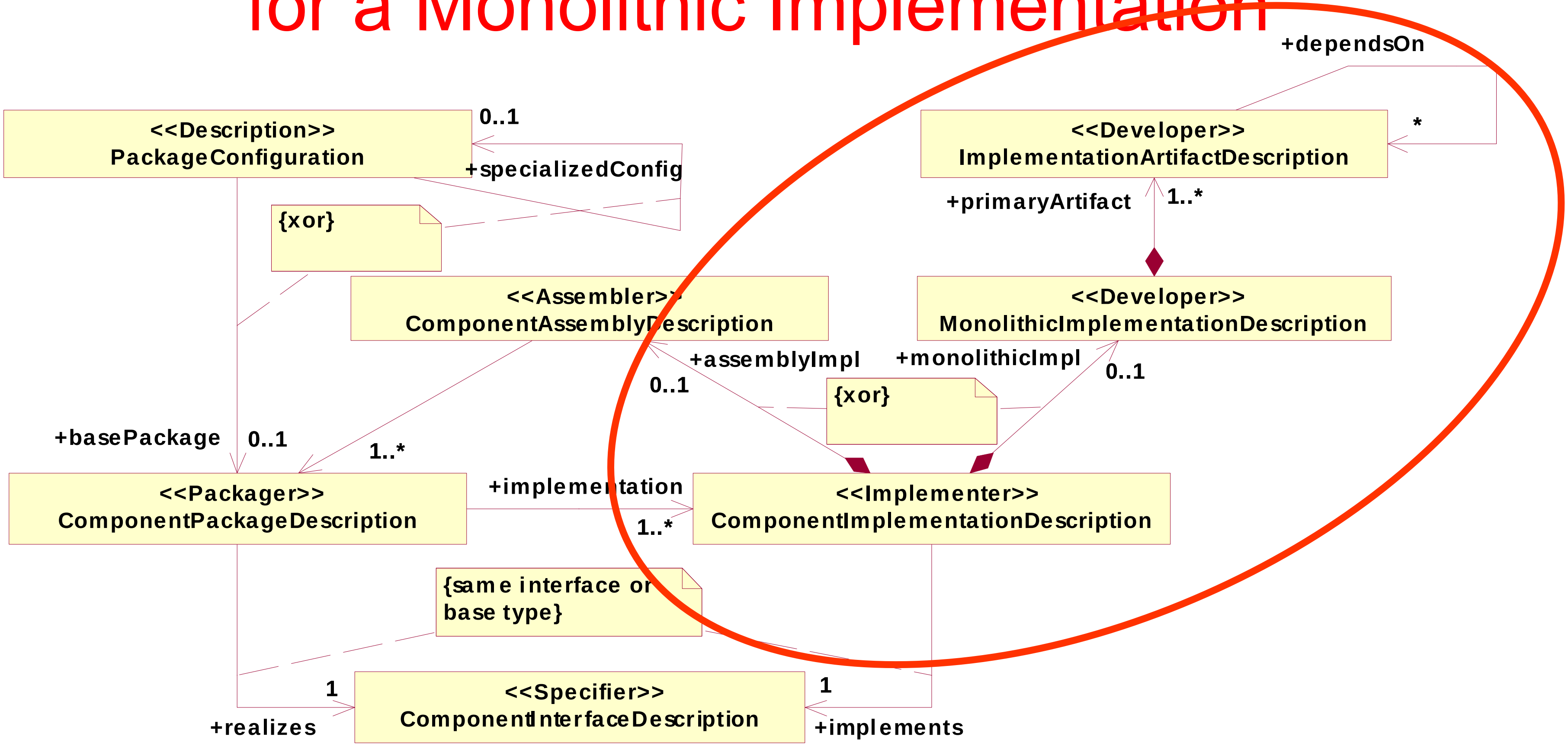


Component Interface Descriptor for NavDisplay component: NavDisplay.ccd (2/2)

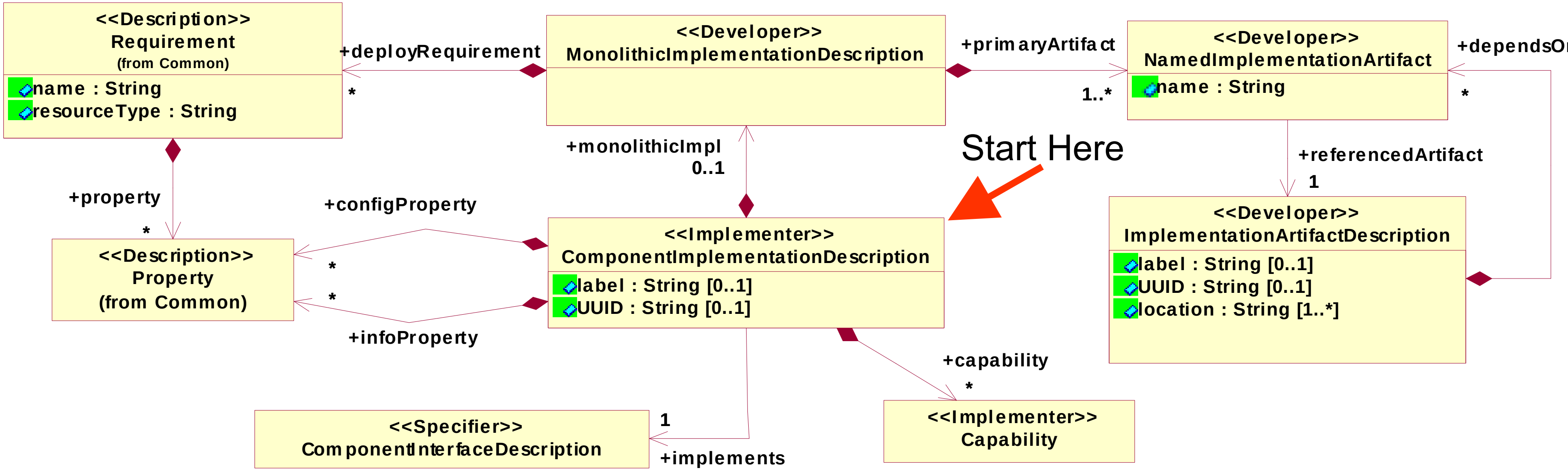
```
<Deployment:ComponentInterfaceDescription>
 [...]
 <port>
    <name>GPSLocation</name>
    <specificType>IDL:HUDisplay/position:1.0</specificType>
    <supportedType>IDL:HUDisplay/position:1.0</supportedType>
    <provider>false</provider>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>true</exclusiveUser>
    <optional>false</optional>
    <kind>SimplexReceptacle</kind>
  </port>
</Deployment:ComponentInterfaceDescription>
```



Component Implementation Description for a Monolithic Implementation



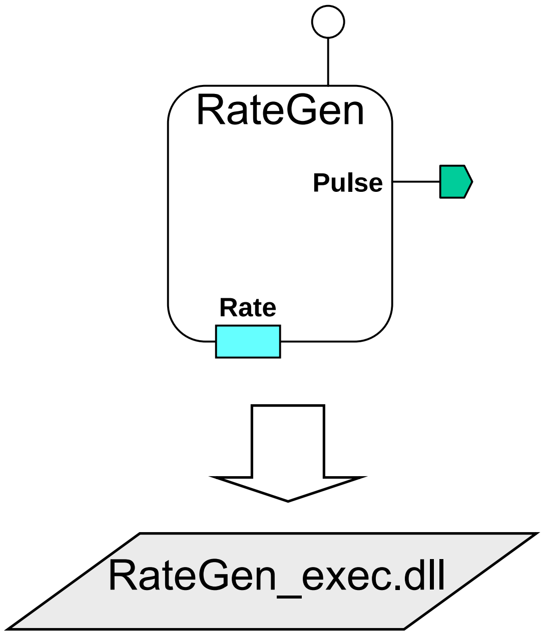
Component Implementation Description for a Monolithic Implementation



- Meta-data to describe a monolithic component implementation
  - Has deployment requirements, QoS capabilities
  - References artifacts by URL, which may have dependencies

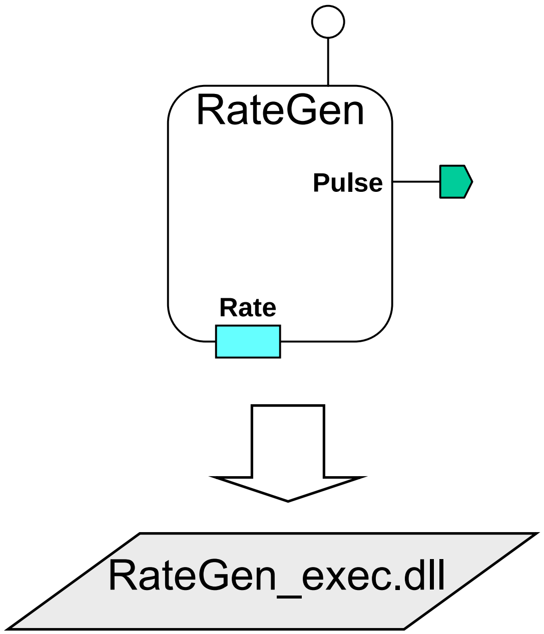
Component Implementation Descriptor for RateGen component: RateGen.cid (1/2)

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Deployment:ComponentImplementation
    xmlns:Deployment='http://www.omg.org/Deployment'
    xmlns:xmi='http://www.omg.org/XMI'>
  <implements href="RateGen.ccd"/>
  <monolithicImpl>
    <primaryArtifact>
      <name>RateGen Executor</name>
      <referencedArtifact>
        <location>RateGen_exec.dll</location>
        <dependsOn>
          <name>CIAO Library</name>
          <referencedArtifact>
            <location>CIAO.dll</location>
          </referencedArtifact>
        </dependsOn>
      </referencedArtifact>
    </primaryArtifact>
    [...]
  </monolithicImpl>
</Deployment:ComponentImplementationDescription>
```



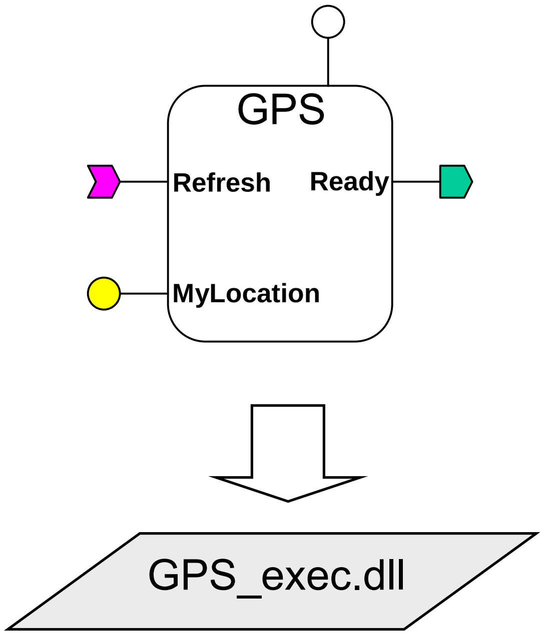
Component Implementation Descriptor for RateGen component: RateGen.cid (2/2)

```
<Deployment:ComponentImplementation>
  <monolithicImpl> [...]
    <deployRequirement>
      <name>os</name>
      <resourceType>Operating System</resourceType>
      <property>
        <name>version</name>
        <value>
          <type>
            <kind>tk_string</kind>
          </type>
          <value>
            <string>Windows 2000</string>
          </value>
        </value>
      </property>
    </deployRequirement>
  </monolithicImpl>
</Deployment:ComponentImplementationDescription>
```



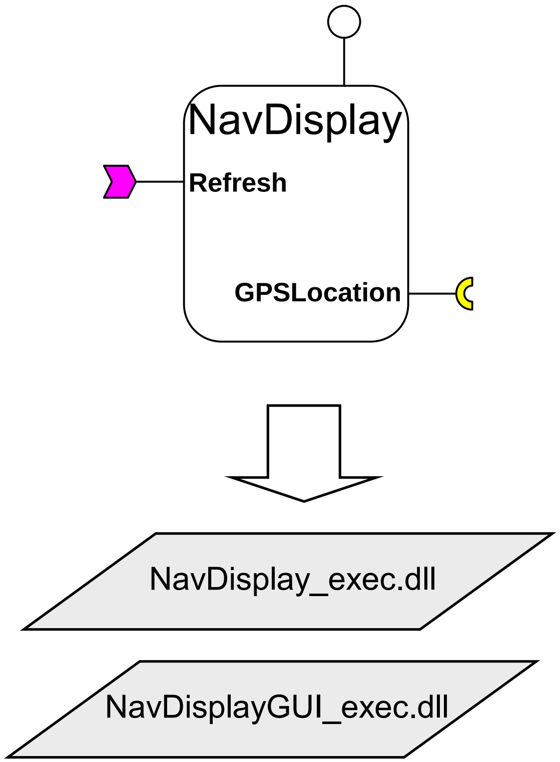
Component Implementation Descriptor for GPS component: GPS.cid (excerpt)

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Deployment:ComponentImplementationDescription>
  <monolithicImpl> [...]
    <deployRequirement>
      <name>GPS</name>
      <resourceType>GPS Device</resourceType>
      <property>
        <name>vendor</name>
        <value>
          <type>
            <kind>tk_string</kind>
          </type>
          <value>
            <string>My Favorite GPS Vendor</string>
          </value>
        </value>
      </property>
    </deployRequirement>
    [... Requires Windows OS ...]
  </monolithicImpl>
</Deployment:ComponentImplementationDescription>
```

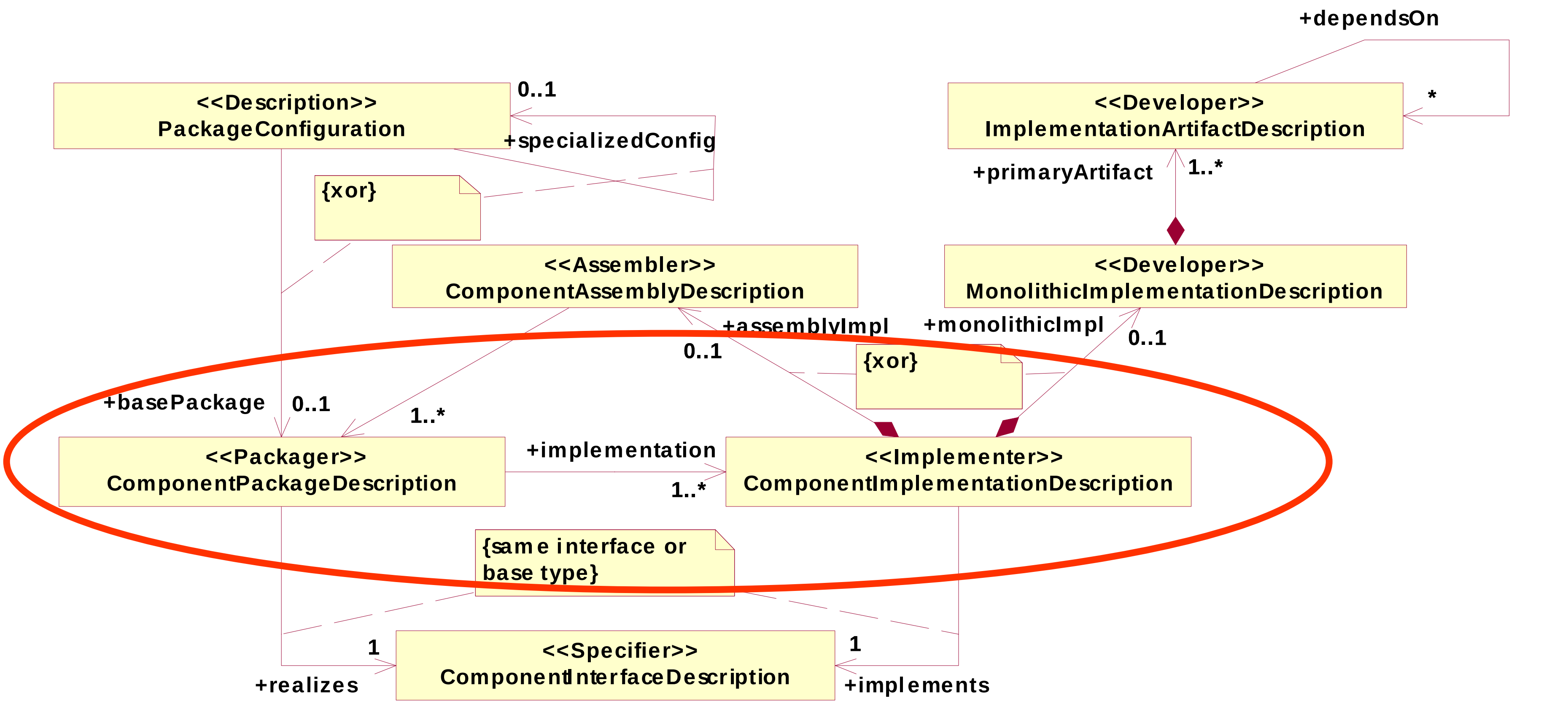


Two Component Implementation Descriptors for NavDisplay component

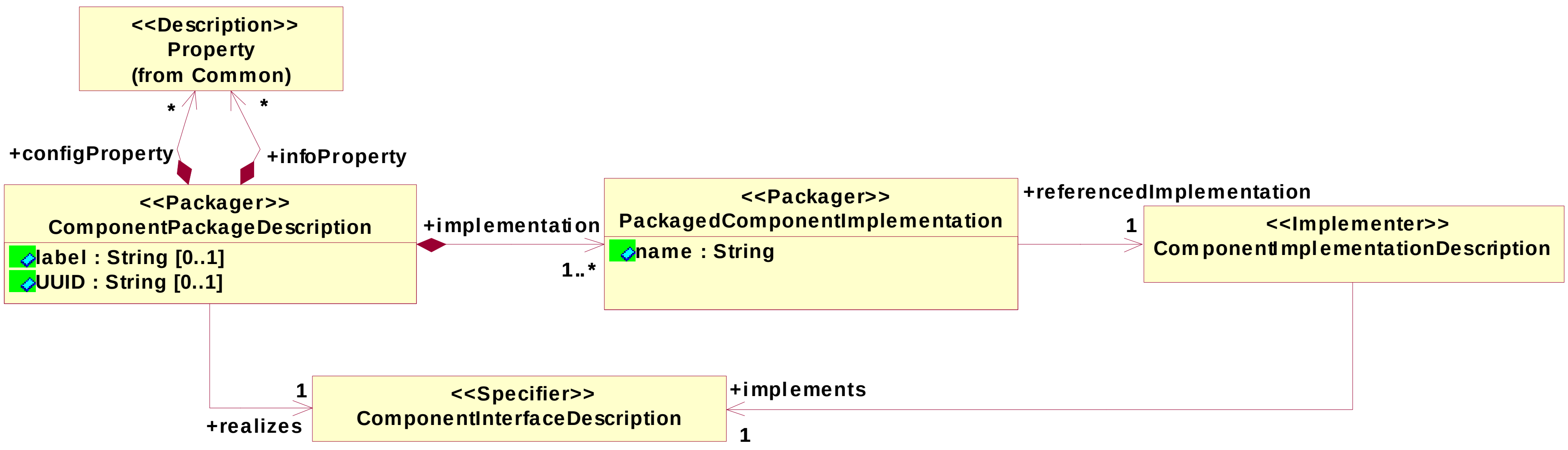
- Two alternative implementations
  - Therefore, two Component Implementation Descriptor files
- NavDisplay.cid
  - text-based implementation
- NavDisplayGUI.cid
  - GUI implementation
  - “deployRequirement” on graphical display
- XML code not shown



Component Package Description



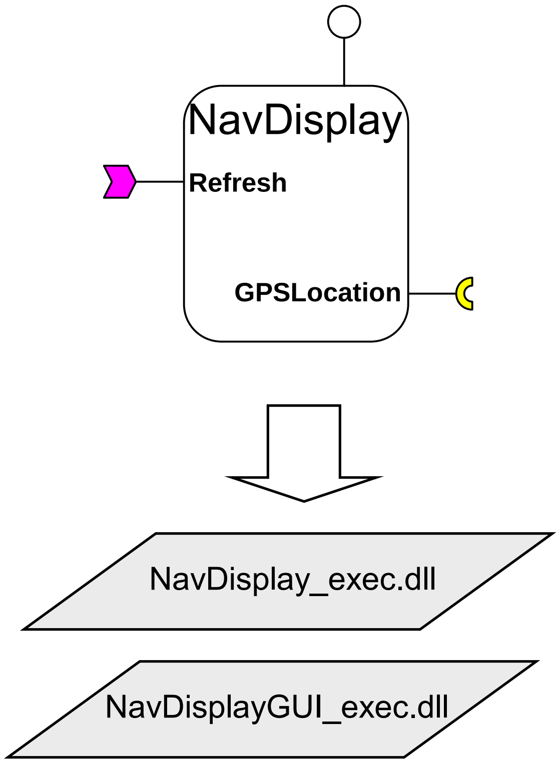
Component Package Description



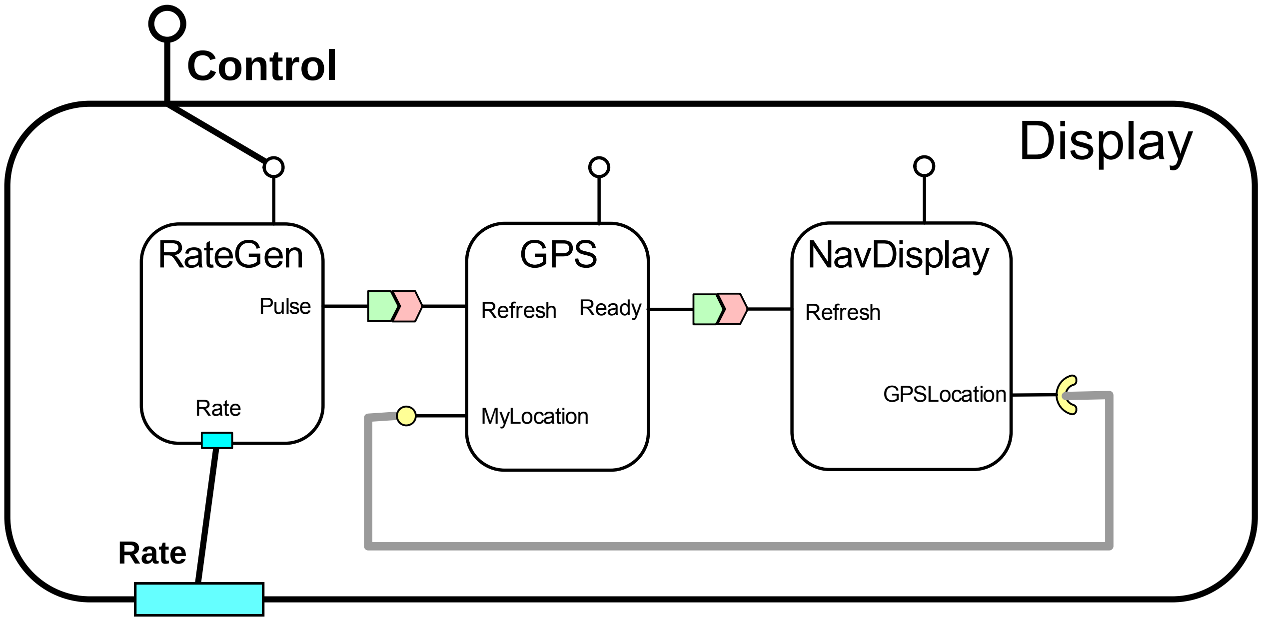
- Meta-data to describe a set of alternative implementations of the same component
  - May redefine (overload) properties

Component Package Descriptor for NavDisplay component: NavDisplay.cpd (1/1)

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Deployment:ComponentPackageDescription
    xmlns:Deployment='http://www.omg.org/Deployment'
    xmlns:xmi='http://www.omg.org/XMI'
    >
  <label>Display Device</label>
  <realizes href="NavDisplay.ccd"/>
  <implementation>
    <name>Text-based Display</name>
    <referencedImplementation href="NavDisplay.cid"/>
  </implementation>
  <implementation>
    <name>Graphical Display</name>
    <referencedImplementation href="NavDisplayGUI.cid"/>
  </implementation>
</Deployment:ComponentPackageDescription>
```

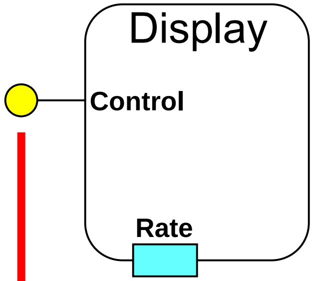


Display Component Assembly

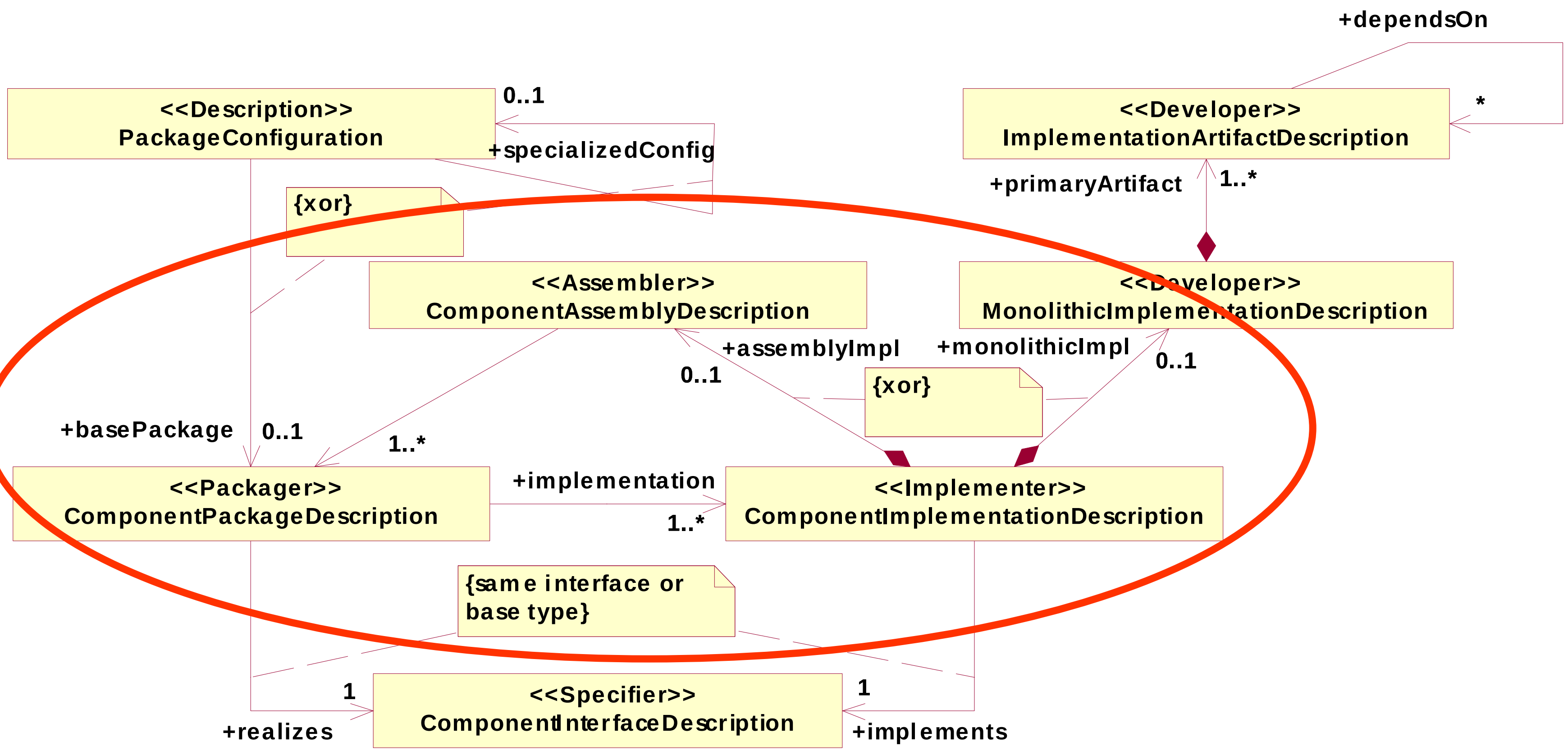


Component Interface Descriptor for Display component: Display.ccd (1/1)

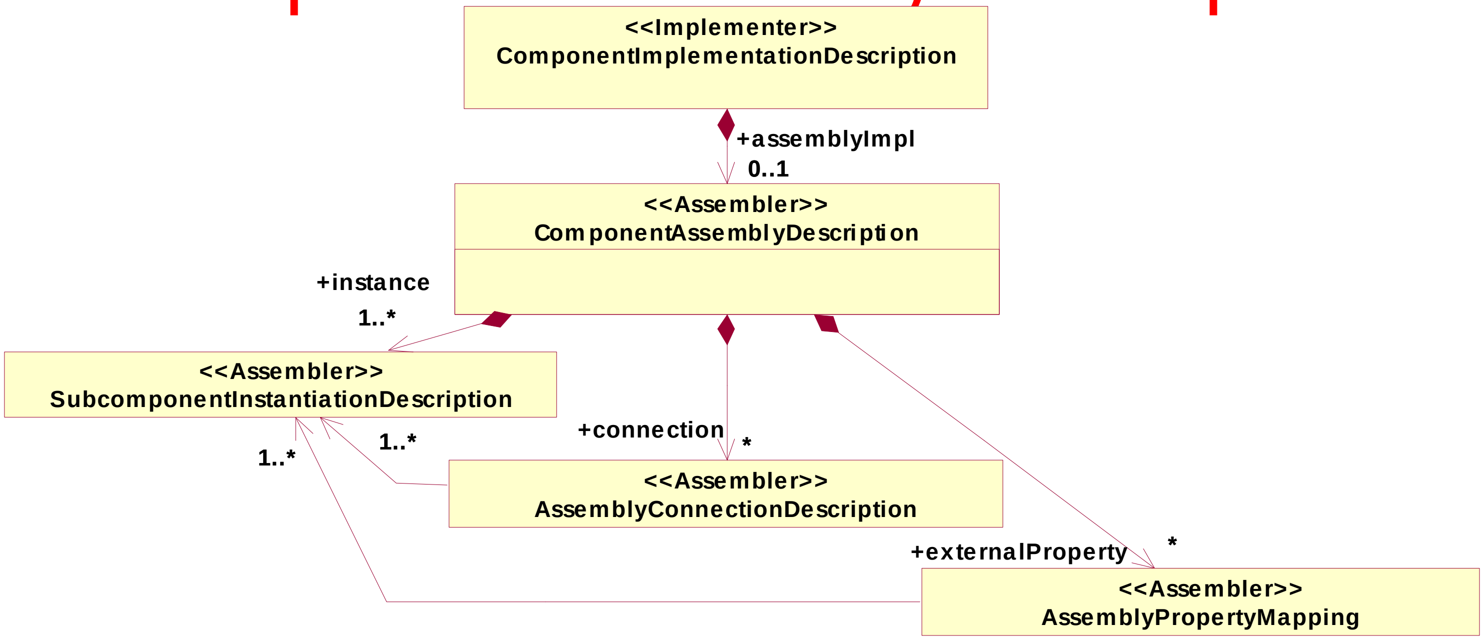
```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Deployment:ComponentInterfaceDescription
    xmlns:Deployment='http://www.omg.org/Deployment'>
  <label>Navigation System</label>
  <specificType>IDL:HUDisplay/Display:1.0</specificType>
  <idlFile>Display.idl</idlFile>
  <port>
    <name>control</name>
    <specificType>IDL:HUDisplay/opmode:1.0</specificType>
    <supportedType>IDL:HUDisplay/opmode:1.0</supportedType>
    <provider>true</provider>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>false</exclusiveUser>
    <optional>true</optional>
    <kind>Facet</kind>
  </port>
  <property>
    <name>Rate</name>
    <type>
      <kind>tk_long</kind>
    </type>
  </property>
</Deployment:ComponentInterfaceDescription>
```



Component Assembly Description



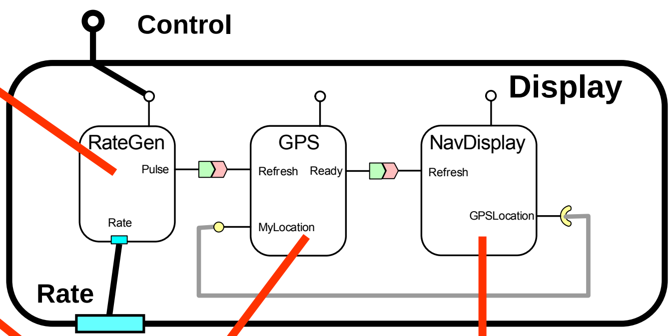
Component Assembly Description



- Meta-data to describe an assembly-based implementation
  - Subcomponent instances
  - Connections between subcomponents' ports
  - Mapping an assembly's properties to subcomponent properties

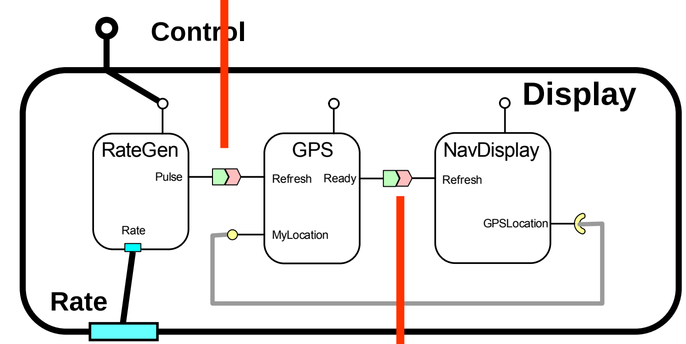
Component Implementation Descriptor for Display component: Display.cid (1/4)

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Deployment:ComponentImplementationDescription
    xmlns:Deployment='http://www.omg.org/Deployment'
    xmlns:xmi='http://www.omg.org/XMI'
    >
  <implements href="Display.ccd"/>
  <assemblyImpl>
    <instance xmi:id="RateGen">
      <name>RateGen Subcomponent</name>
      <package href="RateGen.cpd"/>
    </instance>
    <instance xmi:id="GPS">
      <name>GPS Subcomponent</name>
      <package href="GPS.cpd"/>
    </instance>
    <instance xmi:id="NavDisplay">
      <name>NavDisplay Subcomponent</name>
      <package href="NavDisplay.cpd"/>
    </instance>
    [...]
  </assemblyImpl>
</Deployment:ComponentImplementationDescription>
```



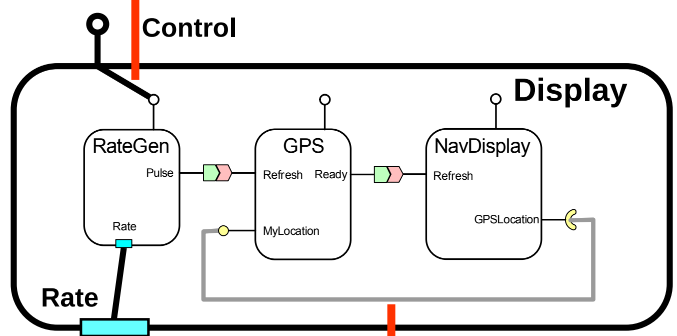
Component Implementation Descriptor for Display component: Display.cid (2/4)

```
<Deployment:ComponentImplementationDescription>
  <assemblyImpl> [...]
    <connection> <name>GPS Trigger</name>
      <internalEndpoint>
        <portName>Pulse</portName>
        <instance href="#RateGen"/>
      </internalEndpoint>
      <internalEndpoint>
        <portName>Refresh</portName>
        <instance href="#GPS"/>
      </internalEndpoint>
    </connection>
    <connection> <name>NavDisplay Trigger</name>
      <internalEndpoint>
        <portName>Ready</portName>
        <instance href="#GPS"/>
      </internalEndpoint>
      <internalEndpoint>
        <portName>Refresh</portName>
        <instance href="#NavDisplay"/>
      </internalEndpoint>
    </connection>
   [...] </assemblyImpl>
</Deployment:ComponentImplementationDescription>
```



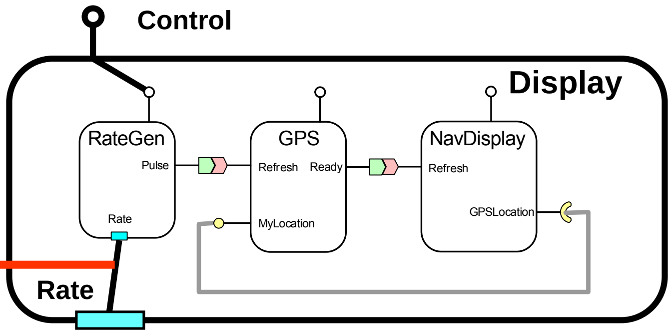
Component Implementation Descriptor for Display component: Display.cid (3/4)

```
<Deployment:ComponentImplementationDescription>
  <assemblyImpl> [...]
    <connection> <name>control port</name>
      <externalEndpoint>
        <portName>Control</portName>
      </externalEndpoint>
      <internalEndpoint>
        <portName>supports</portName>
        <instance href="#RateGen"/>
      </internalEndpoint>
    </connection>
    <connection> <name>Location</name>
      <internalEndpoint>
        <portName>MyLocation</portName>
        <instance href="#GPS"/>
      </internalEndpoint>
      <internalEndpoint>
        <portName>GPSLocation</portName>
        <instance href="#NavDisplay"/>
      </internalEndpoint>
    </connection>
   [...] </assemblyImpl>
</Deployment:ComponentImplementationDescription>
```

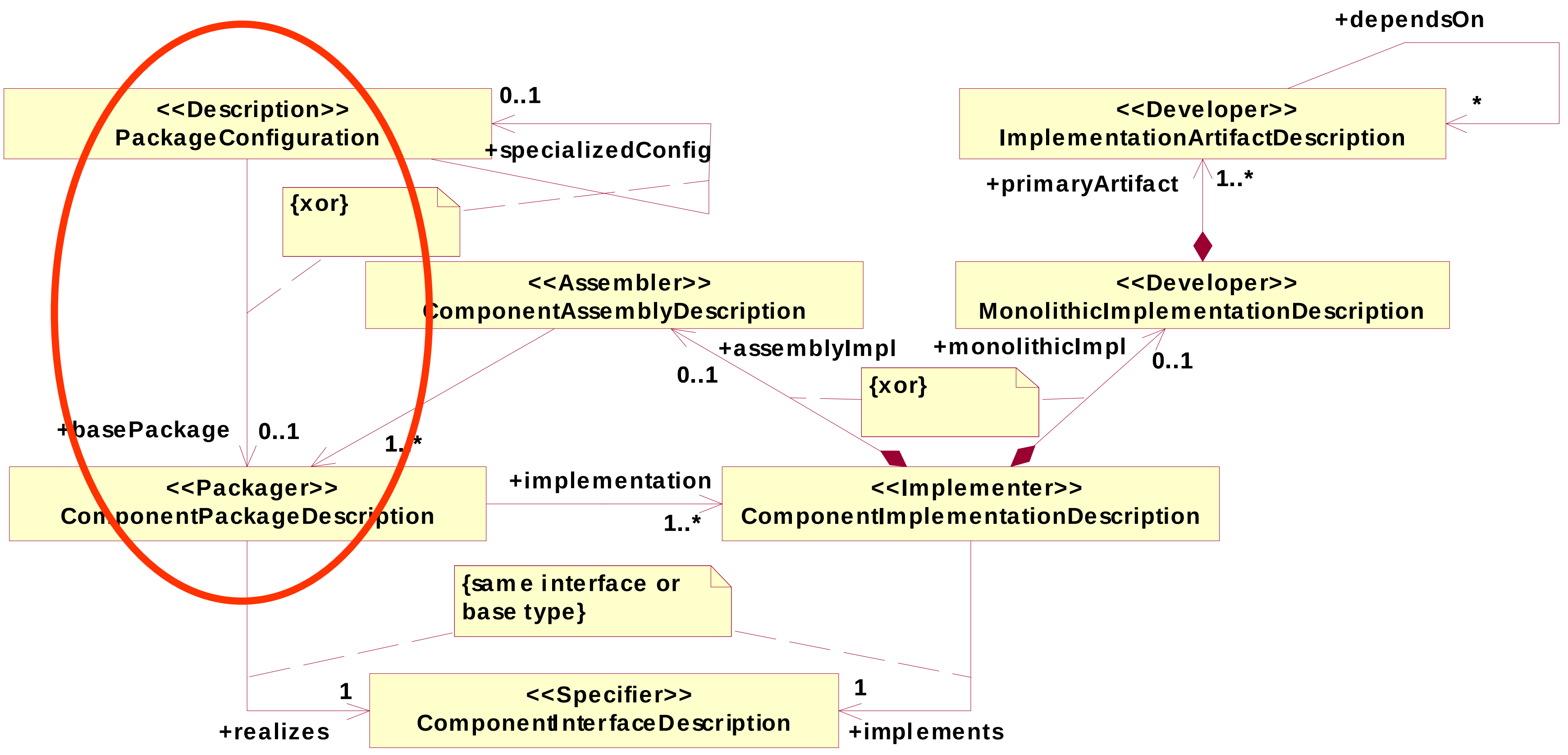


Component Implementation Descriptor for Display component: Display.cid (4/4)

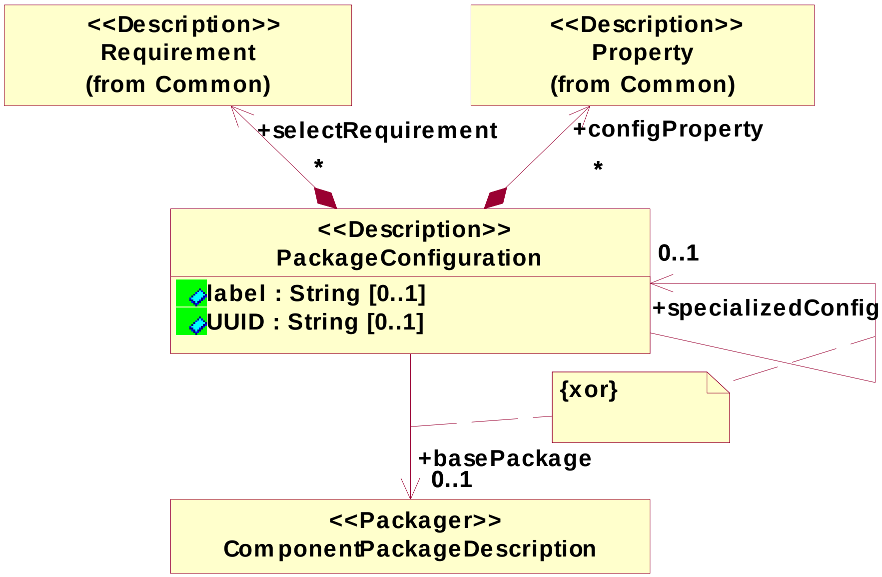
```
<Deployment:ComponentImplementationDescription>
  <assemblyImpl>
    [...]
    <externalProperty>
      <name>Rate Mapping</name>
      <externalName>Rate</externalName>
      <delegatesTo>
        <propertyName>Rate</propertyName>
        <instance href="#RateGen"/>
      </delegatesTo>
    </externalProperty>
  </assemblyImpl>
</Deployment:ComponentImplementationDescription>
```



Package Configuration



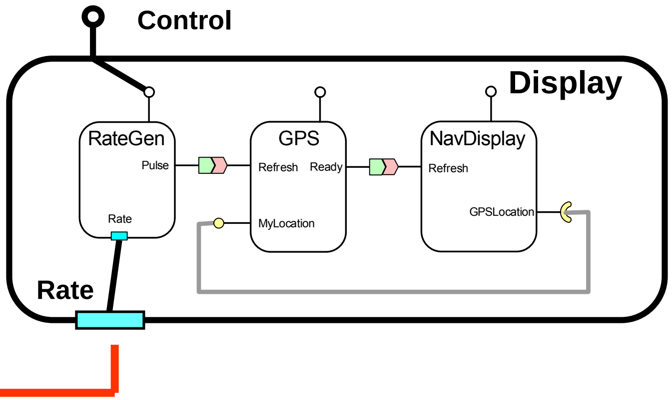
Package Configuration



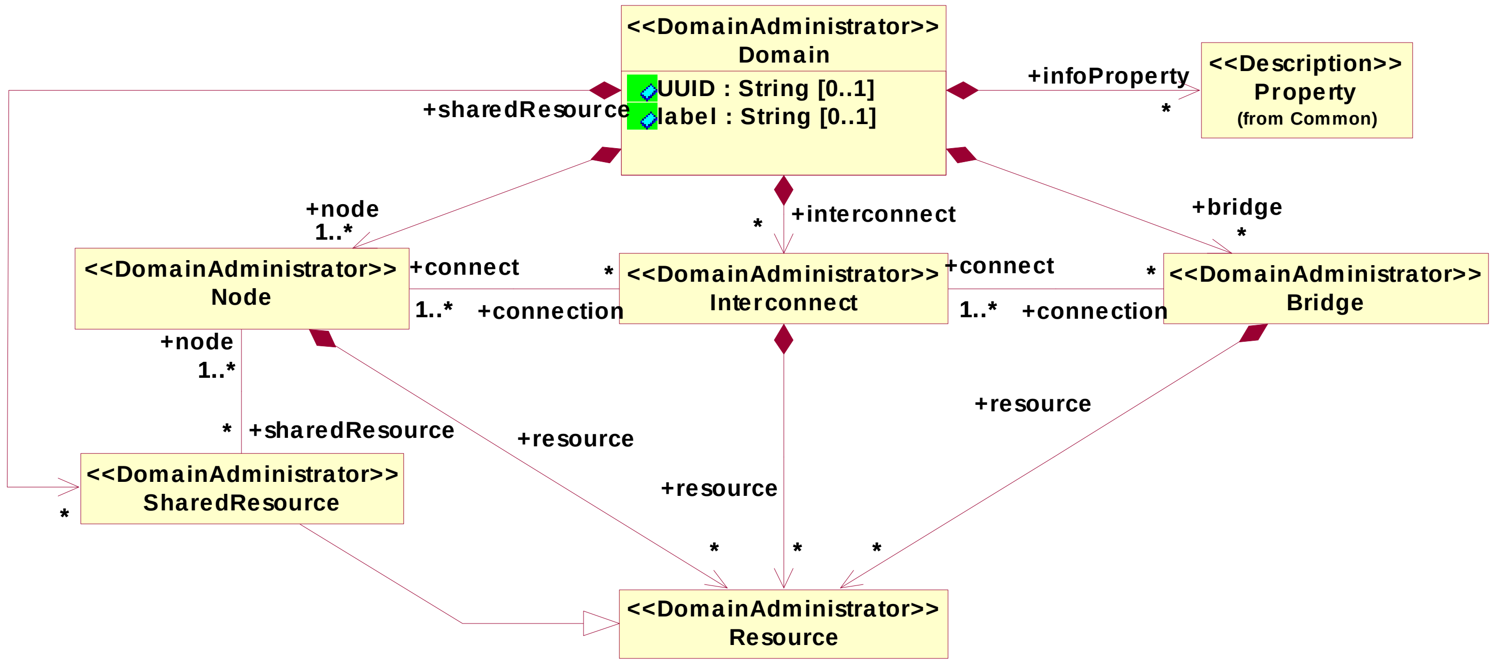
- Meta-data to describe a reusable component package
  - Sets initial configuration
  - Sets QoS requirements
    - to be matched against implementation capabilities
  - May refine (specialize) existing package

Package Configuration for Display application: Display.pcd (1/1)

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Deployment:PackageConfiguration
    xmlns:Deployment='http://www.omg.org/Deployment'
    xmlns:xmi='http://www.omg.org/XMI'
    >
  <label>Display Application</label>
  <configProperty>
    <name>Rate</name>
    <value>
      <type>
        <kind>tk_long</kind>
      </type>
      <value>
        <long>10</long>
      </value>
    </value>
  </configProperty>
  <basePackage href="Display.cpd"/>
</Deployment:PackageConfiguration>
```

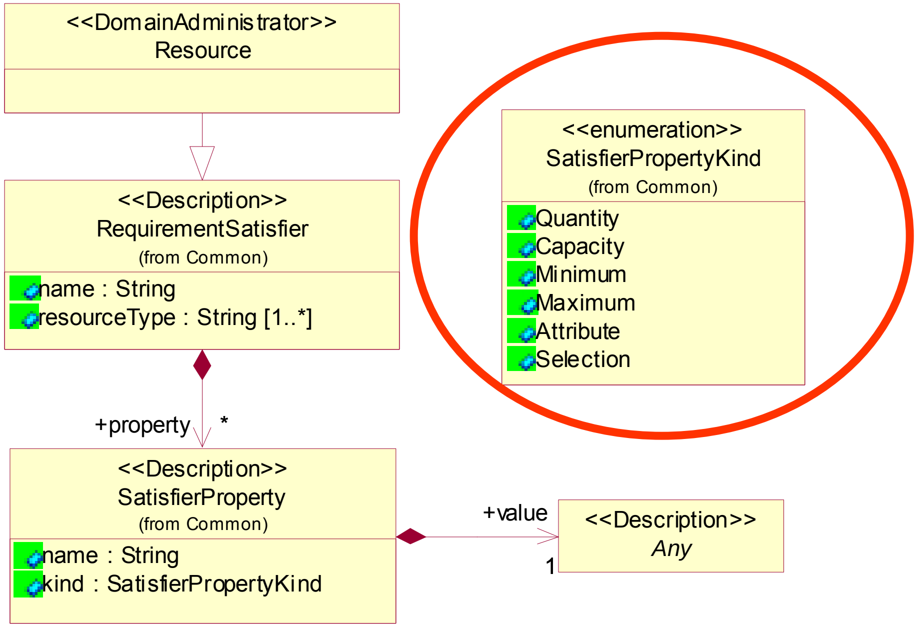


Target Data Model



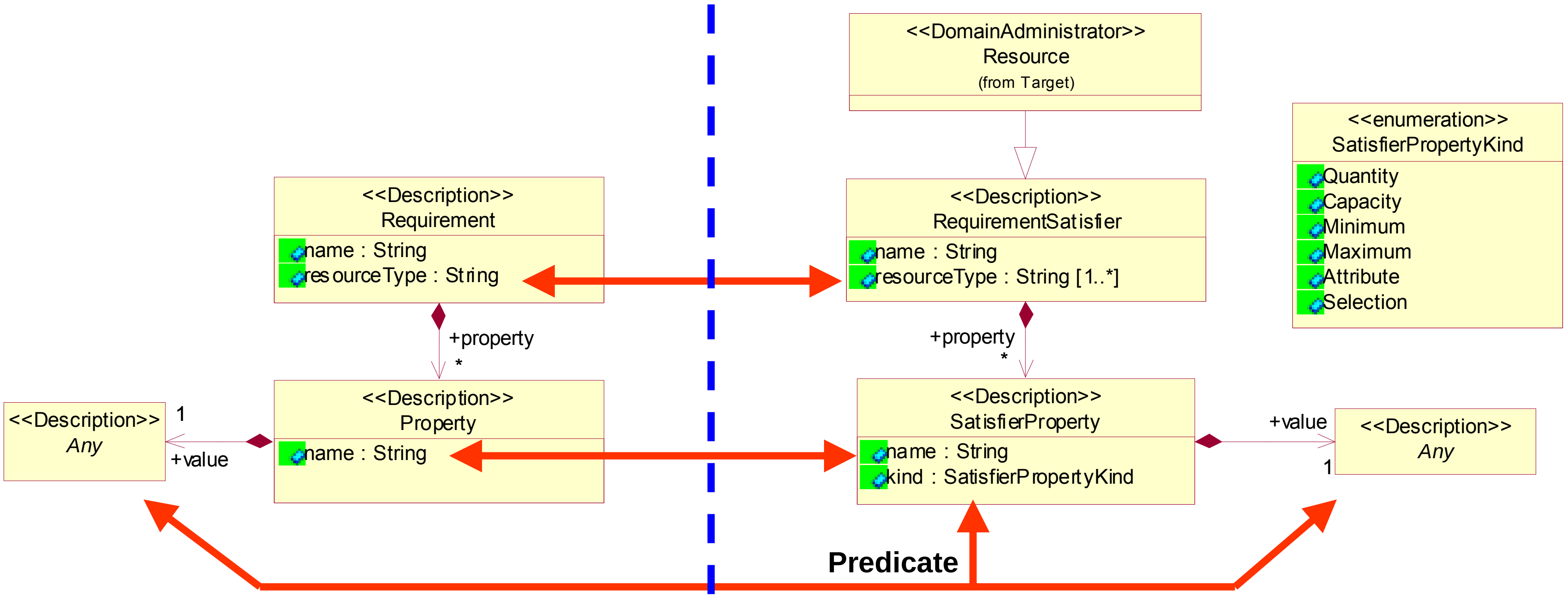
- Meta-data to describe a "target domain"
  - Nodes: targets for executing monolithic component implementations
  - Interconnect: direct connections (e.g., ethernet cable)
  - Bridge: indirect connections (e.g., routers, switches)

Target Data Model: Resources



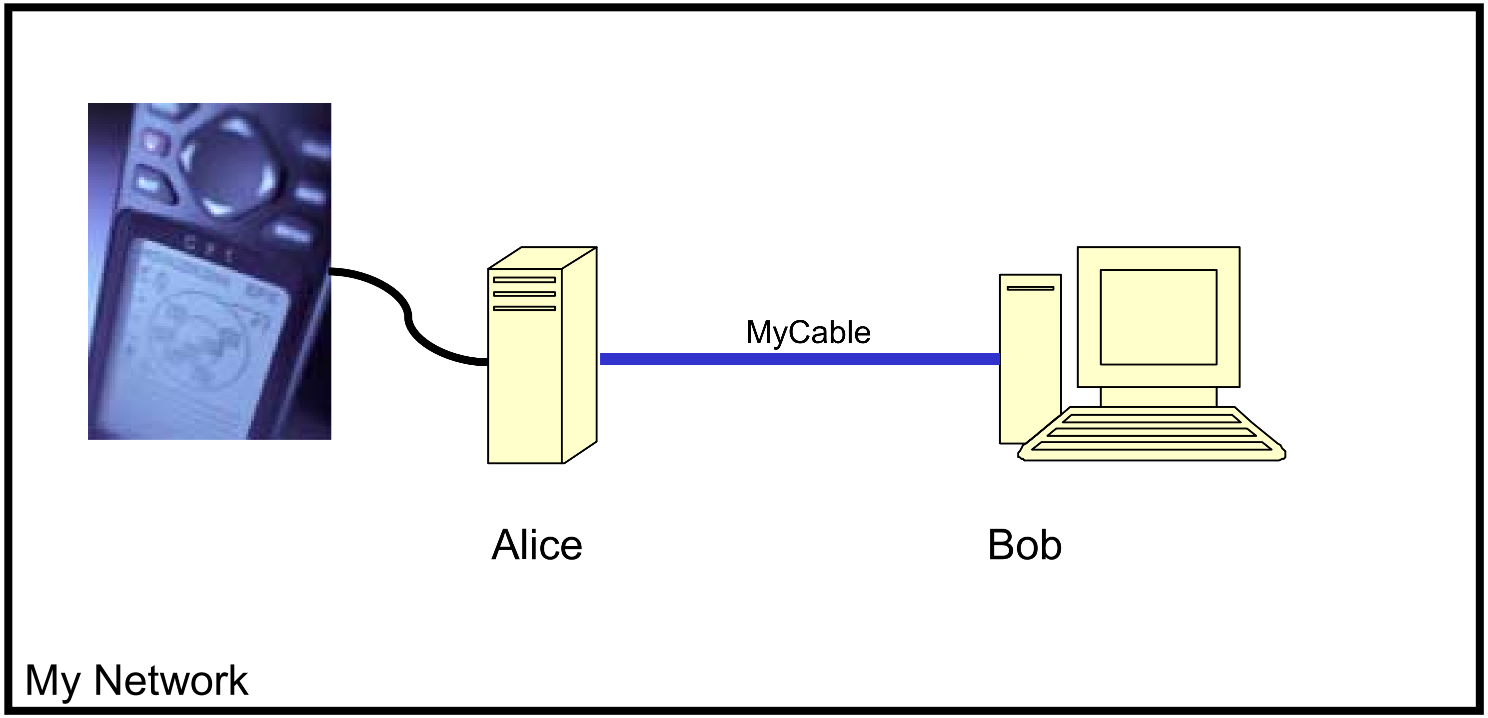
- Meta-data to describe a consumable resource
  - Satisfies a requirement (from monolithic implementation)
  - SatisfierPropertyKind: Operators and predicates to indicate if and how a resource property is "used up"

Matching Requirements against Resources



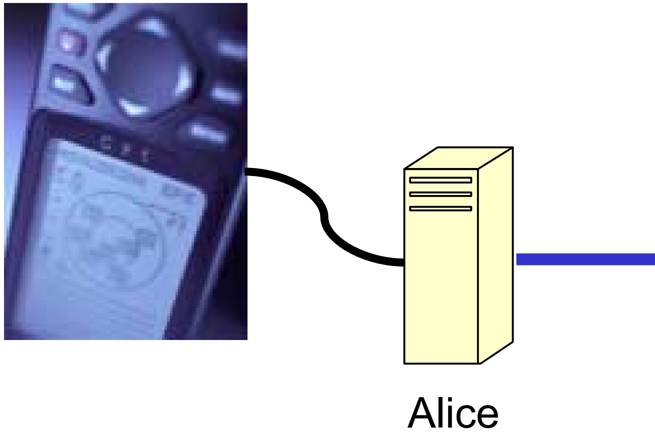
- Generic grammar for defining resources and requirements
- Well-defined, generic matching and accounting algorithm
  – Depending on predicate, resource capacity is "used up"

Example Domain



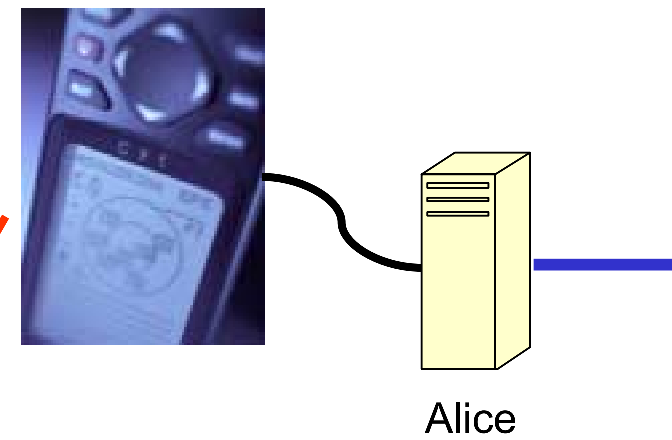
Domain Descriptor: MyNetwork.cdd (1/3)

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Deployment:Domain
    xmlns:Deployment='http://www.omg.org/Deployment'
    xmlns:xmi='http://www.omg.org/XMI'>
  <label>My Network</label>
  <node xmi:id="Alice">
    <name>Alice</name>
    <connection href='#MyCable'/>
    <resource>
      <name>os</name>
      <resourceType>Operating System</resourceType>
      <property>
        <kind>Attribute</kind>
        <name>version</name>
        <value>
          <type><kind>tk_string</kind></type>
          <value><string>Windows 2000</string></value>
        </value>
      </property>
    </resource>
    [...]
  </node>
</Deployment:Domain>
```



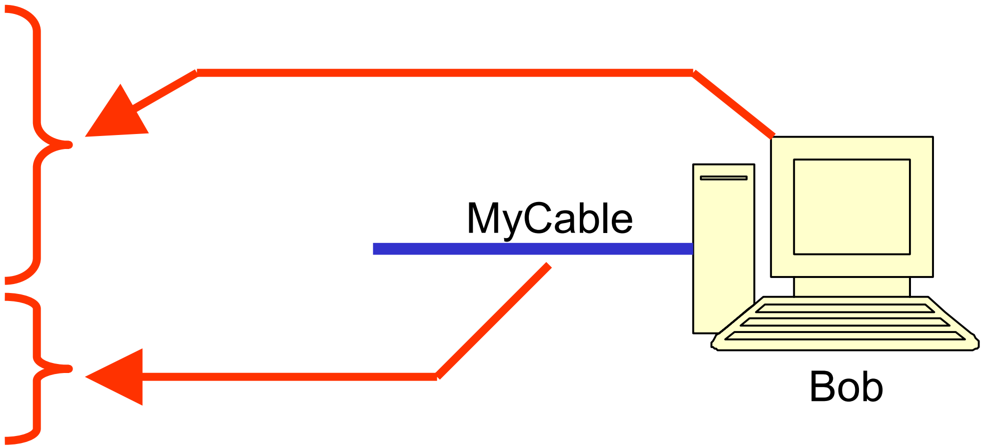
Domain Descriptor: MyNetwork.cdd (2/3)

```
<Deployment:Domain>
  <node>
    [...]
      <resource>
      <name>GPS</name>
      <resourceType>GPS Device</resourceType>
      <property>
        <name>vendor</name>
        <kind>Attribute</kind>
        <value>
          <type>
            <kind>tk_string</kind>
          </type>
          <value>
            <string>My Favorite GPS Vendor</string>
          </value>
        </value>
      </property>
    </resource>
  </node>
  [...]
</Deployment:Domain>
```



Domain Descriptor: MyNetwork.cdd (3/3)

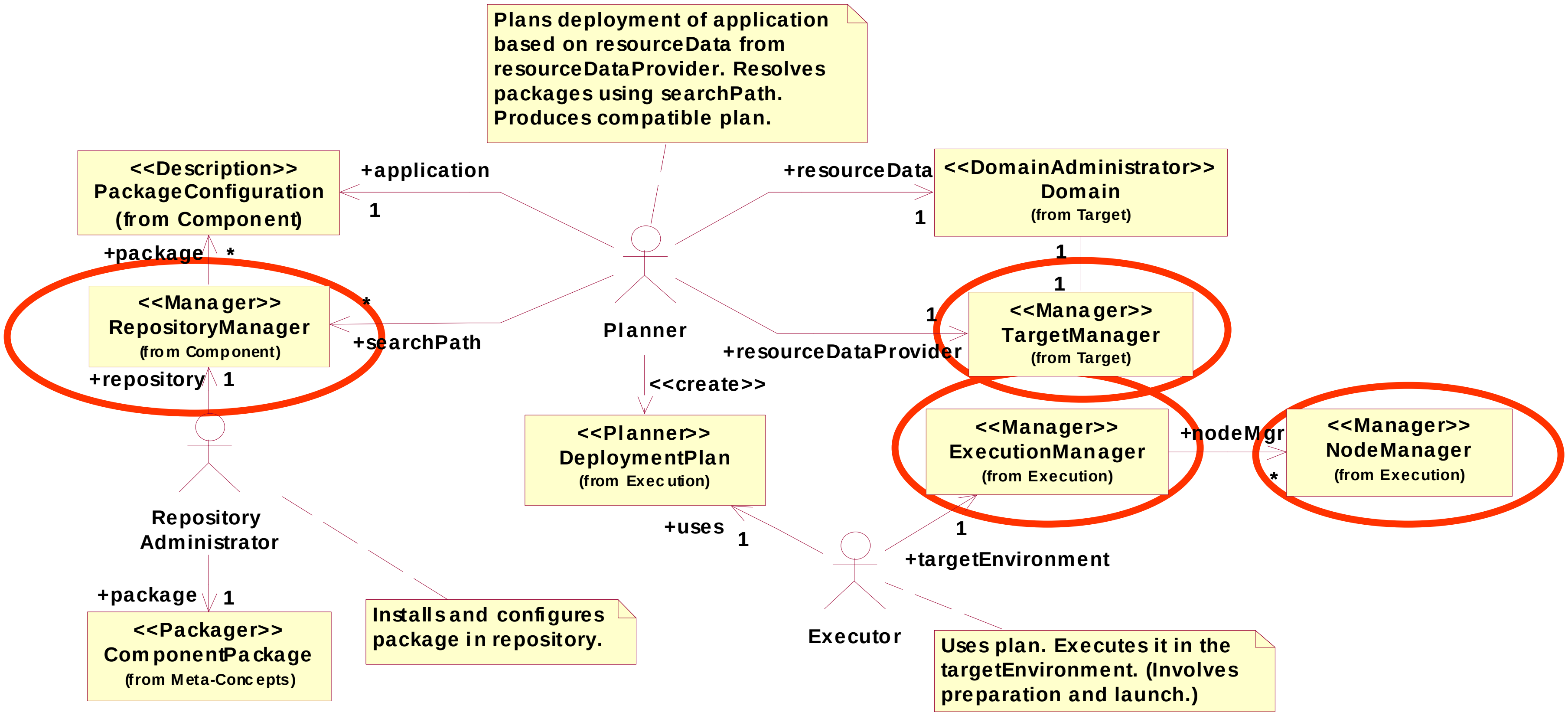
```
<Deployment:Domain>
  [...]
  <node xmi:id='Bob'>
    <name>Bob</name>
    <connection href='#MyCable'/>
    [... "Windows 2000" OS resource ...]
    [... "Graphical Display" resource ...]
  </node>
  <interconnect xmi:id='MyCable'>
    <connect href='#Alice'/>
    <connect href='#Bob'/>
  </interconnect>
</Deployment:Domain>
```



Deployment Infrastructure Overview

- **Repository Manager**
  - Database of applications that are available for deployment ("staging area")
- **Target Manager**
  - Retrieval of target data (i.e., available nodes and resources)
- **Execution Manager**
  - Execution of an application according to a "Deployment Plan"
- **Node Manager**
  - Execution of monolithic component implementations on a node
- The above are independent compliance points, managers from different vendors can interoperate
  - Especially important for vendor-specific Node Managers!

Deployment Infrastructure



Infrastructure (Services)

Deployment Infrastructure: Repository Manager

- Database of applications
  - Meta-data (from Component Data Model)
  - Artifacts (i.e., executable monolithic implementations)
- Applications can be configured
  - e.g., to apply custom policies, e.g., "background color" = "blue"
- Applications are installed from packages
  - ZIP files containing meta-data in XML format, & artifacts
- CORBA interface for installation of packages, retrieval and introspection of meta-data
- HTTP interface for downloading artifacts
  - Used by Node Managers during execution

Deployment Infrastructure: Target Manager

- Singleton service (i.e., one Target Manager per Domain)
- Retrieval of available or total resource capacities
- Allocation and release of resources (during application deployment)
- No "live" monitoring of resources implied (optional)
  - Assumption: all resources are properly allocated and released through this interface
  - Central management and tracking of resources
- Allows "off-line" scenarios where the possibility and the effect of deploying applications is studied
  - e.g., "Given this configuration, is it possible to run this set of applications concurrently? How?"

Deployment Infrastructure: Execution Manager

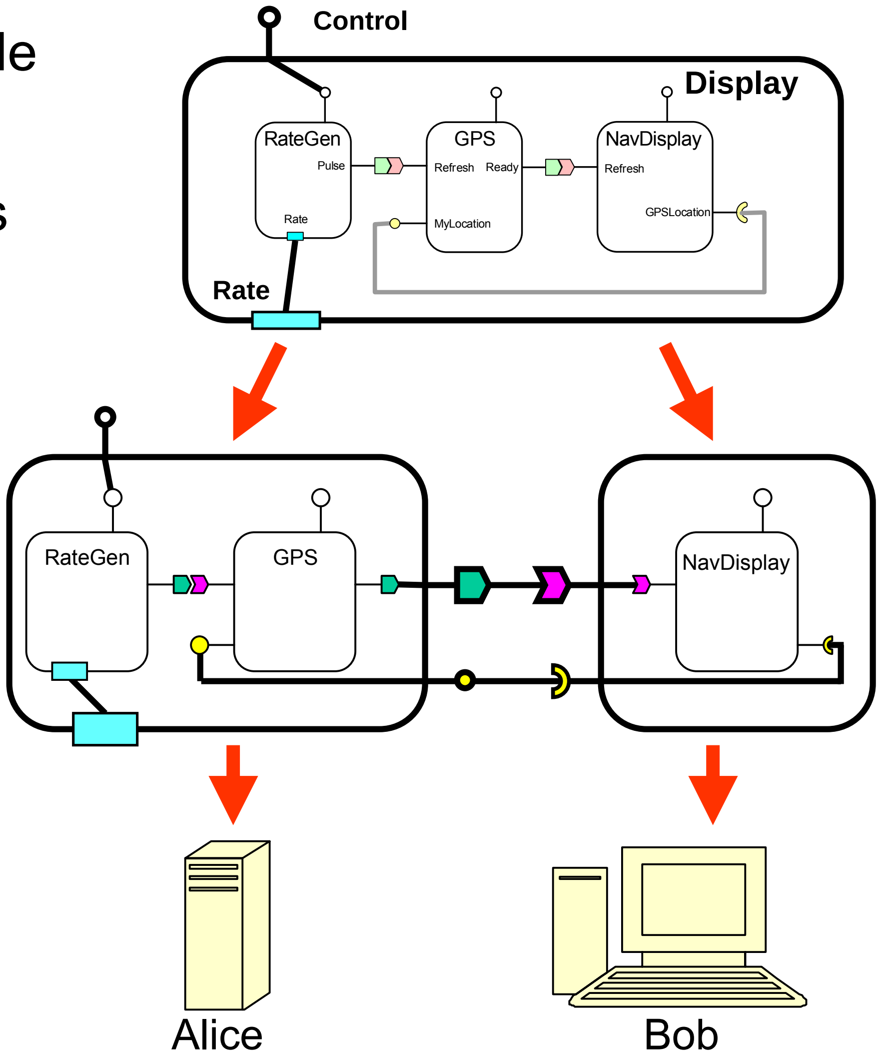
- Singleton service (i.e., one Execution Manager per Domain)
- User-visible front-end for executing a "Deployment Plan"
  - Deployment Plan is the result of planning for the deployment of an application, based on a specific set of nodes and resources (will be elaborated later)
- Instructs Node Managers to execute their respective per-node pieces of an application
  - Splits up the single "per domain" Deployment Plan into several partial, "per node" Deployment Plans
- Then interconnects these pieces

Deployment Infrastructure: Node Manager

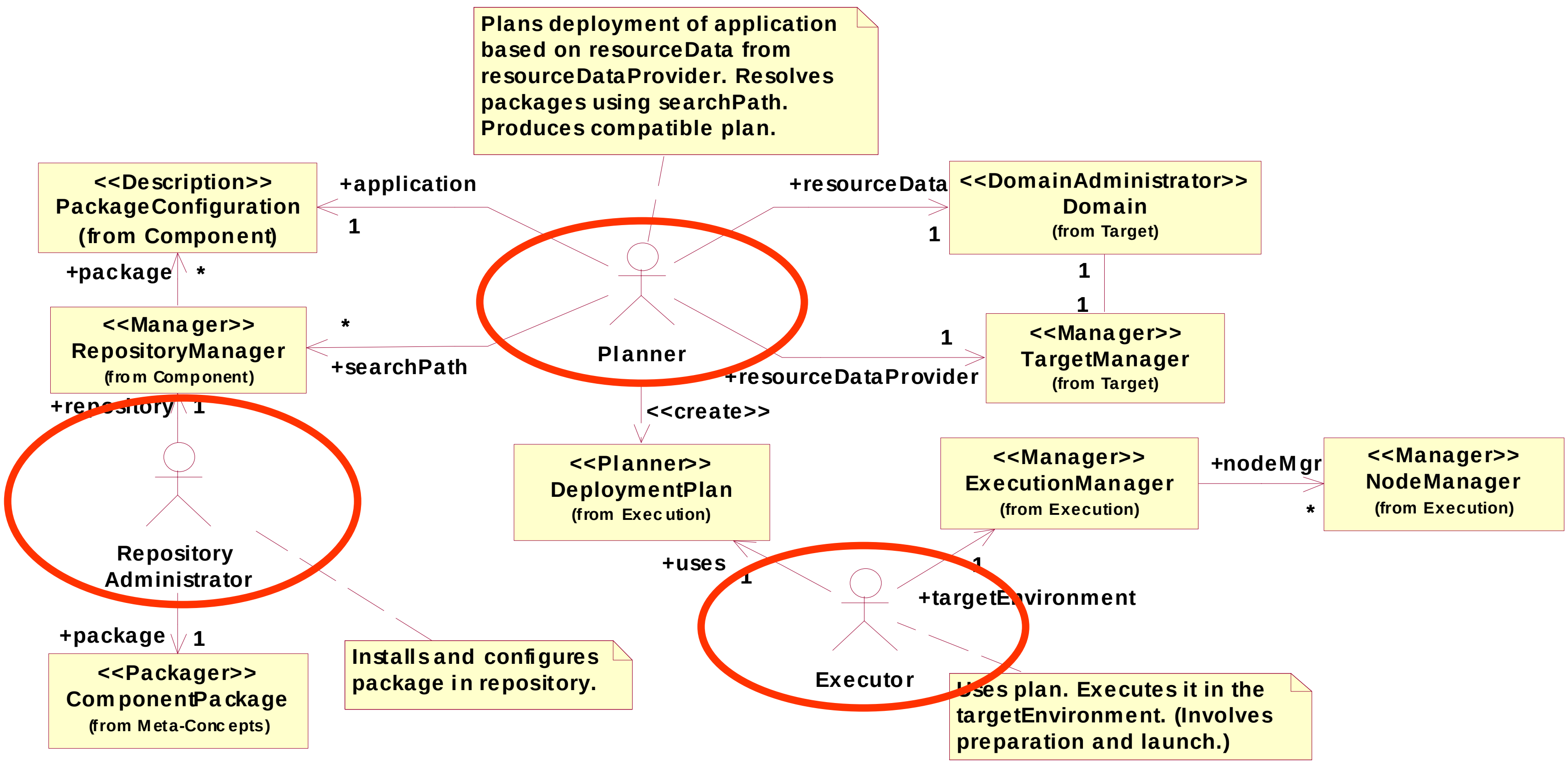
- One Node Manager instance per node; not user visible
- Instantiates component instances from monolithic implementations
- Encompasses CCM “container” concept
  - Not a replacement, just an abstraction
- Very system specific, important inter-vendor boundary:
  - Execution Manager part of generic deployment software
  - Node Manager(s) part of CCM implementation
- No colocation with “its” node implied
  - e.g., Node Manager can execute implementations remotely
  - e.g., for non-GPP nodes such as DSPs or FPGAs that might not be capable of running an ORB/OS/concurrent processes

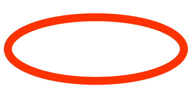
Exeuction/Node Managers Interaction

- Execution Manager computes per-node Deployment Plan
  - "Virtual" assemblies of components on the same node
  - Described using the same data structure
- All parts are sent to their respective Node Manager
  - Can be done concurrently
- Execution Manager then sends "provided" references to their users
- Transparent to "Executor" user



Deployment Actors



 Actors — usually, humans aided by software tools

Deployment Actors: Repository Administrator

- Receives component package from software vendor
- Installs package into repository, using Repository Manager
  - Assigns “installation name”
  - Optionally applies custom configuration properties
    - I.e., sets default values for an application’s external attributes (can be overridden during deployment)
  - Optionally sets “selection requirements”
    - Will be matched against implementation capabilities (during planning)
- Maintains repository contents
  - Browsing repository, updating packages, deleting packages …

Deployment Actors: Planner

- Accesses application meta-data from Repository Manager
  - Resolving referenced (“imported”) packages
- Accesses resource meta-data from Target Manager
  - Live “on-line” data, or simulated “off-line” data
- Matches requirements against resources
- Makes planning decisions
  - Selecting appropriate component implementations
  - Placing monolithic component instances onto nodes, assembly connections onto interconnects & bridges
- Produces Deployment Plan
  - “Off-line” plans can be stored for later (re-) use

Deployment Actors: Executor

- Passes Deployment Plan to Execution Manager
- Separate “Preparation” and “Launch” phases
- Preparation: prepares software for execution
  - Usually involves loading implementation artifacts to nodes
  - May (implementation specific) also involve pre-loading artifacts into memory, for faster launch
- Launch: starts application
  - Instantiating & configuring components
  - Interconnecting components
  - Starting components

Deployment Plan

- Self-contained data structure for executing an application within a specific domain, based on a specific set of resources
  - Records all decisions made during planning
    - Implementation selection, instance to node assignment, resource allocation
  - "Flat" assembly of instances of components with monolithic implementations (all assemblies are resolved)
- Does not contain implementation artifacts
  - Contains URLs to artifacts, as served up by the repository
    - HTTP mandatory, other protocols optional
  - Node Managers will download artifacts using these URLs

Planning Revisited

- Planning requires “intelligence”
  - Large search space for valid deployments
    - Considering all possibilities not practical; heuristics necessary
  - May implement “metric” to compare deployments
    - Prefer one component per node? As many components per node as possible?
  - Wide range of implementation options
    - Completely manual? Fully automatic?

►Planner is a separate piece, “outside” of the specification

- Only described as a “non-normative” actor
- Uses well-defined interfaces, “Deployment Plan” meta-data

Dynamic Planning Rationale

- Common D+C criticism: “Deployment Plan is too static”
  - Based on a snapshot of available resources
  - “Not well adapted to dynamic domain, when resource allocation changes, requiring to plan again from scratch”
- However, Deployment Plan is a necessity
  - Its information *must* be fully known at some point
- Future Idea:
  - Build more dynamic “planning” infrastructure *on top of* D+C’s building blocks – by extension, not replacement
    - E.g., “proto-plan” considering homogeneous nodes as equivalence classes (deferring concrete assignments)
    - Refinement into Deployment Plan as late as possible

Deployment Plan Rationale

- Common D+C criticism: "Who needs a Deployment Plan anyway?"
  - Why not have a combined Planner/Executor that immediately deploys components on nodes as soon as decisions are made?
    - Wouldn't that be more efficient, & avoid "concurrent planning" issues?
- Race conditions between Planners & Executors are unavoidable, unless there is domain-wide locking or transactioning
  - e.g., the above would require backtracking upon conflict
- In D+C, planning decision making is an entirely local process
  - Interacting with nodes incurs large latency
  - Not interacting with nodes is better tradeoff
- Also, Deployment Plan is an important inter-vendor boundary!

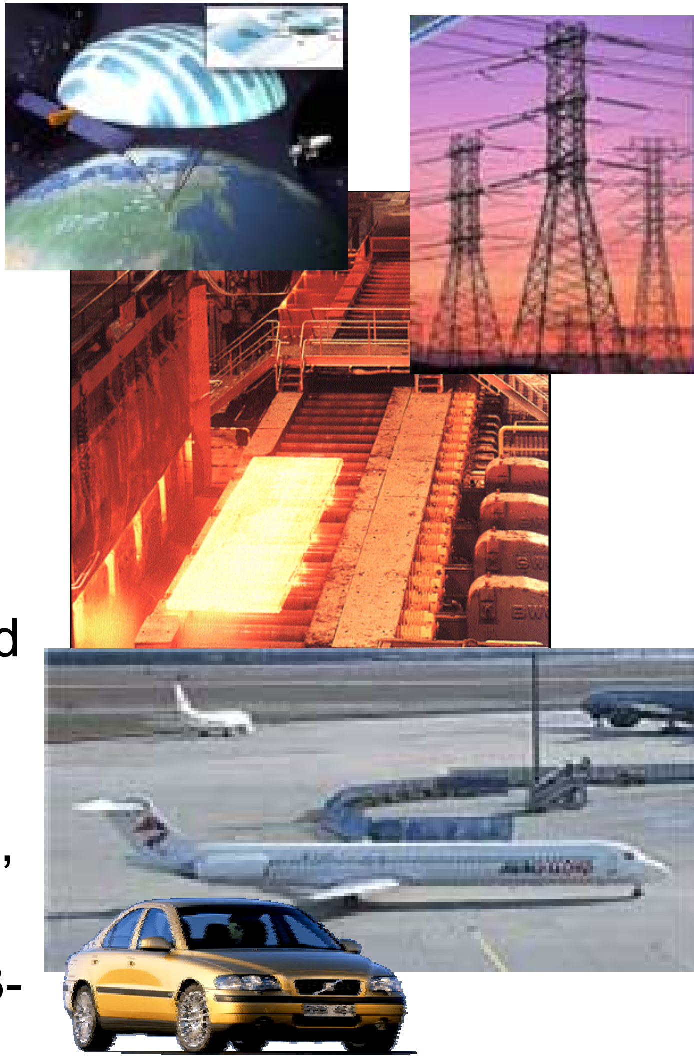
Summary of Deployment & Configuration Spec

- Powerful concepts for the deployment of component-based applications
  - Evolution of the original CCM's packaging
    - Hierarchical assemblies, allowing better component reuse
    - Resource management
    - Automated distribution & deployment
- Well-defined inter-vendor boundaries
  - Planner and Repository, Target, Execution, & Node Managers can be replaced separately
- Designed for distributed, real-time, & embedded systems
  - But also useful for general-purpose distributed component systems

Overview of Lightweight CCM Specification

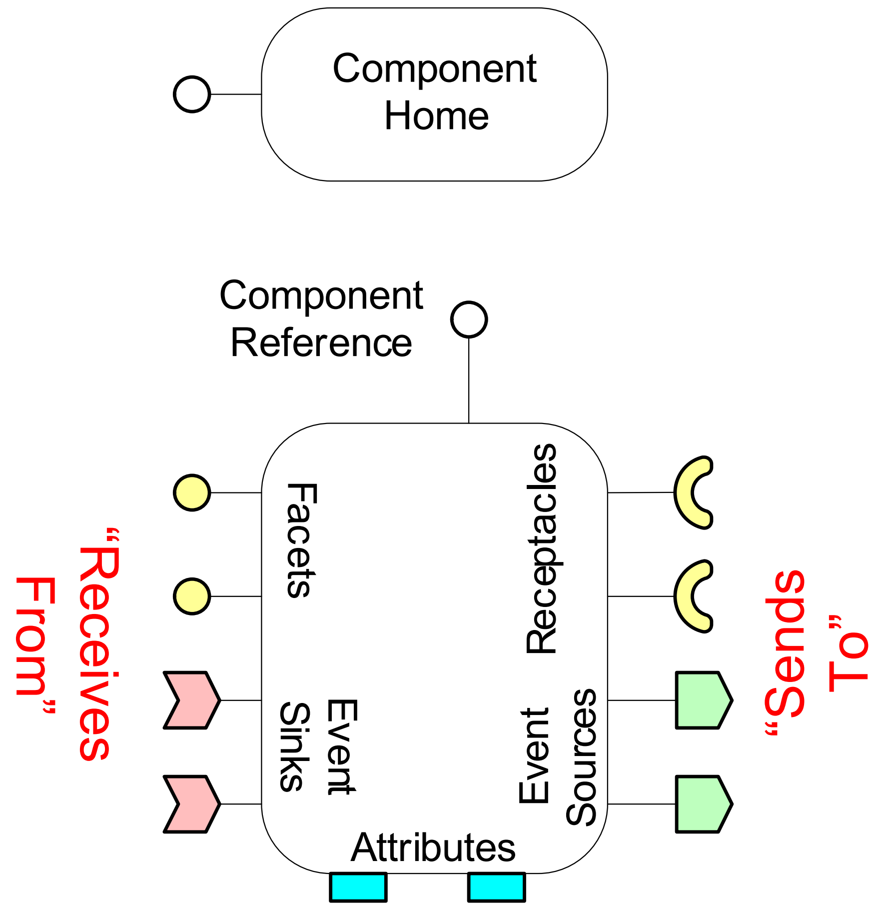
Motivation for Lightweight CCM

- Many DRE CORBA applications can't use enterprise CCM due to design constraints imposed by their operational environment
  - e.g., small code size in embedded environments & limited processing overhead for performance-intensive applications
- These constrained environments need CCM functionality packaged as a "lightweight" version
- 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
  - http://www.omg.org/cgi-bin/doc?realtime/2003-05-05



CCM Features Retained in 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 components & containers



CCM Features Excluded from Subset

- Keyed homes
  - Large overhead & complexity
- Process & Entity container
  - Persistence often not relevant in DRE systems domain
- Component segmentation
  - Unnecessary with introduction of D+C
- CIDL
  - Not needed after removal of PSDL & segmentation
  - IDL 3 is 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 not relevant in DRE systems domain
- Equivalent IDL for port management
  - Redundant, can use generic operations
  - Generic interface is required for D+C

Wrapping Up

Tutorial Summary

- **CCM**
  - 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 runtime environment with Containers & Component Servers
  - Specifies packaging & deployment framework
- **Deployment & Configuration** specification separates key configuration concerns
  - Server configuration
  - Object/service configuration
  - Application configuration
  - Object/service deployment

Additional Information

- OMG specifications pertaining to CCM
  - CORBA Component Model (CCM)
    - formal/2002-06-65
  - Lightweight CCM
    - realtime/03-05-05
  - QoS for CCM RFP
    - mars/03-06-12
  - Streams for CCM RFP
    - mars/03-06-11
  - UML Profile for CCM
    - mars/03-05-09
  - Deployment & Configuration (D+C)
    - ptc/03-06-03
- Books pertaining to CCM
  - *CORBA 3 Fundamentals and Programming*, Dr. John Siegel, published at John Wiley & Sons
- Web resources
  - “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