# Tutorial on the Lightweight CORBA Component Model (CCM)

## Industrializing the Development of Distributed Real-time & Embedded Applications

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

Professor of EECS Vanderbilt
University Nashville, Tennessee

*Other contributors include Kitty Balasubramanian, Tao Lu, Bala Natarajan, Jeff Parsons, Frank Pilhofer, Craig Rodrigues, & Nanbor Wang*

# Tutorial Overview

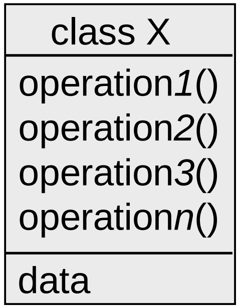
- The purpose of this tutorial is to
  - Motivate the need for the CORBA Component Model (CCM) & contrast it with the CORBA 2.x distributed object computing (DOC) model
  - Introduce CCM features most relevant to distributed real-time & embedded (DRE) applications
    - e.g., Lightweight CCM & the new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

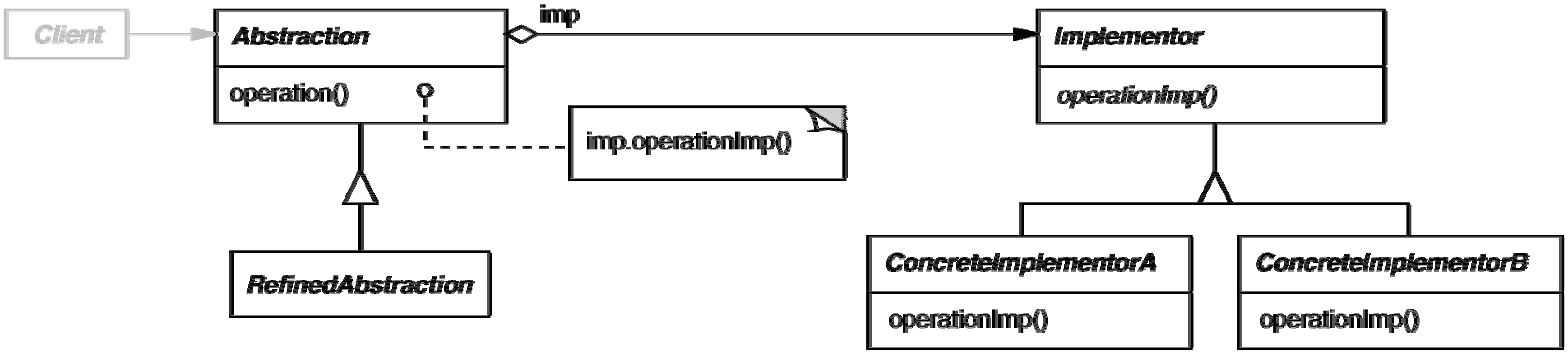
# Motivation & Overview of Component Middleware

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

# Where We Started: Object-Oriented Programming

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

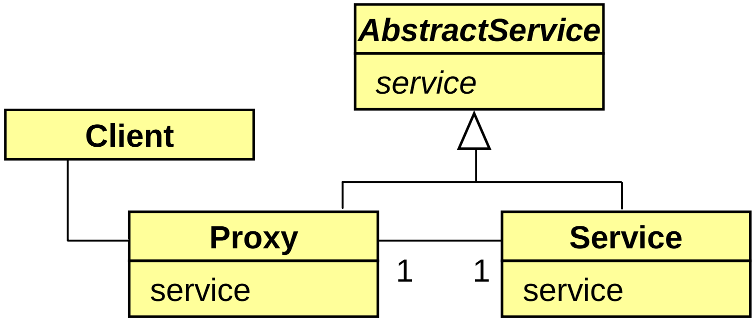


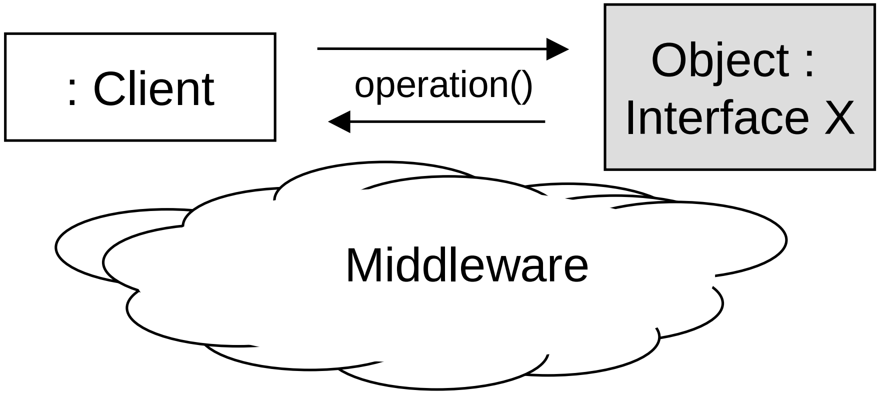


**Well-written OO programs exhibit recurring structures that promote abstraction, flexibility, modularity, & elegance**

# Next Step: Distributed Object Computing (DOC)

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

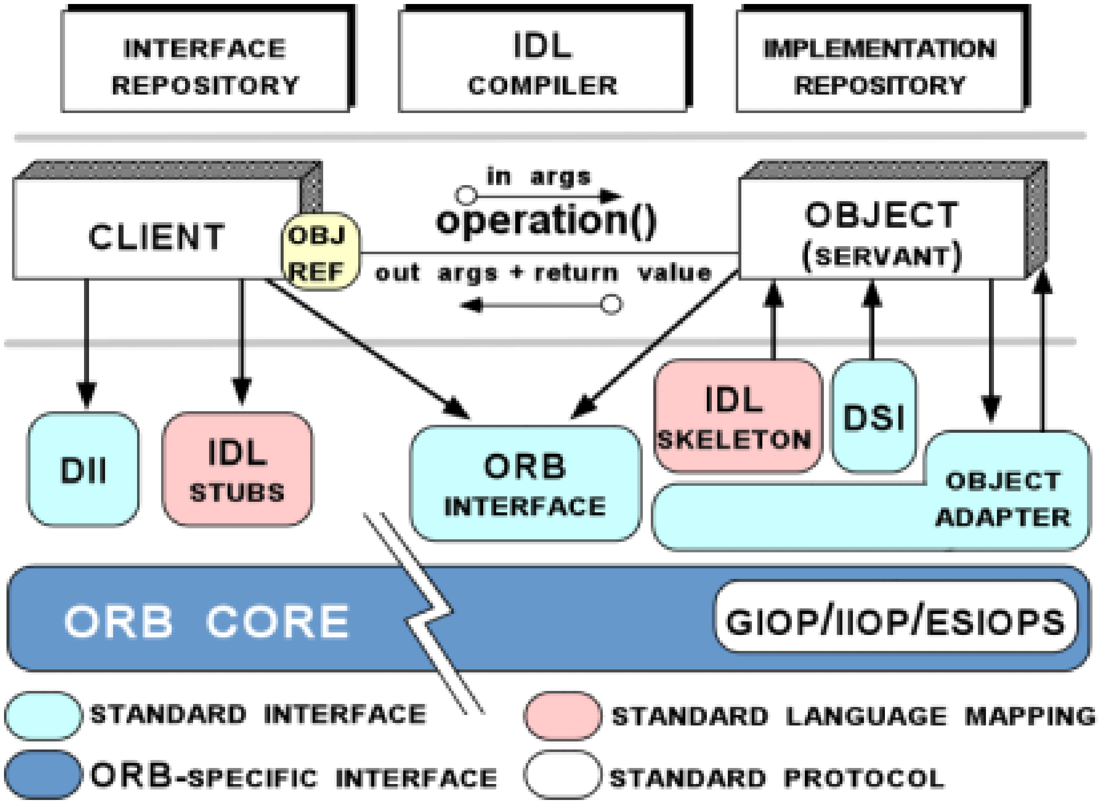




**We now have more robust software & more powerful distributed systems**

# Overview of CORBA 2.x Standard

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

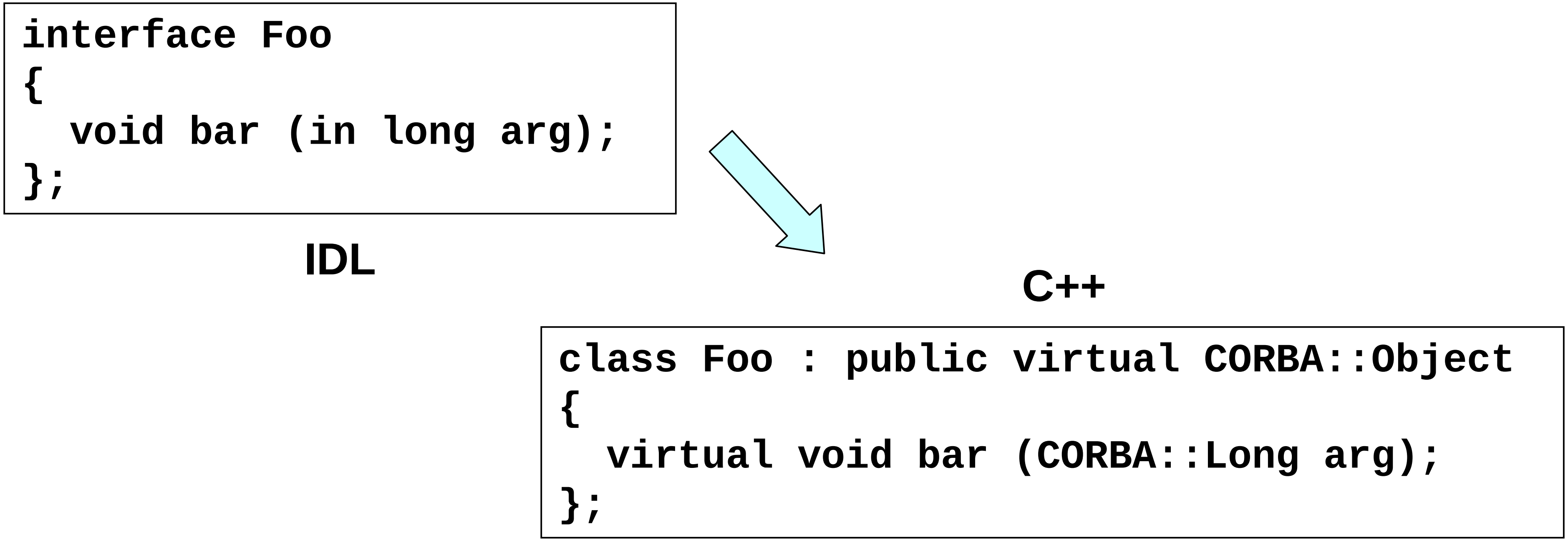


- CORBA 2.x automates
  - Object location
  - Connection & memory mgmt.
  - Parameter (de)marshaling
  - Event & request demultiplexing
  - Error handling & fault tolerance
  - Object/server activation
  - Concurrency & synchronization
  - Security

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

# Example: Applying OO to Network Programming

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



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

**IDL**

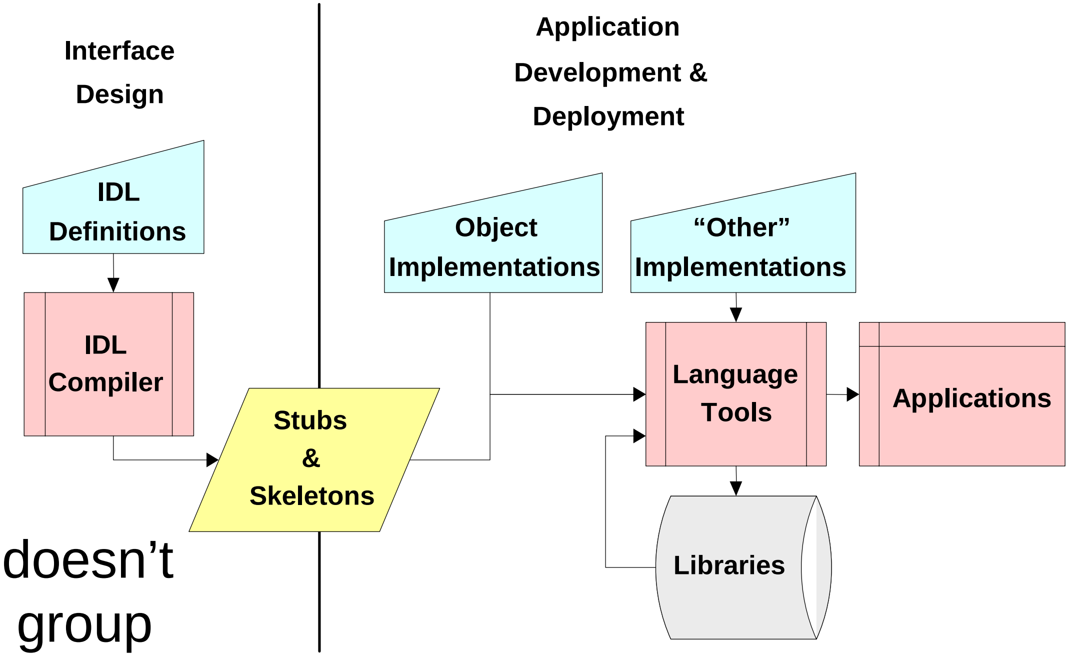
**C++**

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

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

# Drawbacks of DOC-based CORBA 2.x Middleware

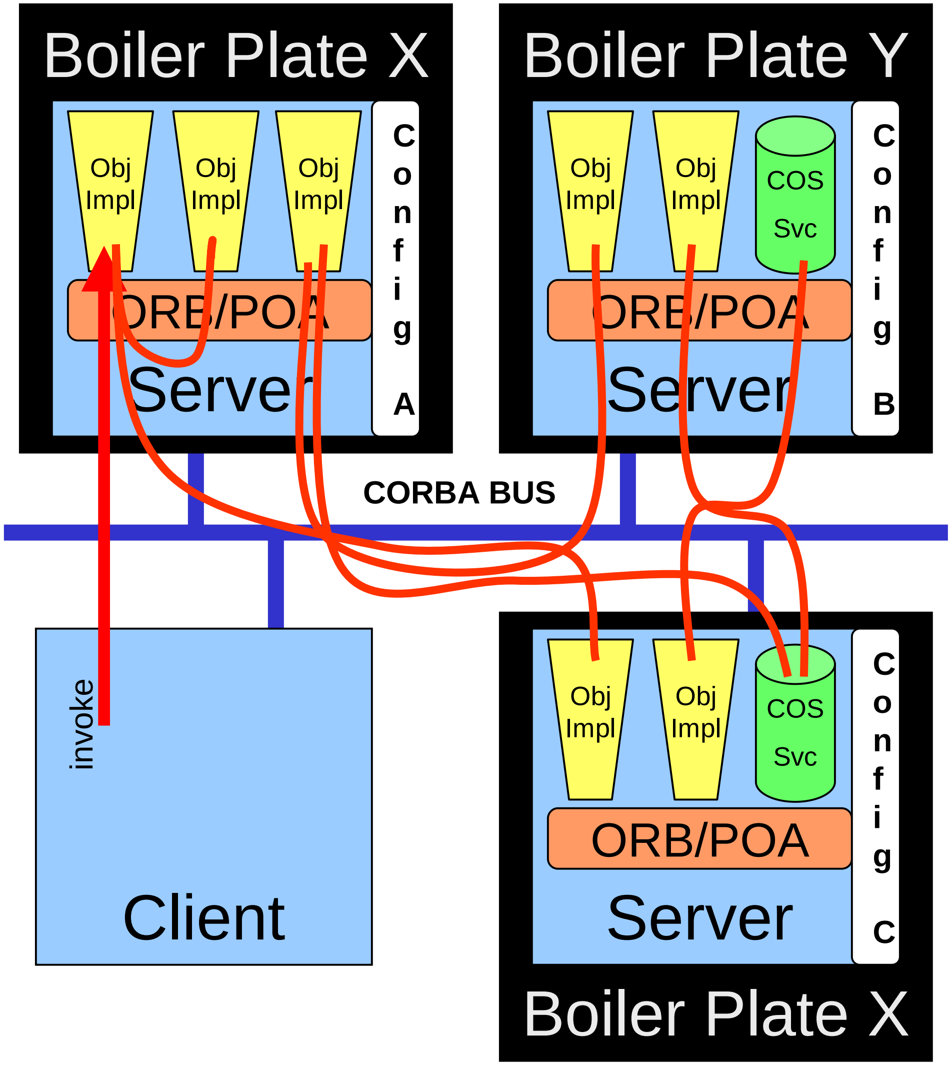
CORBA 2.x application development is unnecessarily tedious & error-prone



- CORBA 2.x IDL doesn't provide a way to group together related interfaces to offer a service family
  - Such "bundling" must be done by developers via CORBA idioms & patterns

- CORBA 2.x doesn't specify how configuration & deployment of objects should be done to create complete applications
  - Proprietary infrastructure & scripts are written by developers to enable this

# Example: Limitations of CORBA 2.x Specification

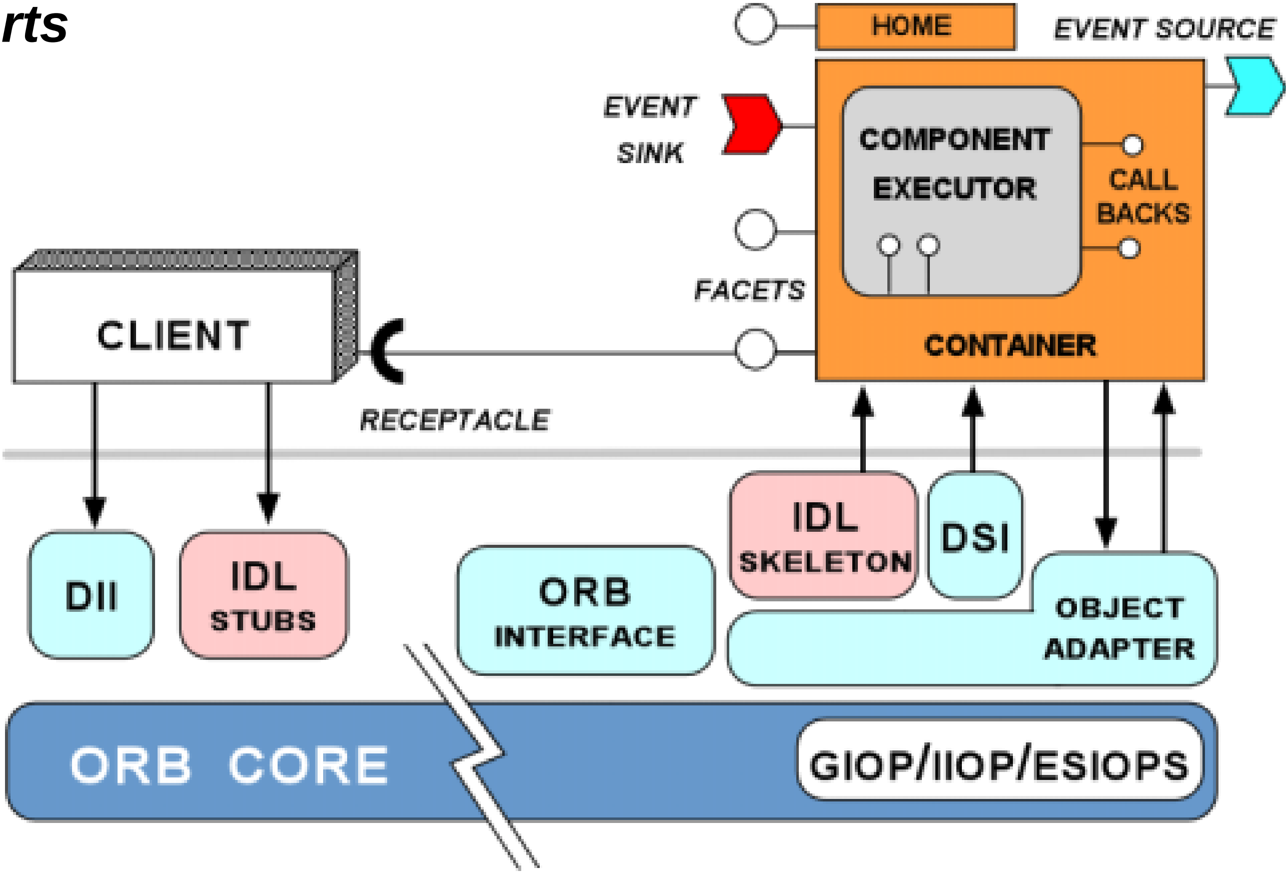


- Requirements of non-trivial DRE systems:
  - Collaboration of multiple objects & services
  - Deployment on diverse platforms
- CORBA 2.x limitations – lack of ***standards*** for
  - Server/node configuration
  - Object/service configuration
  - Application assembly
  - Object/service deployment
-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

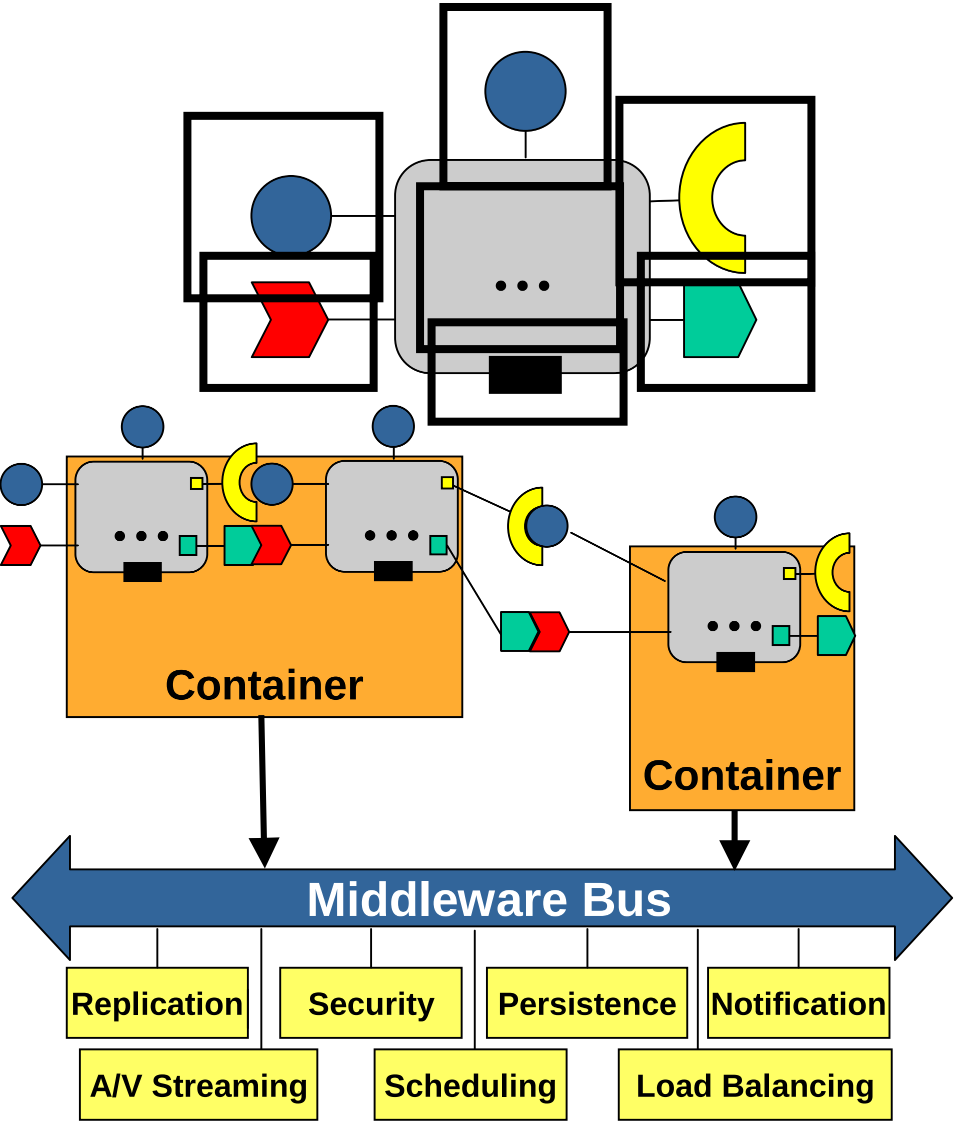
# Solution: Component Middleware

**Component middleware capabilities:**

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



# Birdseye View of Component Middleware

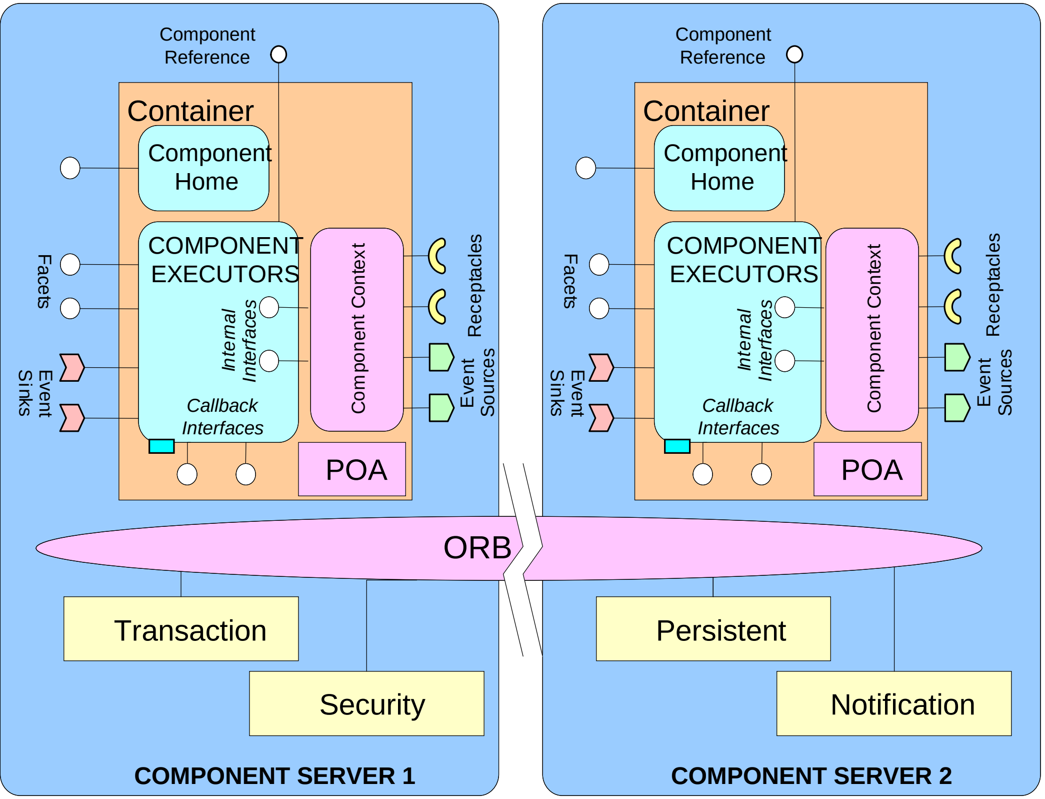


- *Components* encapsulate application “business” logic
- Components interact via *ports*
  - *Provided interfaces*, e.g.,facets
  - *Required connection points*, e.g., receptacles
  - *Event sinks & sources*
  - *Attributes*
- *Containers* provide execution environment for components with common operating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

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

# Overview of the CORBA Component Model (CCM)

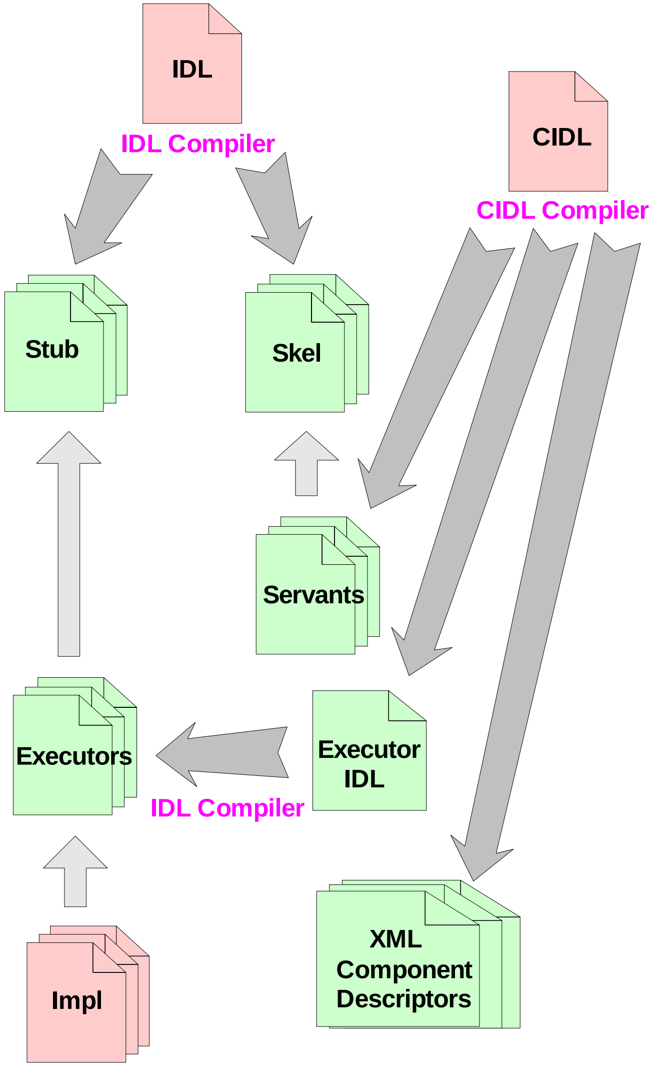
# Capabilities of the CORBA Component Model (CCM)



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

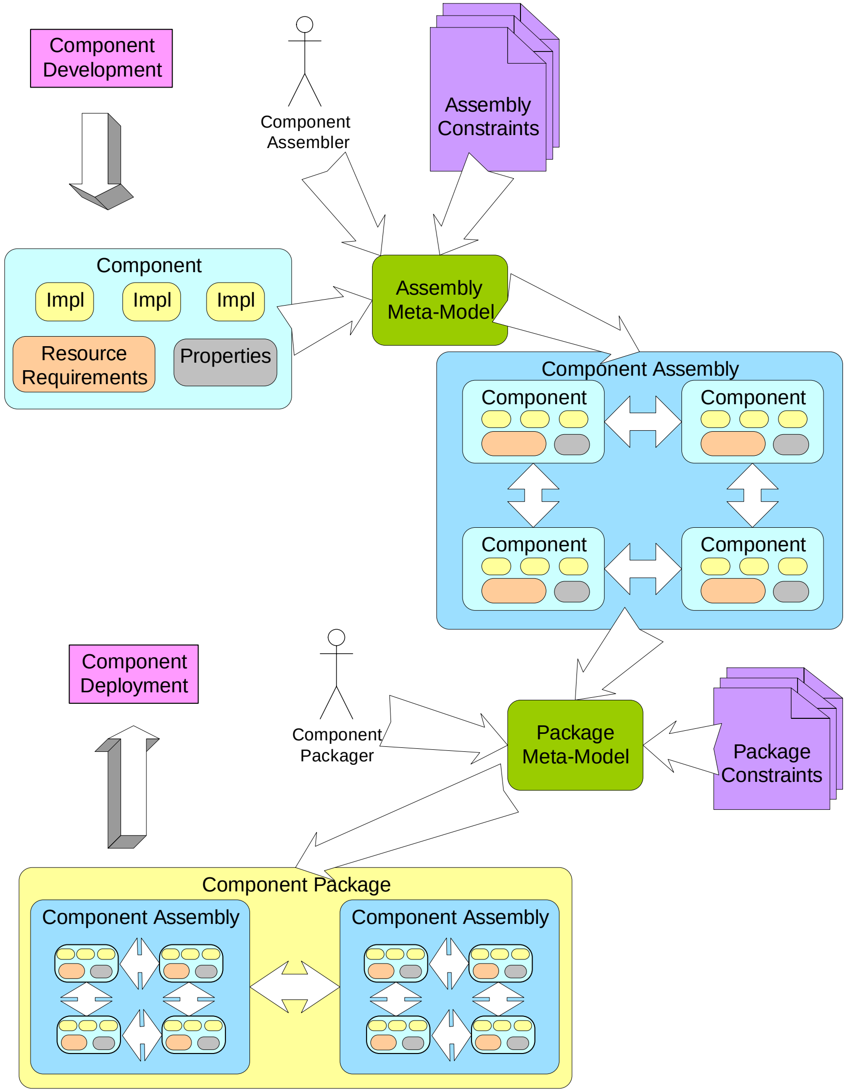
- Containers define operations that enable component executors to access common middleware services & runtime policies

# Capabilities of the CORBA Component Model (CCM)



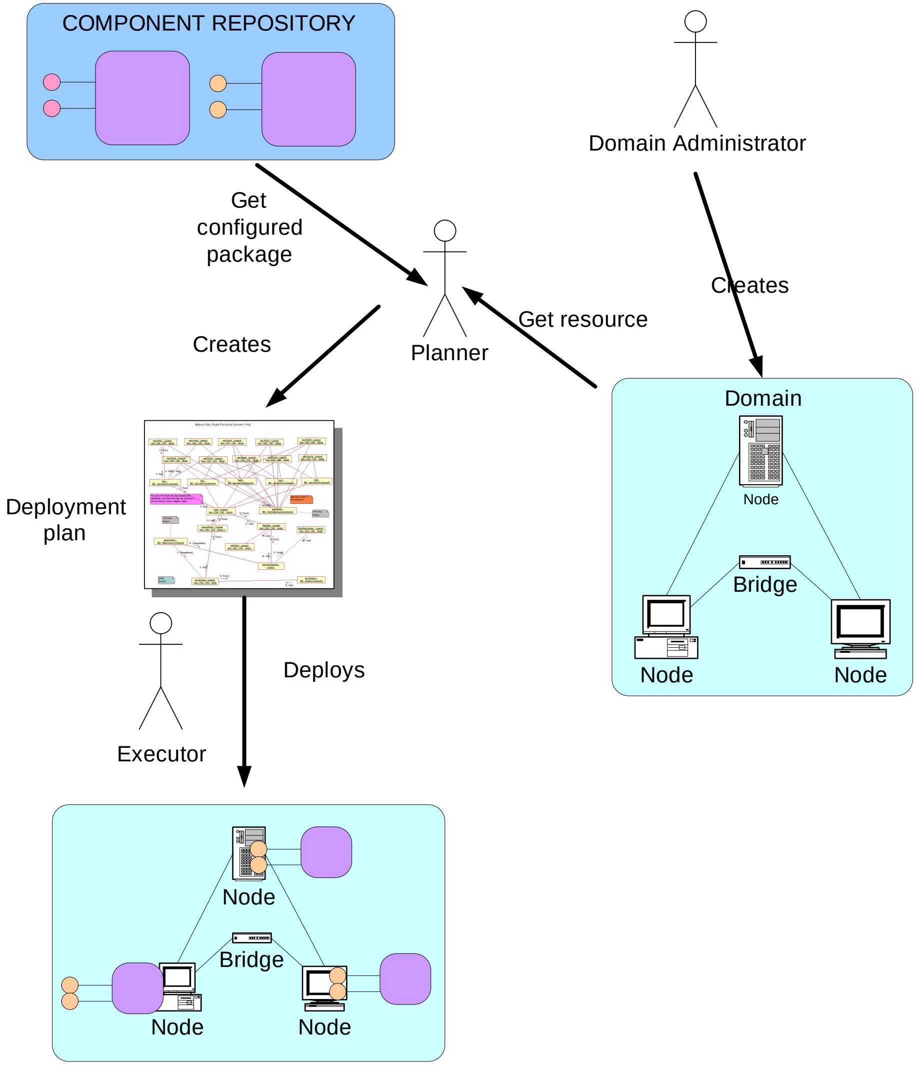
- Component Server
  - A generic server process for hosting containers & component/home executors
- **Component Implementation Framework (CIF)**
  - Automates the implementation of many component features
- Component packaging tools
  - Compose implementation & configuration information into deployable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

# Capabilities of CORBA Component Model (CCM)



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

# Capabilities of CORBA Component Model (CCM)



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

# Available CCM Implementations

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

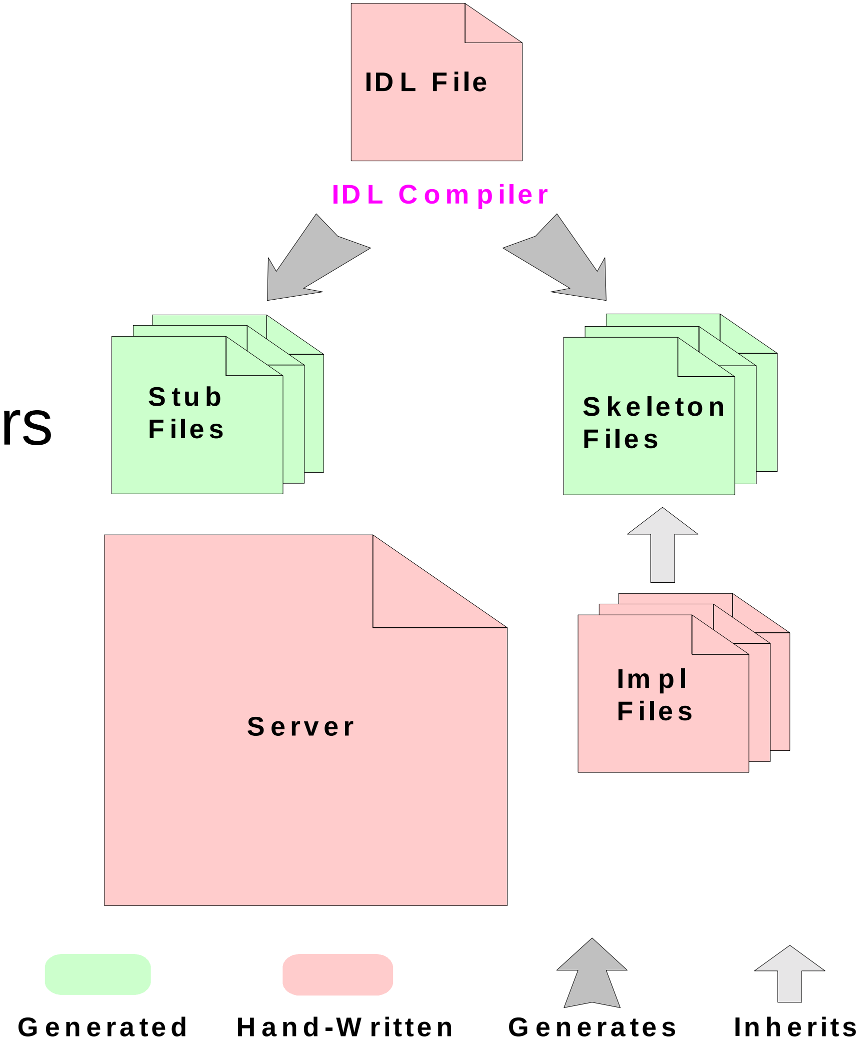
# CCM Compared to EJB, COM, & .NET

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

# Comparing Application Development with CORBA 2.x vs. CCM

# CORBA 2.x User Roles

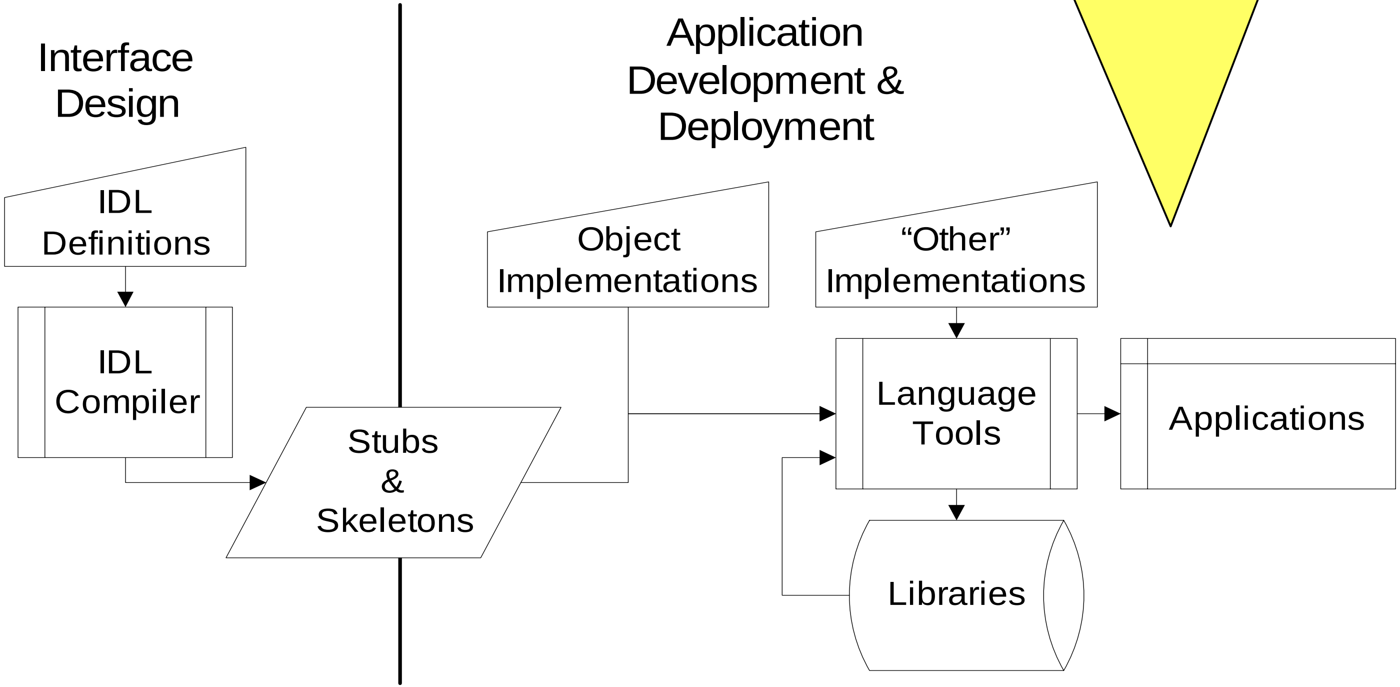
- Object interface designers
- Server developers
- Client application developers



# CORBA 2.x Application Development Lifecycle

Specification of IDL interfaces of objects

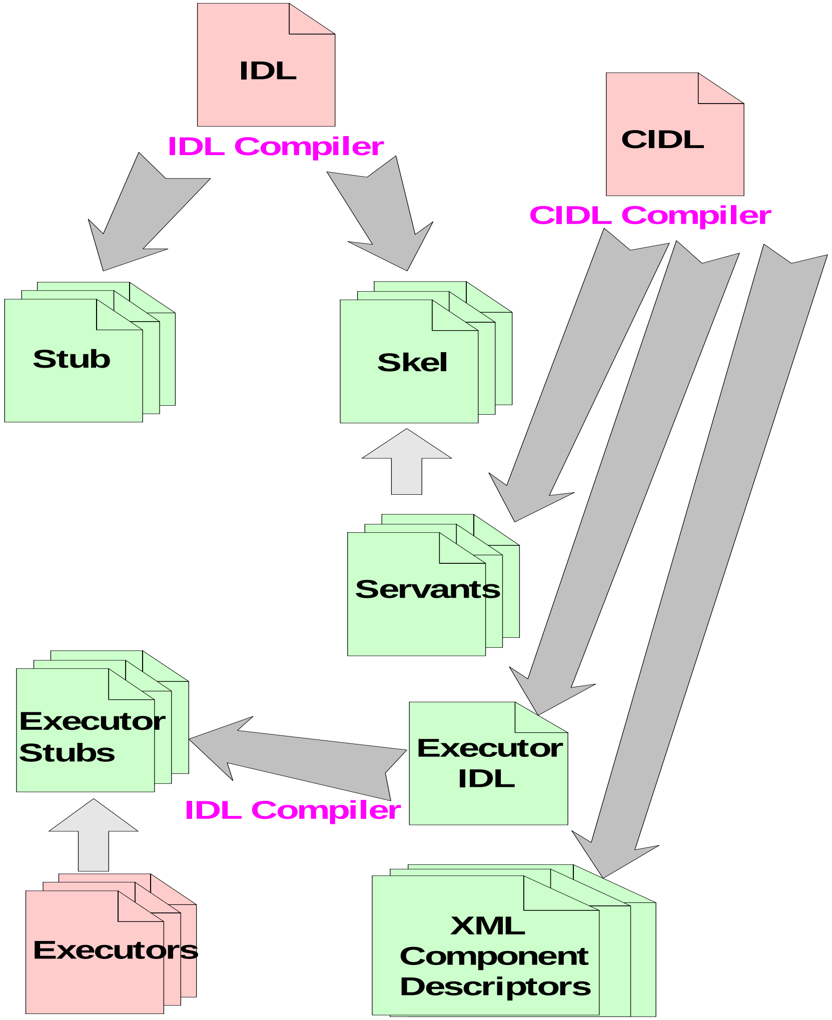
Implement servants & write ***all*** the code required to bootstrap & run the server



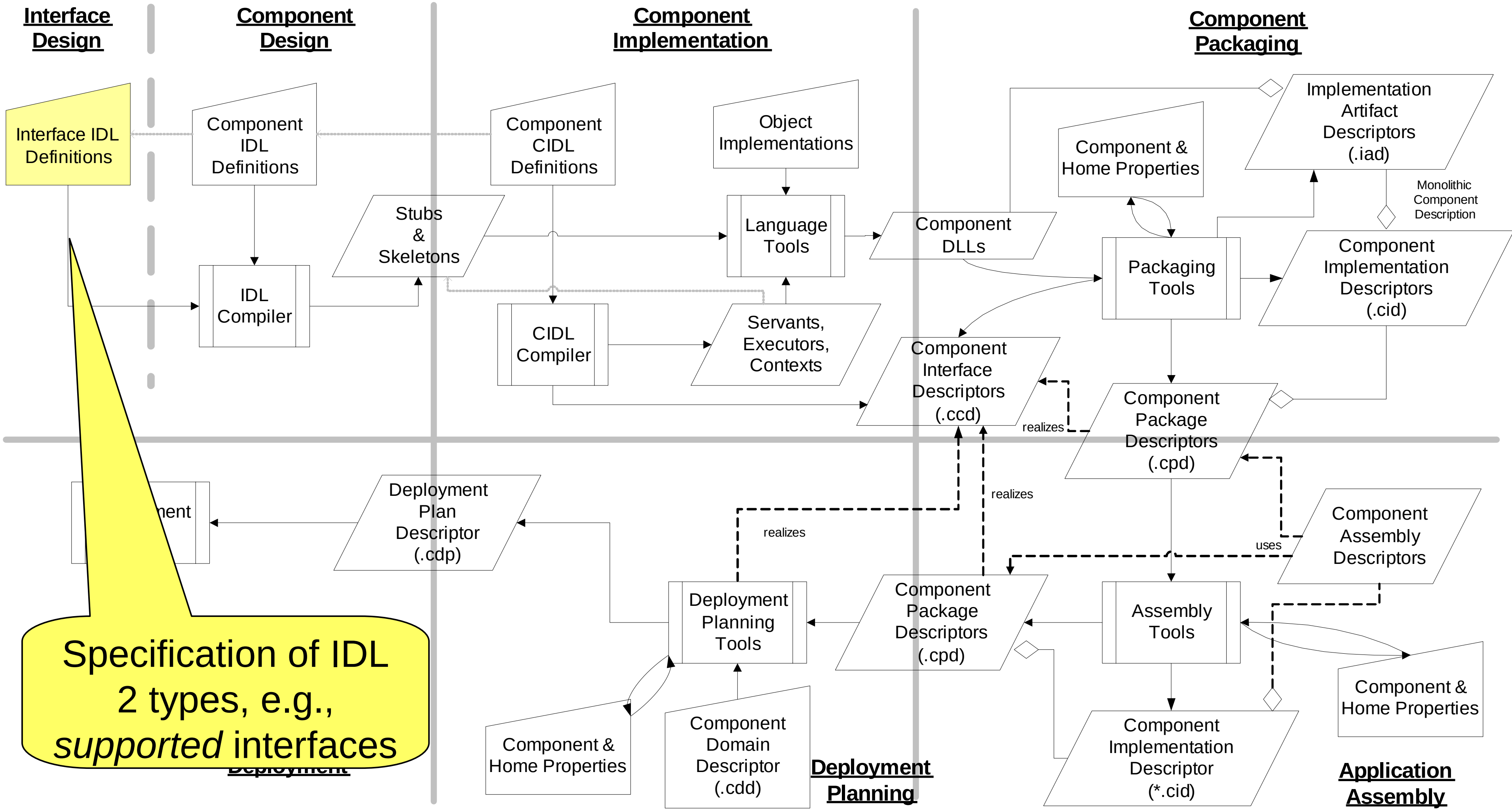
CORBA 2.x supports programming by development (engineering) rather than programming by assembly (manufacturing)

# CCM User Roles

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

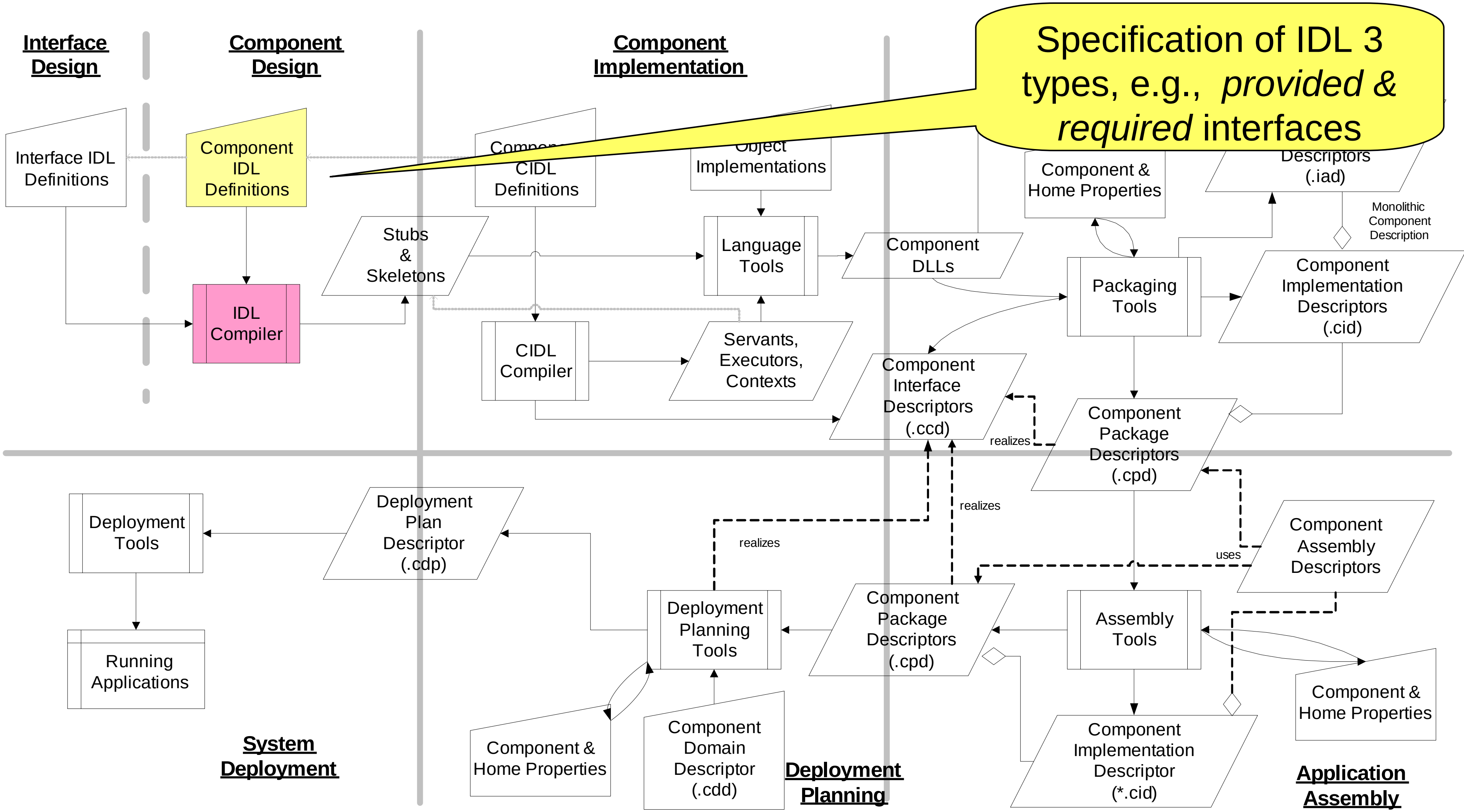


# CCM Application Development Lifecycle


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Component & Home Properties
Implementation Artifact Descriptors (.iad)
Monolithic Component Description
Stubs & Skeletons
Language Tools
Component DLLs
Packaging Tools
Component Implementation Descriptors (.cid)
IDL Compiler
CIDL Compiler
Servants, Executors, Contexts
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
realizes
realizes
uses
Component Assembly Descriptors
nent
Deployment Plan Descriptor (.cdp)
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
Component & Home Properties
Deployment
Deployment Planning
Application Assembly

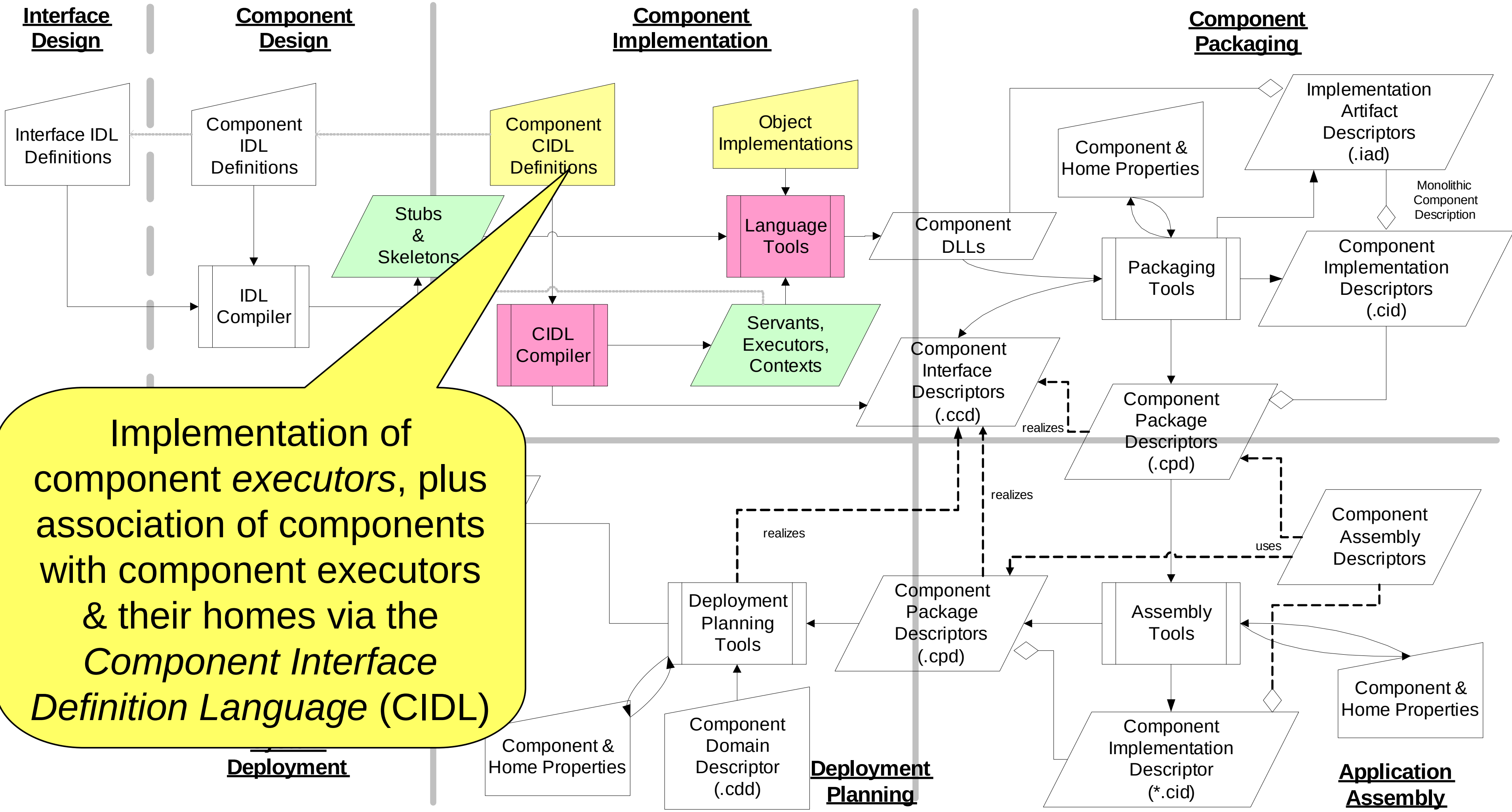
Specification of IDL 2 types, e.g., *supported* interfaces

# CCM Application Development Lifecycle


Interface Design
Component Design
Component Implementation
Interface IDL Definitions
Component IDL Definitions
Compon...
CIDL Definitions
Object Implementations
Component & Home Properties
Descriptors (.iad)
IDL Compiler
Stubs & Skeletons
Language Tools
Component DLLs
Monolithic Component Description
Packaging Tools
Component Implementation Descriptors (.cid)
CIDL Compiler
Servants, Executors, Contexts
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
Deployment Tools
Deployment Plan Descriptor (.cdp)
realizes
realizes
uses
Component Assembly Descriptors
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Running Applications
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
Component & Home Properties
System Deployment
Deployment Planning
Application Assembly

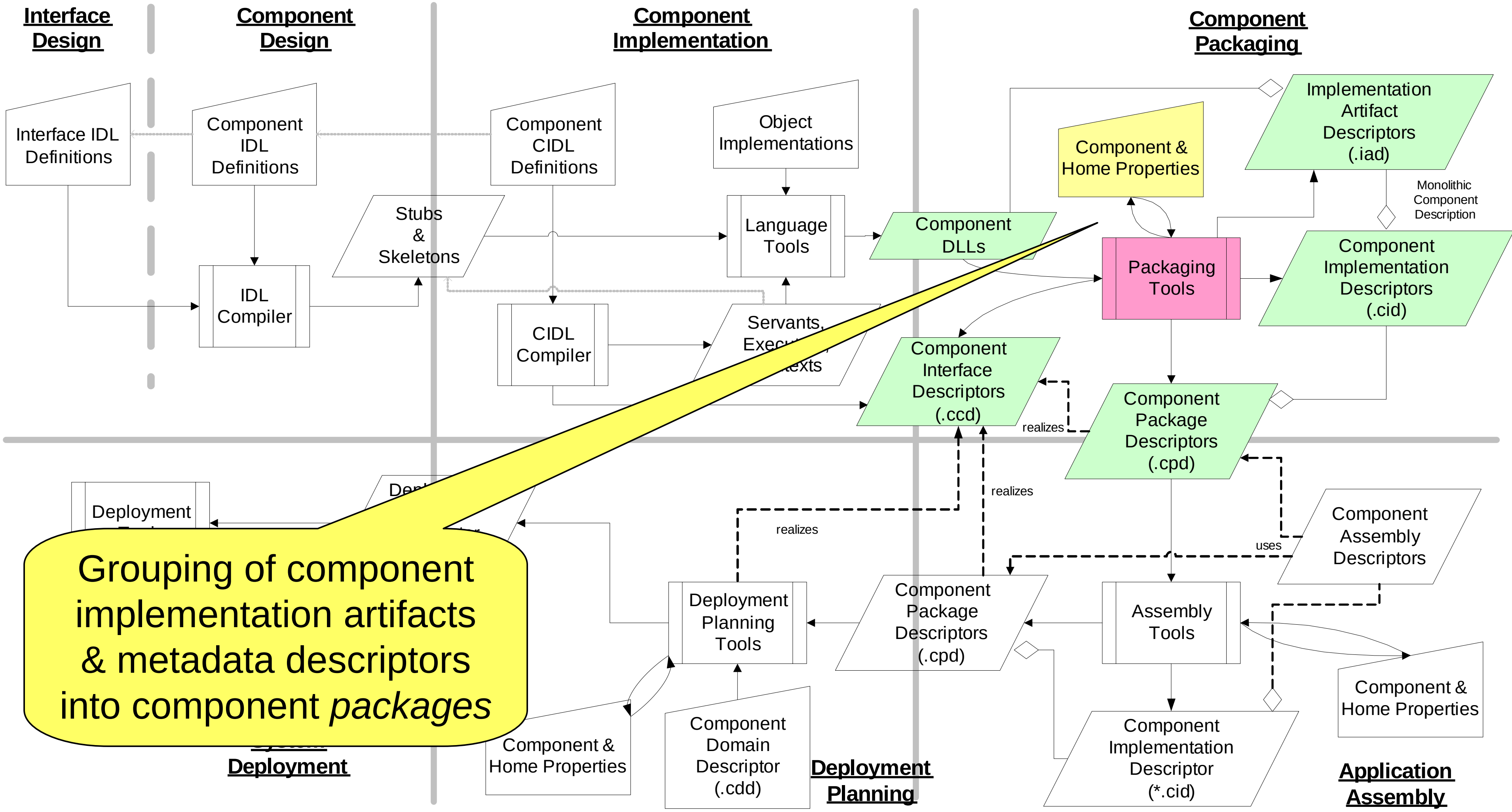
Specification of IDL 3 types, e.g., *provided & required* interfaces

# CCM Application Development Lifecycle


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Stubs & Skeletons
Language Tools
IDL Compiler
CIDL Compiler
Servants, Executors, Contexts
Component DLLs
Component & Home Properties
Implementation Artifact Descriptors (.iad)
Monolithic Component Description
Packaging Tools
Component Implementation Descriptors (.cid)
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
realizes
realizes
uses
Component Assembly Descriptors
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
Component & Home Properties
Deployment
Deployment Planning
Application Assembly

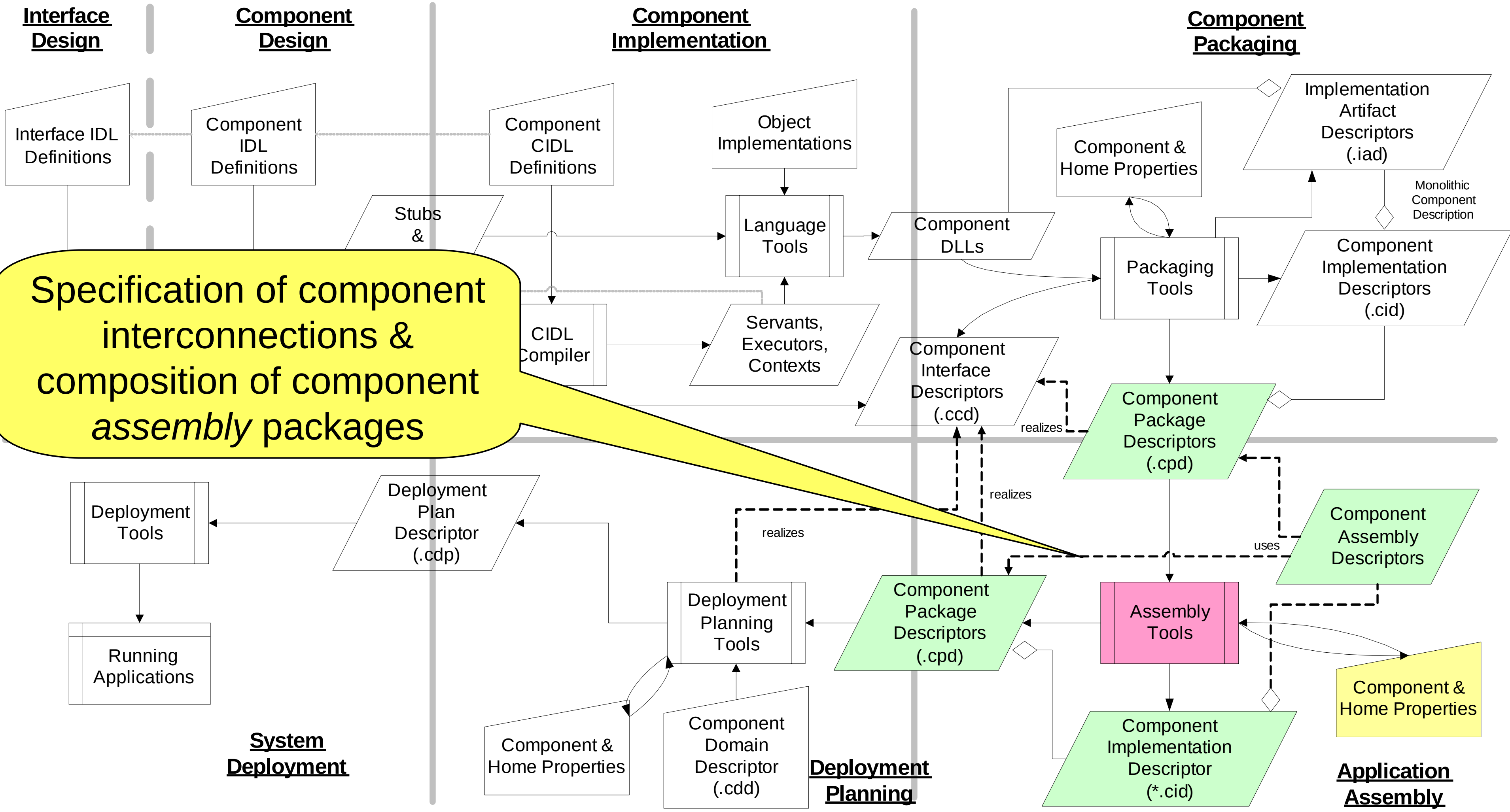
Implementation of component *executors*, plus association of components with component executors & their homes via the *Component Interface Definition Language* (CIDL)

# CCM Application Development Lifecycle


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Implementation Artifact Descriptors (.iad)
Component & Home Properties
Stubs & Skeletons
Language Tools
Component DLLs
Monolithic Component Description
IDL Compiler
Packaging Tools
Component Implementation Descriptors (.cid)
CIDL Compiler
Servants, Exec exts
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
Deployment
Dep
realizes
realizes
uses
Component Assembly Descriptors
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component & Home Properties
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
Deployment
Deployment Planning
Application Assembly

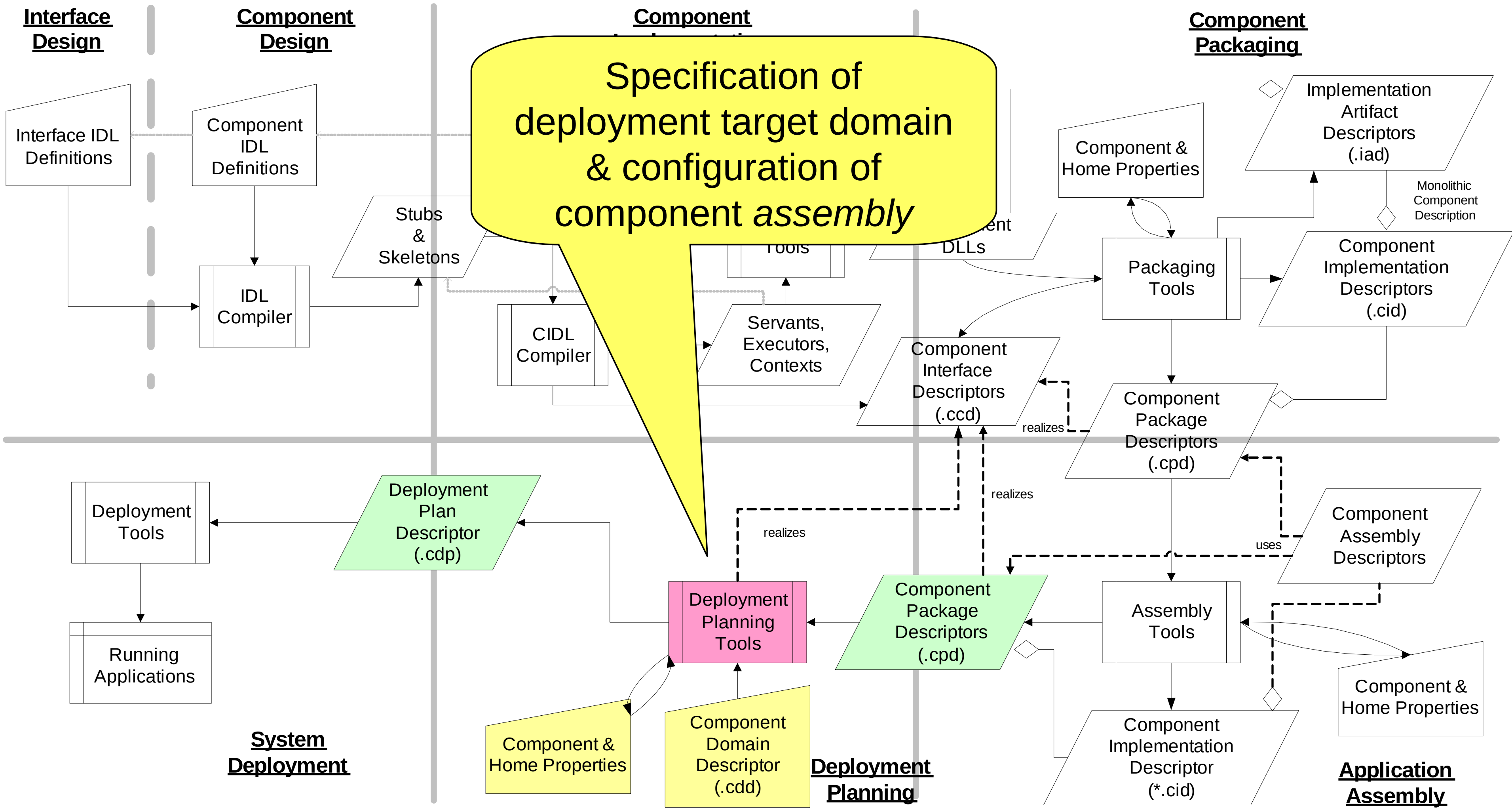
Grouping of component implementation artifacts & metadata descriptors into component *packages*

# CCM Application Development Lifecycle


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Stubs &
Language Tools
Component DLLs
Component & Home Properties
Implementation Artifact Descriptors (.iad)
Monolithic Component Description
Packaging Tools
Component Implementation Descriptors (.cid)
CIDL Compiler
Servants, Executors, Contexts
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
Deployment Tools
Deployment Plan Descriptor (.cdp)
realizes
realizes
uses
Component Assembly Descriptors
Running Applications
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component & Home Properties
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
System Deployment
Deployment Planning
Application Assembly

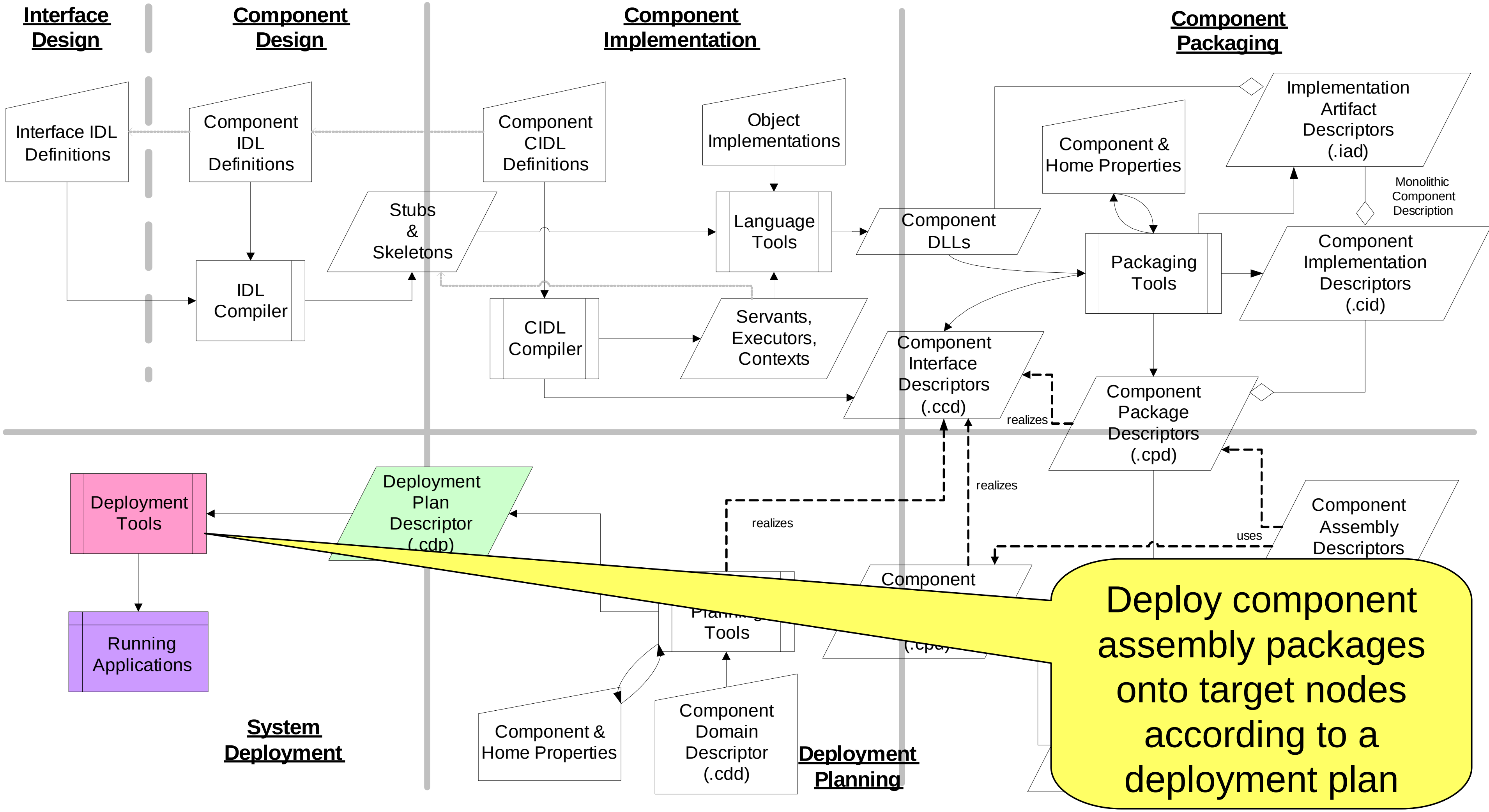
Specification of component interconnections & composition of component *assembly* packages

# CCM Application Development Lifecycle


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
IDL Compiler
Stubs & Skeletons
CIDL Compiler
Servants, Executors, Contexts
Tools
Component DLLs
Component & Home Properties
Implementation Artifact Descriptors (.iad)
Monolithic Component Description
Packaging Tools
Component Implementation Descriptors (.cid)
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
realizes
Deployment Tools
Deployment Plan Descriptor (.cdp)
realizes
Component Assembly Descriptors
uses
Running Applications
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component & Home Properties
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
System Deployment
Deployment Planning
Application Assembly

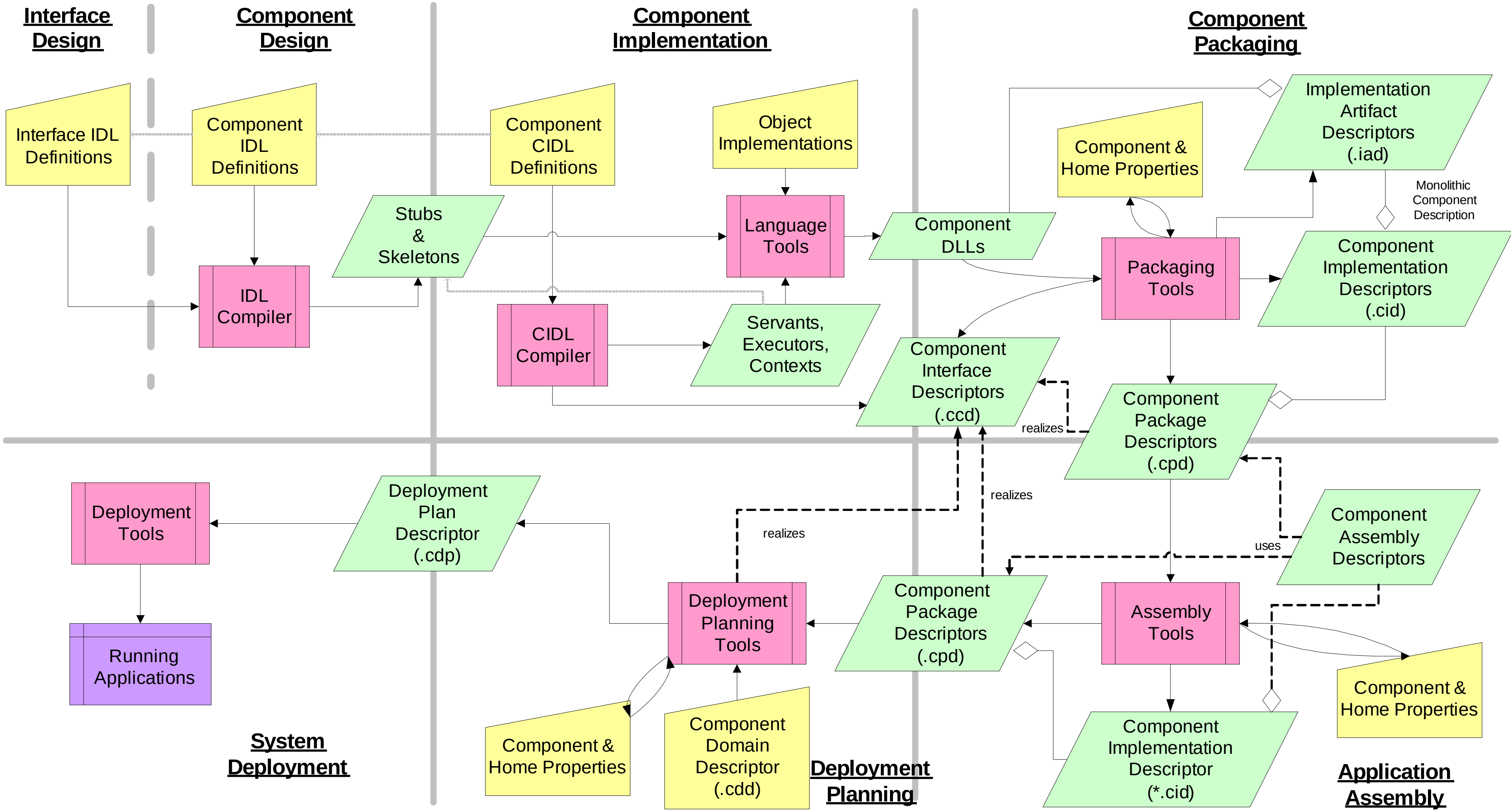
Specification of deployment target domain & configuration of component *assembly*

# CCM Application Development Lifecycle


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Stubs & Skeletons
Language Tools
IDL Compiler
CIDL Compiler
Servants, Executors, Contexts
Component DLLs
Component & Home Properties
Implementation Artifact Descriptors (.iad)
Monolithic Component Description
Packaging Tools
Component Implementation Descriptors (.cid)
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
realizes
realizes
uses
Component Assembly Descriptors
Deployment Tools
Deployment Plan Descriptor (.cdp)
Running Applications
Component
(.cpa)
Tools
System Deployment
Component & Home Properties
Component Domain Descriptor (.cdd)
Deployment Planning

Deploy component assembly packages onto target nodes according to a deployment plan

# CCM Application Development Lifecycle


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Component & Home Properties
Implementation Artifact Descriptors (.iad)
Monolithic Component Description
Stubs & Skeletons
Language Tools
Component DLLs
Packaging Tools
Component Implementation Descriptors (.cid)
IDL Compiler
CIDL Compiler
Servants, Executors, Contexts
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
Deployment Tools
Deployment Plan Descriptor (.cdp)
realizes
realizes
Component Assembly Descriptors
uses
Running Applications
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component & Home Properties
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
System Deployment
Deployment Planning
Application Assembly

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

# CORBA Component Model (CCM) Features

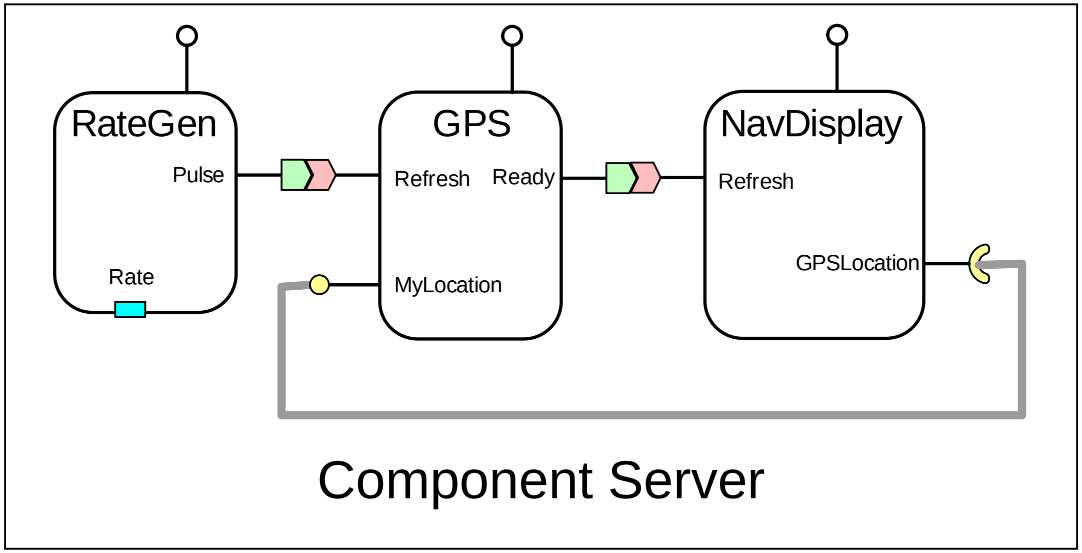
# Example CCM DRE Application

Avionics example used throughout tutorial as typical DRE application

Rate Generator

Positioning Sensor

Display Device

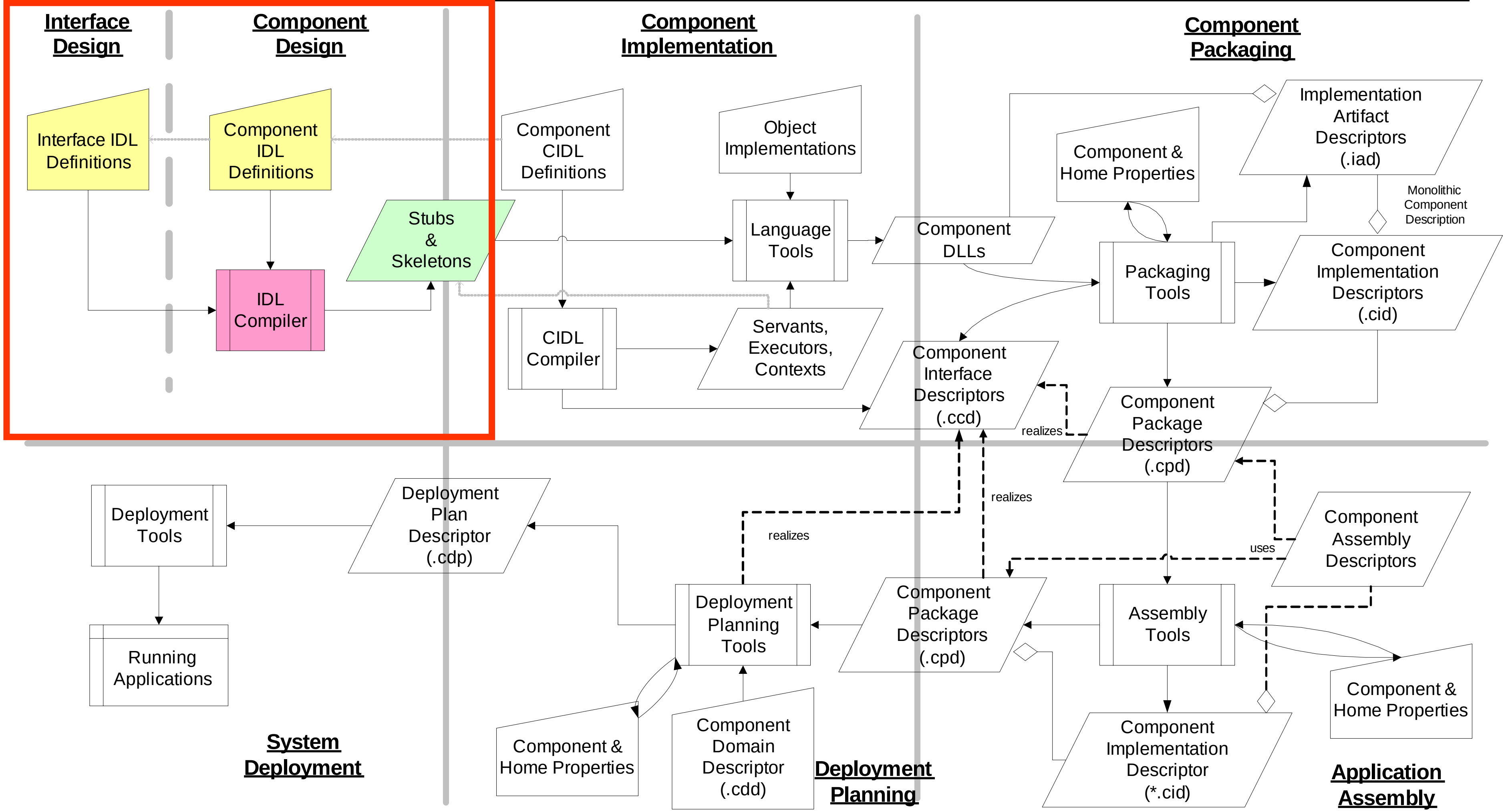


**`$CIAO_ROOT/examples/OEP/Display/`**

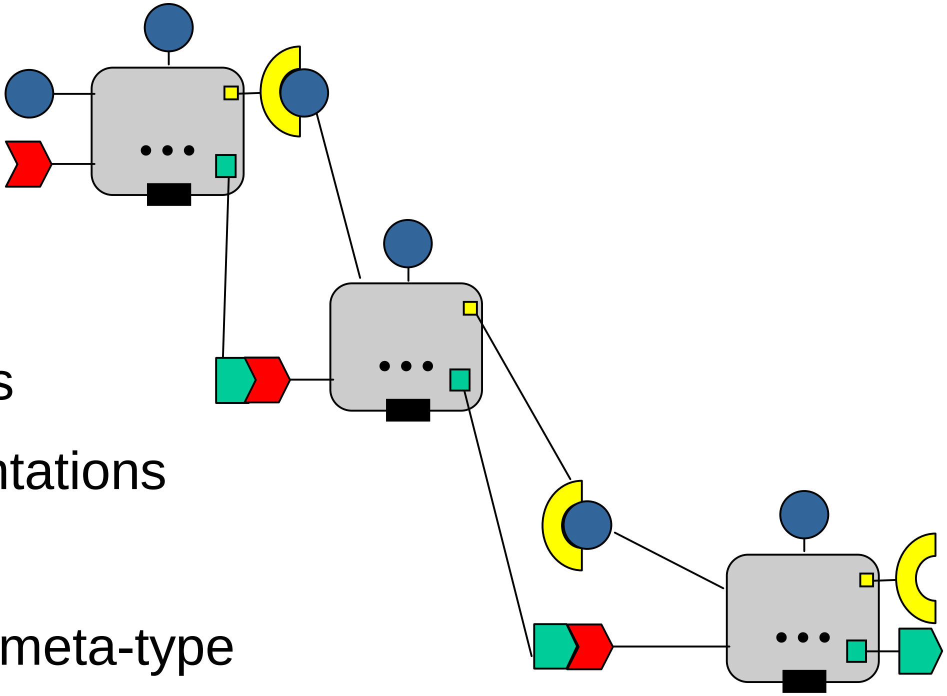
- ***Rate Generator***
  - Sends periodic **`Pulse`** events to consumers
- ***Positioning Sensor***
  - Receives **`Refresh`** events from suppliers
  - Refreshes cached coordinates available thru **`MyLocation`** facet
  - Notifies subscribers via **`Ready`** events
- ***Display Device***
  - Receives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


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Stubs & Skeletons
IDL Compiler
Language Tools
CIDL Compiler
Servants, Executors, Contexts
Component DLLs
Component & Home Properties
Implementation Artifact Descriptors (.iad)
Monolithic Component Description
Packaging Tools
Component Implementation Descriptors (.cid)
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
Deployment Tools
Deployment Plan Descriptor (.cdp)
realizes
realizes
uses
Component Assembly Descriptors
Running Applications
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component & Home Properties
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
System Deployment
Deployment Planning
Application Assembly

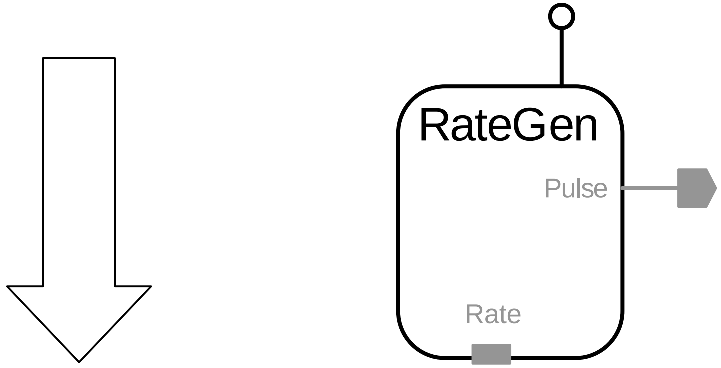
# Unit of Business Logic & Composition in CCM



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

# Simple CCM Component Example

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



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

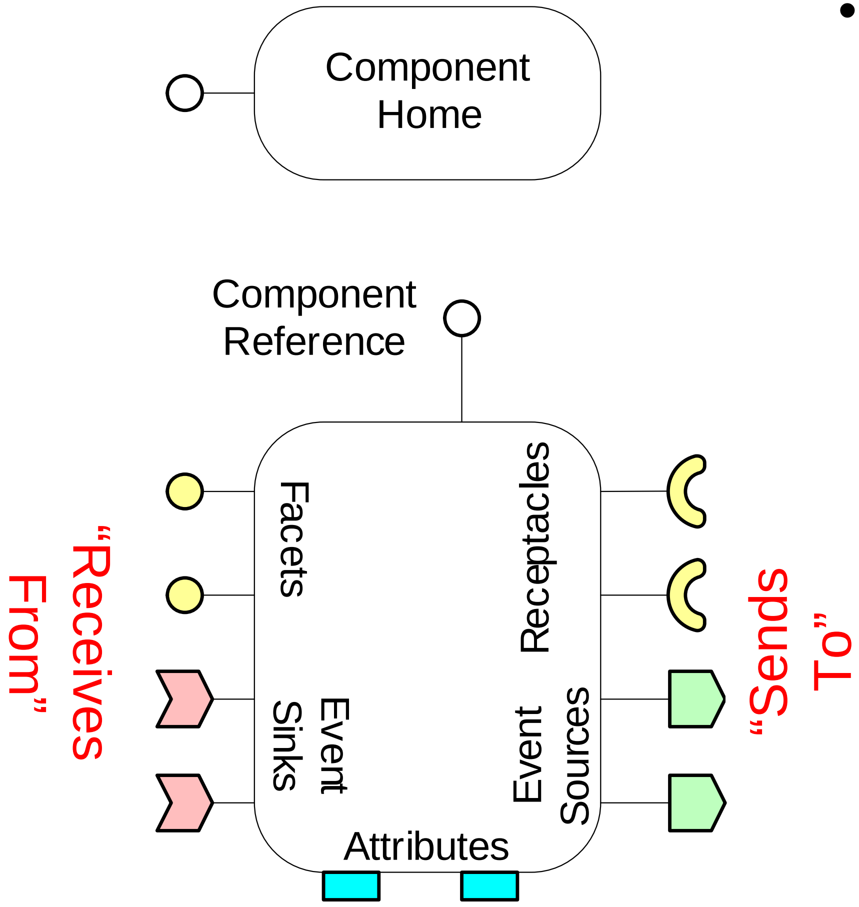
- Roles played by CCM **component**
  - Define a unit of reuse & implementation
  - Encapsulate an interaction & configuration model
- A CORBA **component** has several derivation options, i.e.,
  - It can *inherit* from a single component type

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

  - It can *support* multiple IDL interfaces

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

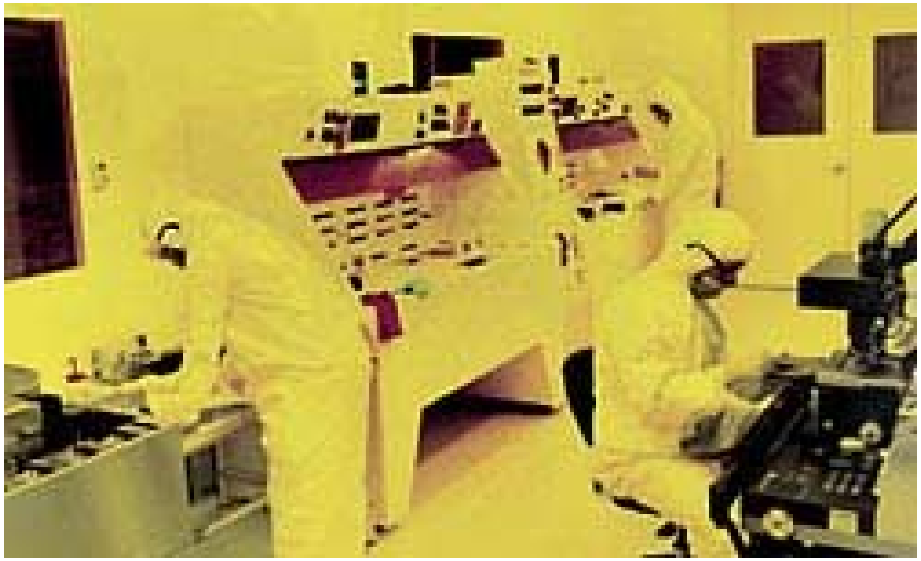
# CORBA Component Ports



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

# Managing Component Lifecycle

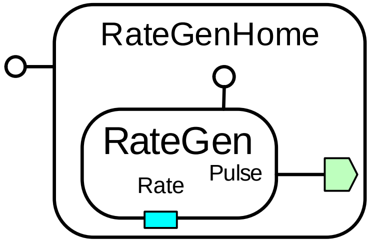
- Context
  - Components need to be created by the CCM run-time
- Problems with CORBA 2.x
  - No standard way to manage component's lifecycle
  - Need standard mechanisms to strategize lifecycle management



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

# A CORBA Component Home

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



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

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

# A Quick CCM Client Example

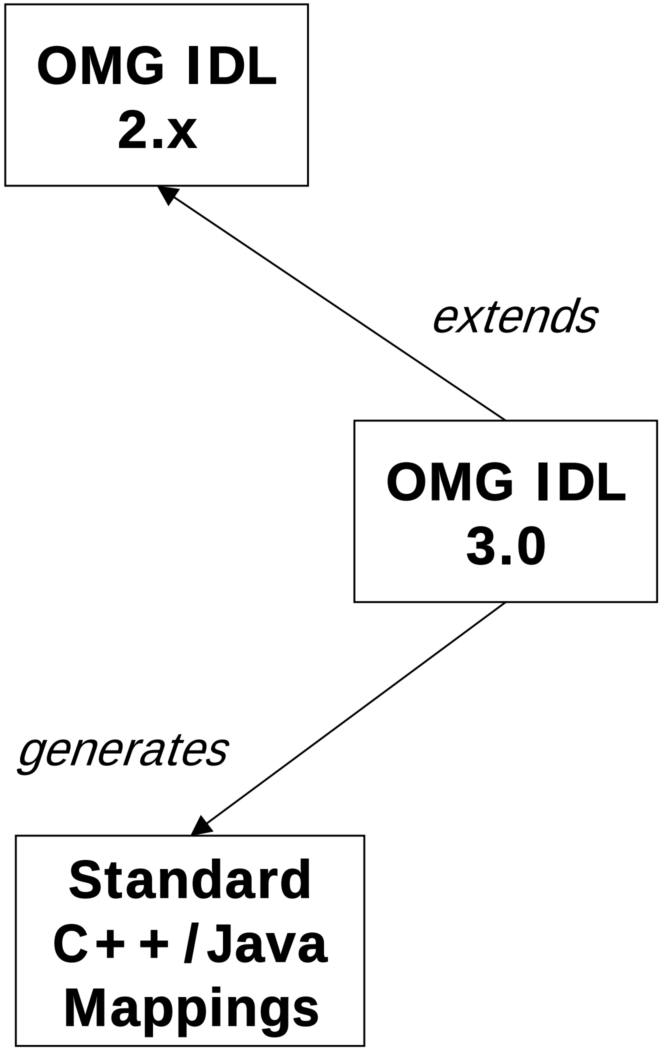
# Component & Home for Simple **HelloWorld**

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

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

# The Client OMG IDL Mapping

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



# Simple Client for **HelloWorld** Component

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

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

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

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

# CCM Component Features in Depth

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

# Components Can Offer Different Views

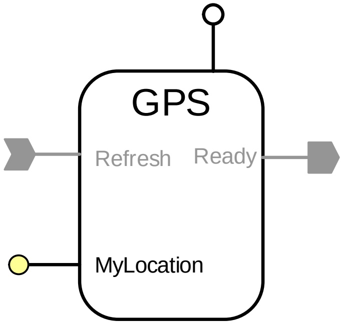
- Context
  - –Components need to collaborate with other types of components
  - –These collaborating components may understand different interfaces
- Problems with CORBA 2.x
  - –Hard to extend interface without breaking/bloating it
  - –No standard way to acquire new interfaces
- CCM Solution
  - –Define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

# Component Facets

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

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

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

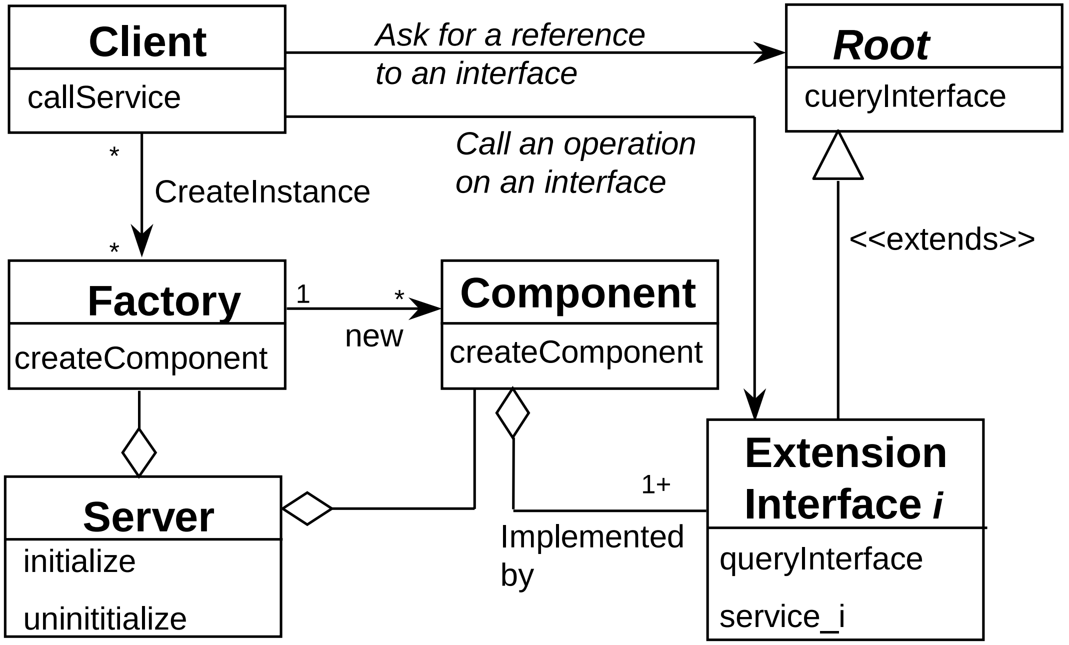


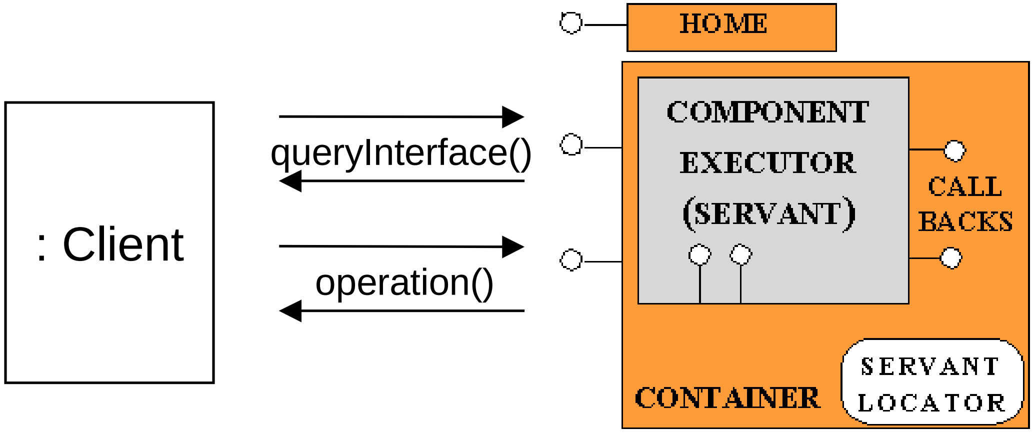
# Extension Interface Pattern

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

- breaking of client code &
- bloating of interfaces

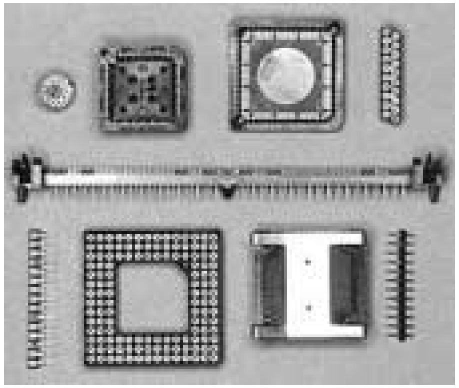
when developers extend or modify component functionality

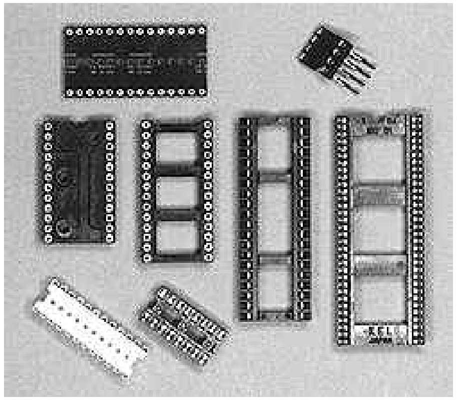




# Using Other Components

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



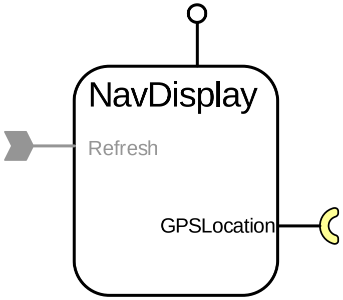


# Component Receptacles

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

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

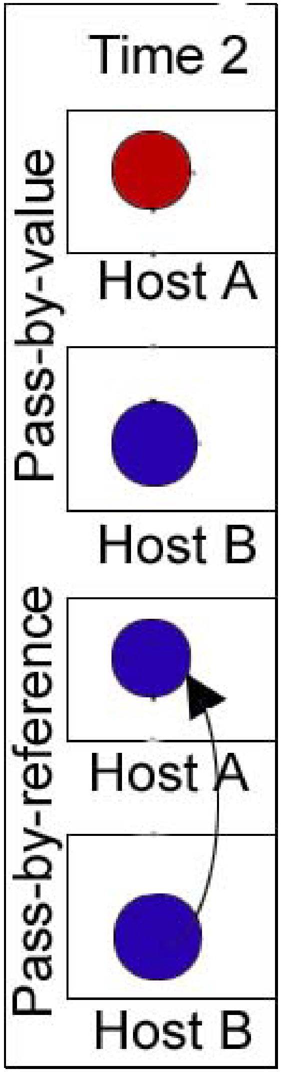
- Receptacle characteristics
  - Define a way to connect one or more *required* interfaces to this component
    - Specified with **uses** keyword
    - Can be *simplex* or *multiplex*
      - Connections are established *statically* via configuration & deployment tools during initialization stage or assembly stage
      - Connections are managed *dynamically* at run-time by containers to offer interactions with clients or other components via callbacks



# Event Passing

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

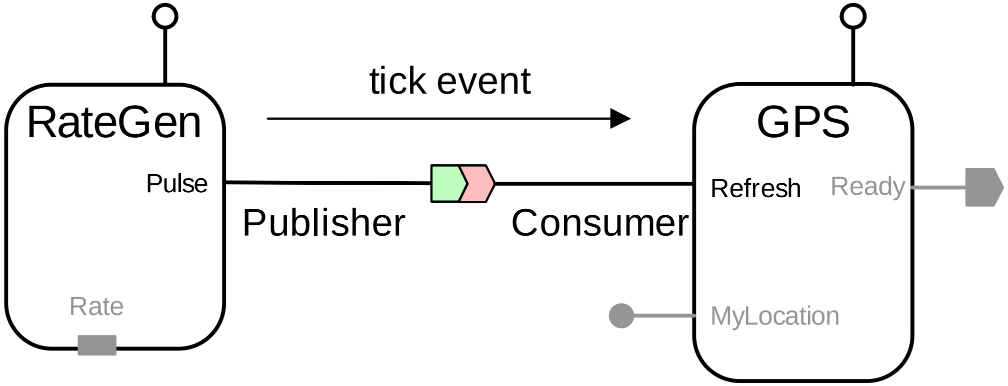
# CORBA Valuetypes



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

# Component Events

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



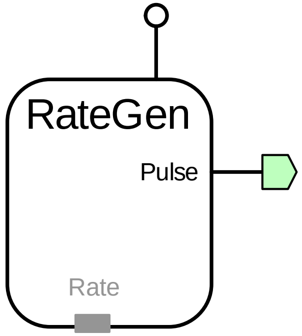
```
// Equivalent IDL 2
valuetype tick : Components::EventBase
{
  public rateHz Rate;
};
interface tickConsumer :
  Components::EventConsumerBase {
  void push_tick
    (in tick the_tick);
};
```

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

# Component Event Sources

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

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



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

# CCM Cookies

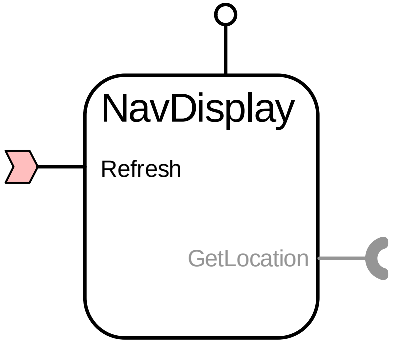
```
module Components
{
  valuetype Cookie
  {
    private CORBA::OctetSeq
            cookieValue;
  };

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

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

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

# Component Event Sinks

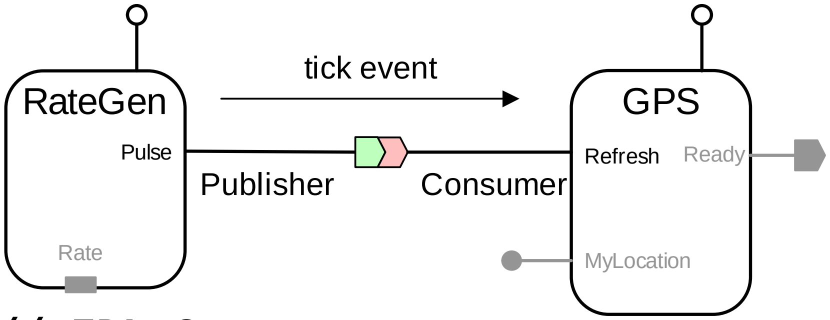
```
// IDL 3
component NavDisplay
{
 …
 consumes tick Refresh;
};
```

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



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

# CCM Events



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

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

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

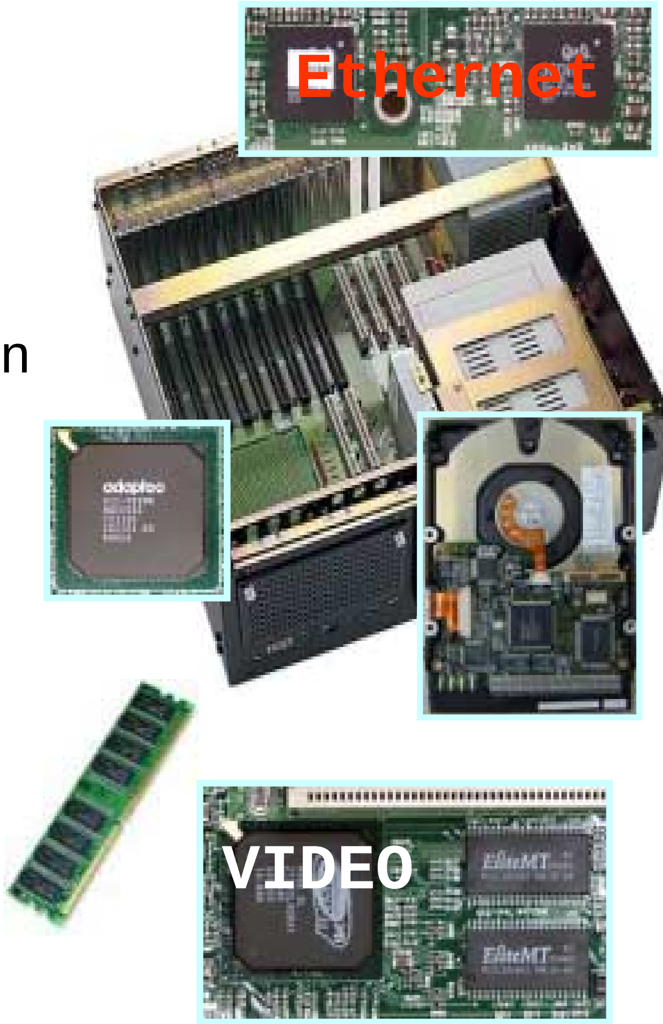
- Context
  - Generic event **push()** operation requires a generic event type
- Problem
  - User-defined **eventtypes** are not generic
- CCM Solution
  - **EventBase abstract valuetype**

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

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

Enables both statically- & dynamically-typed event passing

# The Need to Configure Components

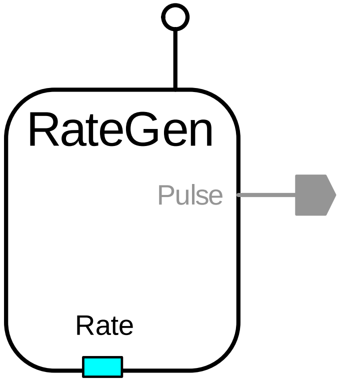
- Context
  - To make component implementations more adaptable, components should be (re)configurable
- Problems
  - Applications shouldn't commit to a configuration too early
  - No standard way to specify component's configurable knobs in CORBA 2.x
  - Need standard mechanisms to configure components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



# Component Attributes

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

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

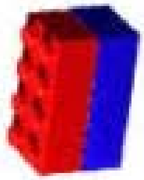


- Attribute characteristics
  - Named configurable properties intended for component configuration
    - e.g., optional behaviors, modality, resource hints, etc.
  - Can raise user-defined exceptions (new CCM capability)
  - Exposed through accessors & mutators
  - Can be set by various types of configuration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

# Connecting Components

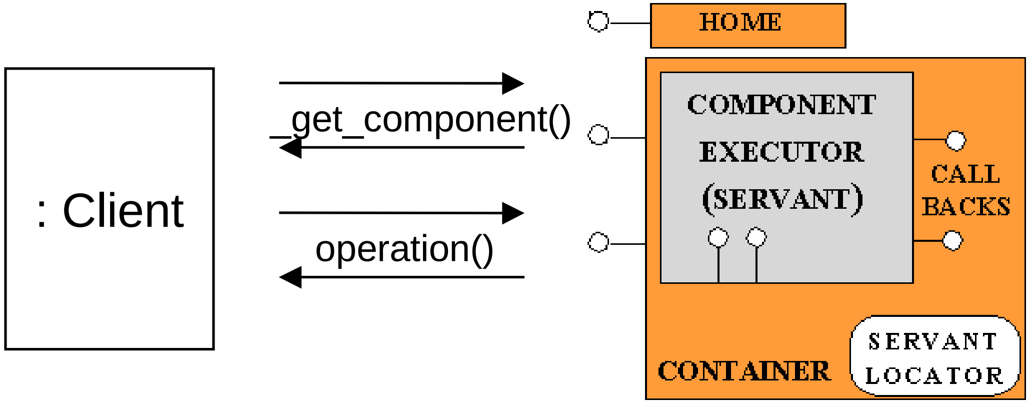
- Context
  - Components need to be connected together to form complete applications
- Problems
  - Components can have multiple ports with different types & names
  - It's not scalable to write code manually to connect a set of components for a specific application

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



# CCM Navigation & Introspection

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



# Using Navigation Interfaces of a Component

```
int
main (int argc, char *argv[])
{
  CORBA::ORB_var orb =
    CORBA::ORB_init (argc, argv);
    // Get the NameService reference…
  CORBA::Object_var o = ns->resolve_str ("HelloHome");
  HelloHome_var hh = HelloHome::_narrow (o.in ());
  HelloWorld_var hw = hh->create ();
  // Get all facets & receptacles
  Components::FacetDescriptions_var fd = hw->get_all_facets ();
  Components::ReceptacleDescriptions_var rd =
    hw->get_all_receptacles ();
  // Get a named facet with a name “Farewell”
  CORBA::Object_var fobj = hw->provide (“Farewell”);
  // Can invoke sayGoodbye() operation on Farewell after
  // narrowing to the Goodbye interface.
  return 0;
}
```

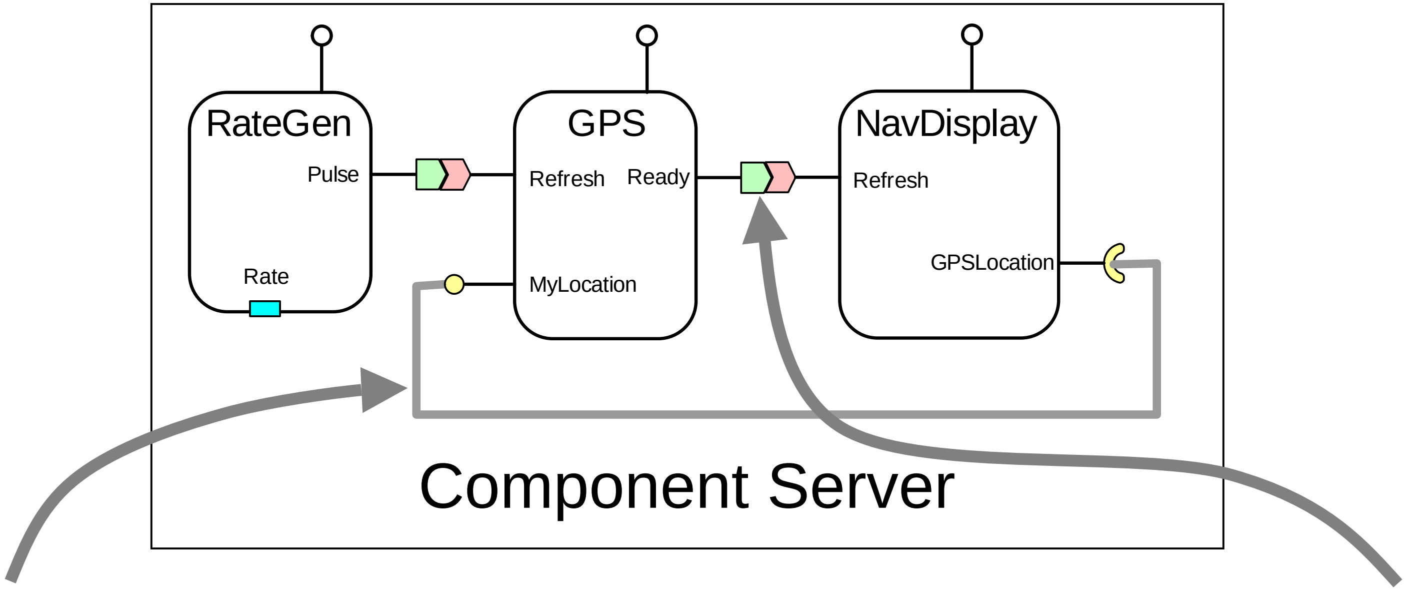
# Generic Port Operations

| Port | Equivalent IDL2 Operations | Generic Port Operations (**`CCMObject`**) |
|---|---|---|
| Facets | `provide_name ();` | `provide ("name");` |
| Receptacles | `connect_name (con);`<br>`disconnect_name ();` | `connect ("name", con);`<br>`disconnect ("name");` |
| Event sources (publishes only) | `subscribe_name (c);`<br>`unsubscribe_name ();` | `subscribe ("name", c);`<br>`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 CCM deployment & configuration tools
  - Lightweight CCM spec doesn't include equivalent IDL 2 operations

# Example of Connecting Components

CCM components are connected via deployment tools during launch phase



- Facet → Receptacle

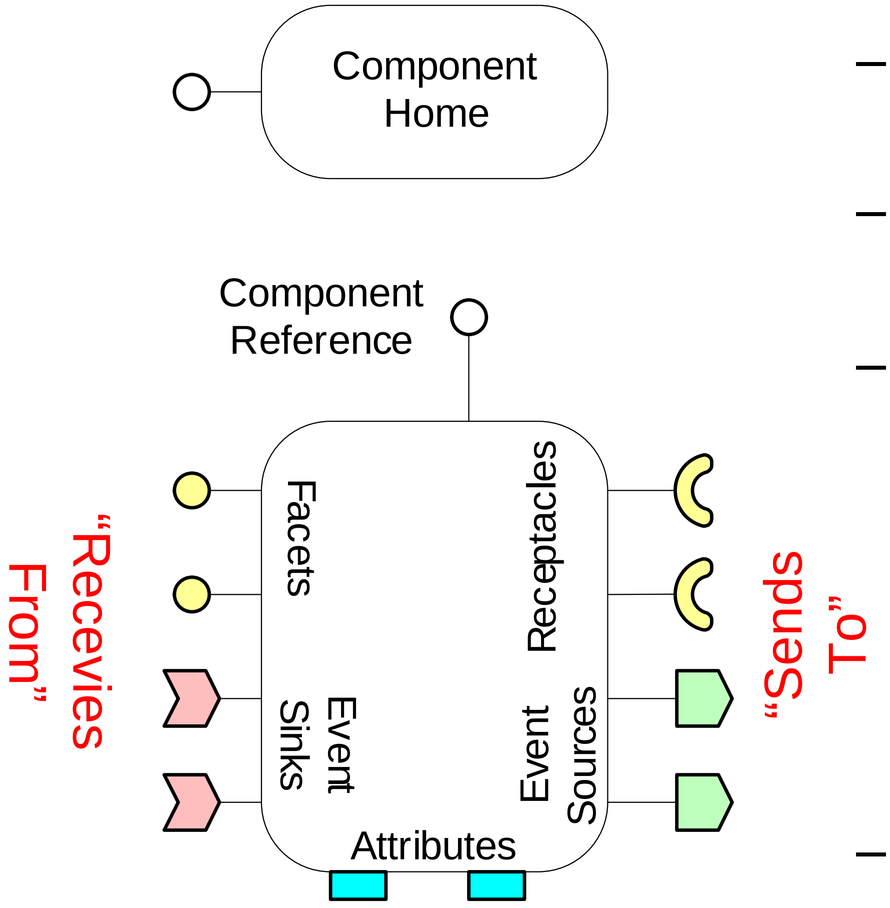
```
objref = GPS->provide
  (“MyLocation”);
NavDisplay->connect
  (“GPSLocation”, objref);
```

- Event Source → Event Sink

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

Connected object references are managed by containers

# Recap – CCM Component Features



- IDL 3 component from a *client* perspective
  - Define component life cycle operations (*i.e.*, **home**)
  - Define what a component **`provides`** to other components
  - Define what a component **`requires`** from other components
  - Define what *collaboration modes* are used between components
    - Point-to-point via operation invocation
    - Publish/subscribe via event notification
  - Define which component **`attributes`** are configurable
- IDL 3 maps to "equivalent IDL 2 Interfaces"

# Summary of Client OMG IDL Mapping Rules

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

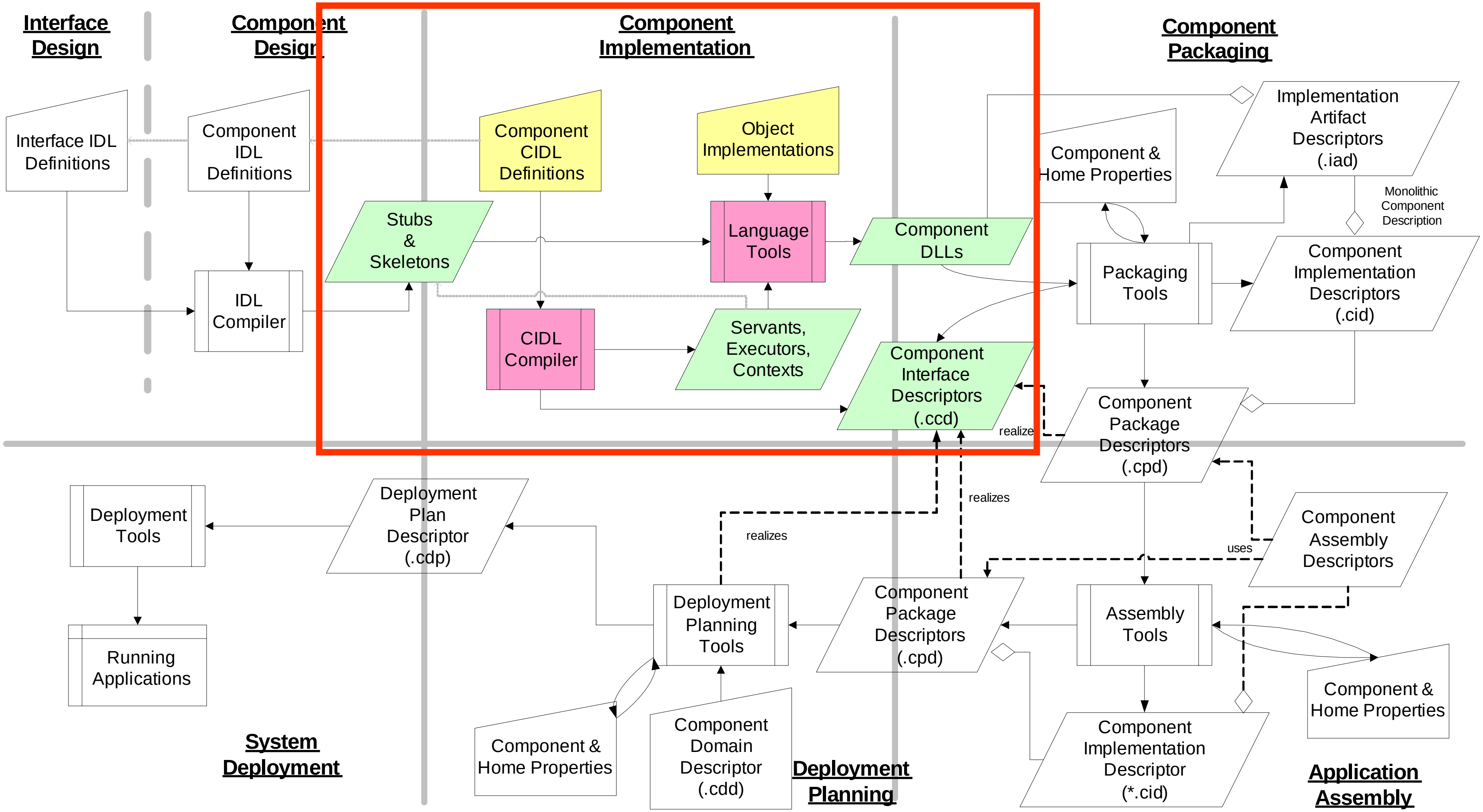
We explored all of these mappings in detail in previous slides

# CCM Component Run-time Environment & Containers

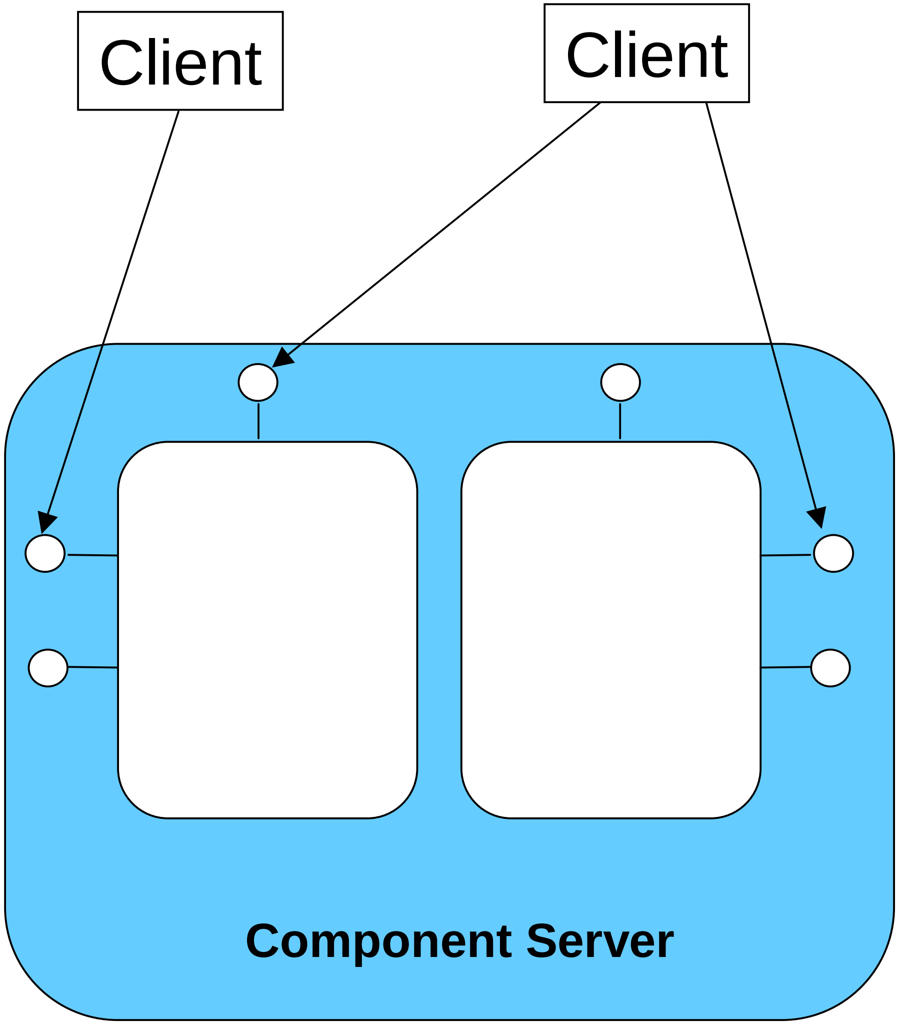
www.cs.wustl.edu/~schmidt/cuj-18.doc

# Component Implementation Stage

Goal: Implement components in the context of containers

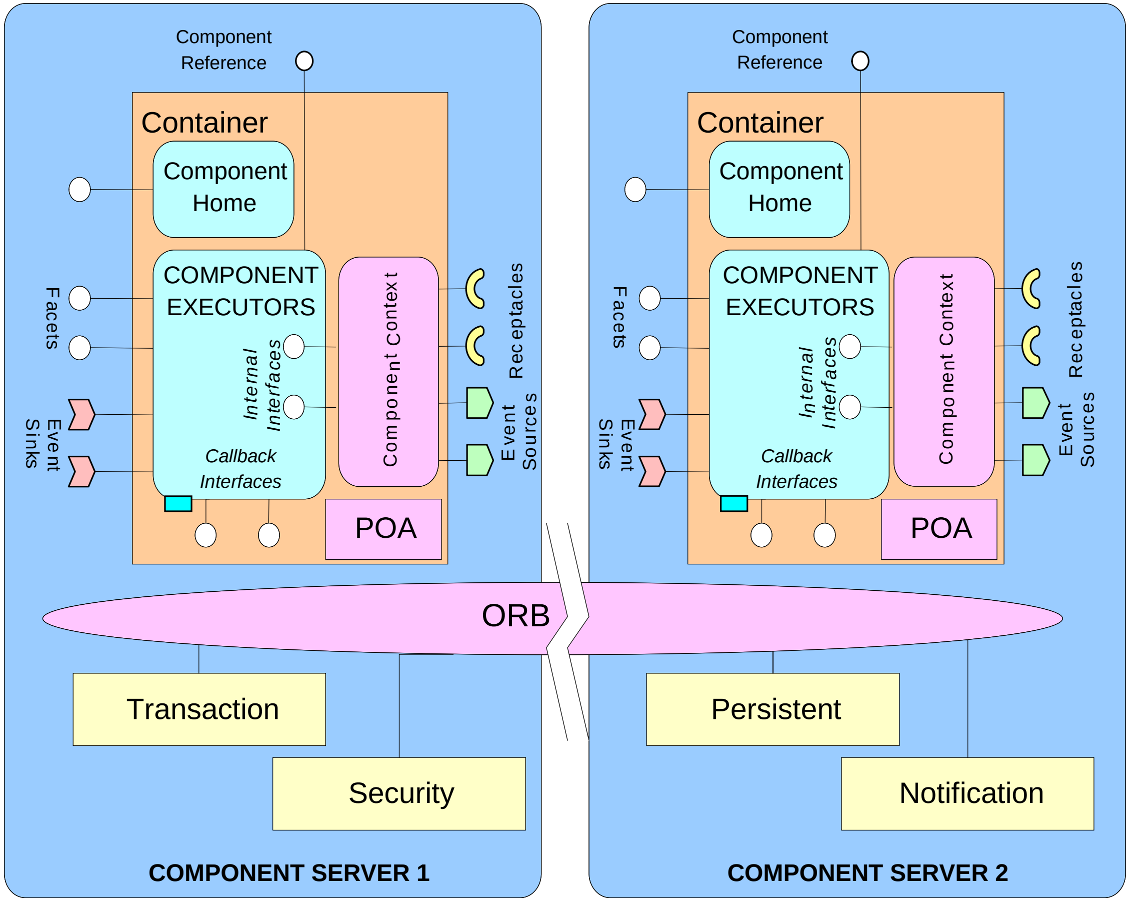

Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Stubs & Skeletons
Language Tools
Component DLLs
IDL Compiler
CIDL Compiler
Servants, Executors, Contexts
Component Interface Descriptors (.ccd)
realize
Component & Home Properties
Implementation Artifact Descriptors (.iad)
Monolithic Component Description
Packaging Tools
Component Implementation Descriptors (.cid)
Component Package Descriptors (.cpd)
Deployment Tools
Deployment Plan Descriptor (.cdp)
realizes
realizes
Component Assembly Descriptors
uses
Running Applications
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component & Home Properties
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
System Deployment
Deployment Planning
Application Assembly

# CCM Component Server Features



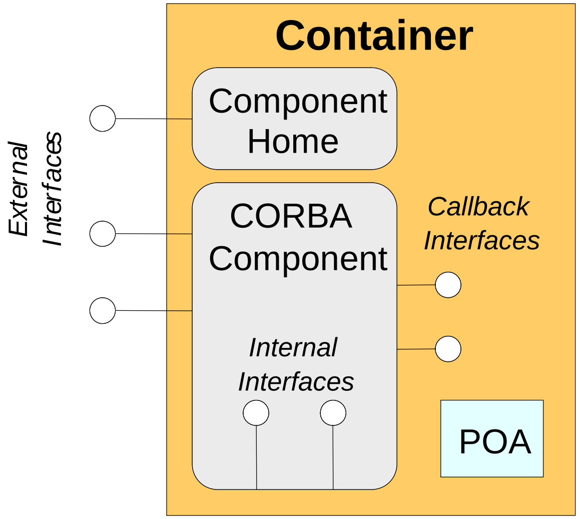
- CCM's primary enhancement to CORBA 2.x is its focus on *component servers & application configuration/deployment*
- Enhance CORBA 2.x by supporting
  - Higher-level abstractions of common servant usage models
  - Tool-based configuration & *meta-programming* techniques, e.g.:
    - Reusable run-time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

# The CCM Container Framework



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

# External, Internal, & Container Interfaces



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

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

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

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

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

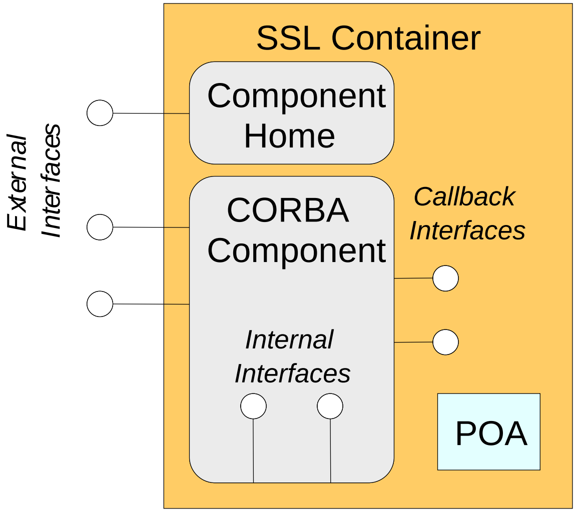
# CCM Component/Container Categories

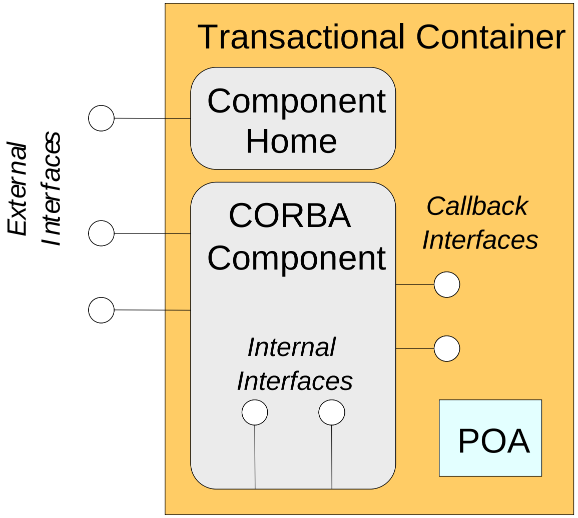
| *COMPONENT CATEGORY* | *CONTAINER IMPL TYPE* | *CONTAINER TYPE* | *EXTERNAL TYPE* |
|---|---|---|---|
| ***Service*** | Stateless | Session | Keyless |
| *Session* | *Conversational* | *Session* | *Keyless* |
| ***Process*** | Durable | Entity | Keyless |
| ***Entity*** | Durable | Entity | Keyfull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

These categories can be specified *declaratively* via a CIDL file

# Container-managed CORBA Policies

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





# CORBA Implementation Framework (CIF)

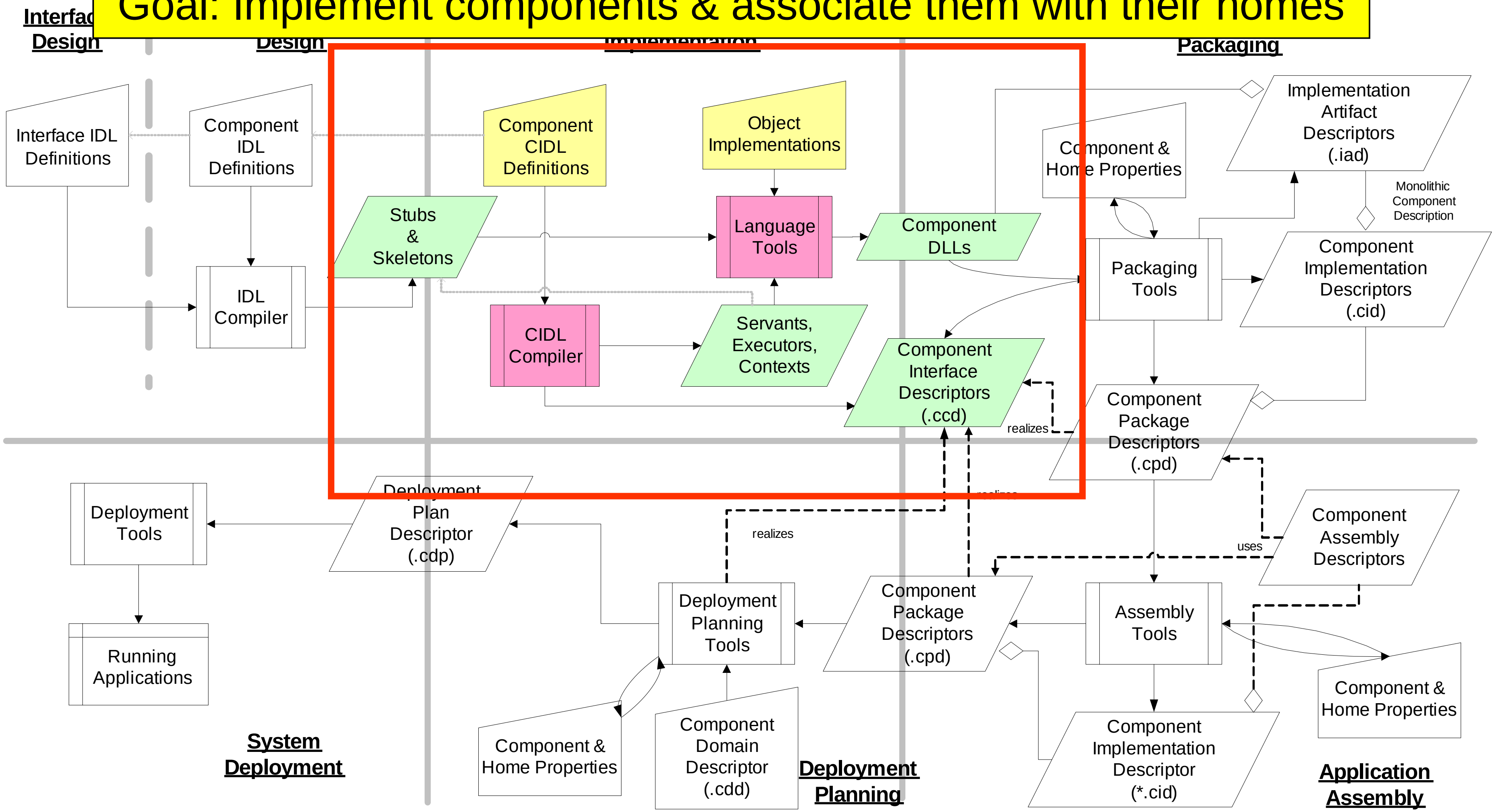
# &

# Component Implementation Definition Language (CIDL)

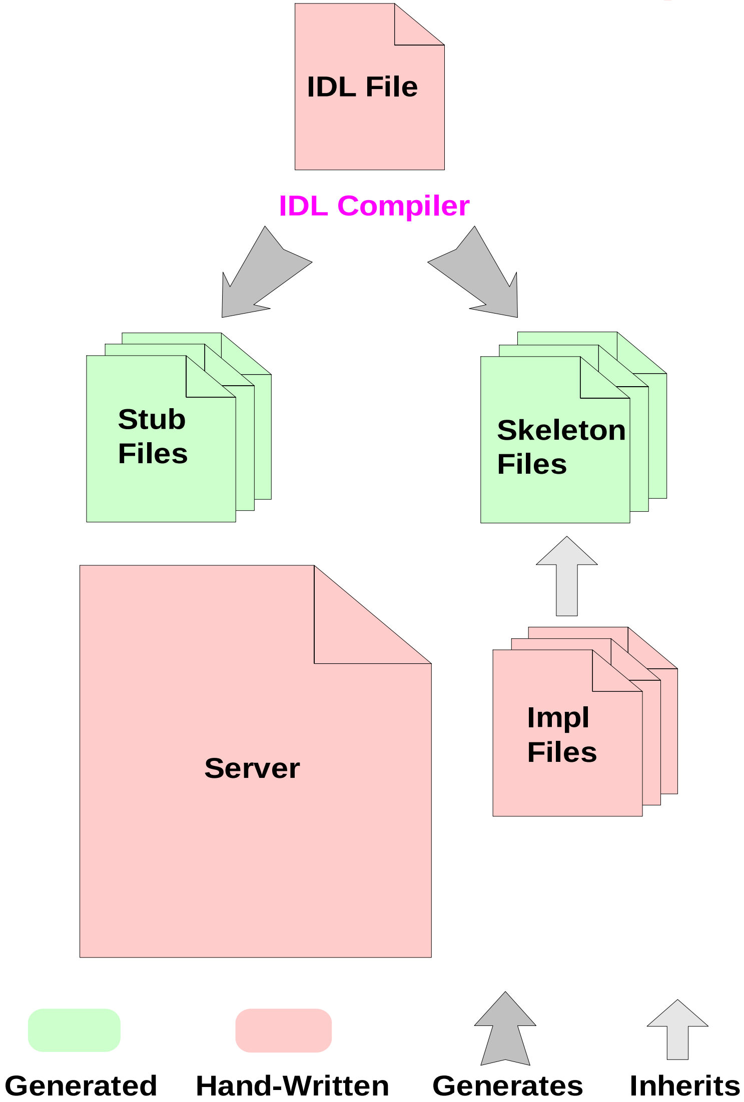
www.cs.wustl.edu/~schmidt/cuj-18.doc

# Component Implementation Stage

Goal: Implement components & associate them with their homes


Interfac
Design
Design
Implementation
Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Component & Home Properties
Implementation Artifact Descriptors (.iad)
Stubs & Skeletons
Language Tools
Component DLLs
Monolithic Component Description
IDL Compiler
Packaging Tools
Component Implementation Descriptors (.cid)
CIDL Compiler
Servants, Executors, Contexts
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
Deployment Tools
Deployment Plan Descriptor (.cdp)
realizes
realizes
uses
Component Assembly Descriptors
Running Applications
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component & Home Properties
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
System Deployment
Deployment Planning
Application Assembly

# Difficulties with Implementing CORBA 2.x Objects



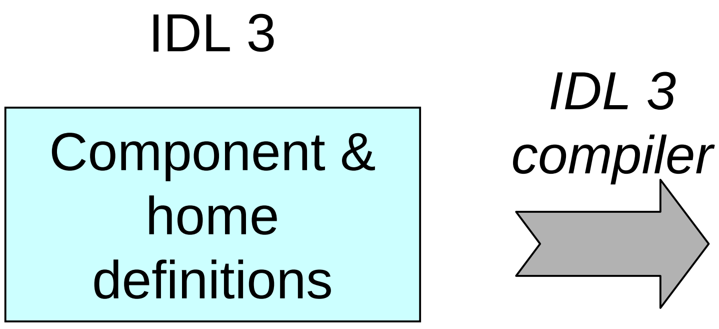
- **Problems**
  - Generic lifecycle & initialization server code must be handwritten, e.g.
    - Server initialization & event loop code
    - Support for introspection & navigation of object interfaces
  - Server application developers must
    - Keep track of dependencies their objects have on other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

# Approach for Implementing Components

## Requirements

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

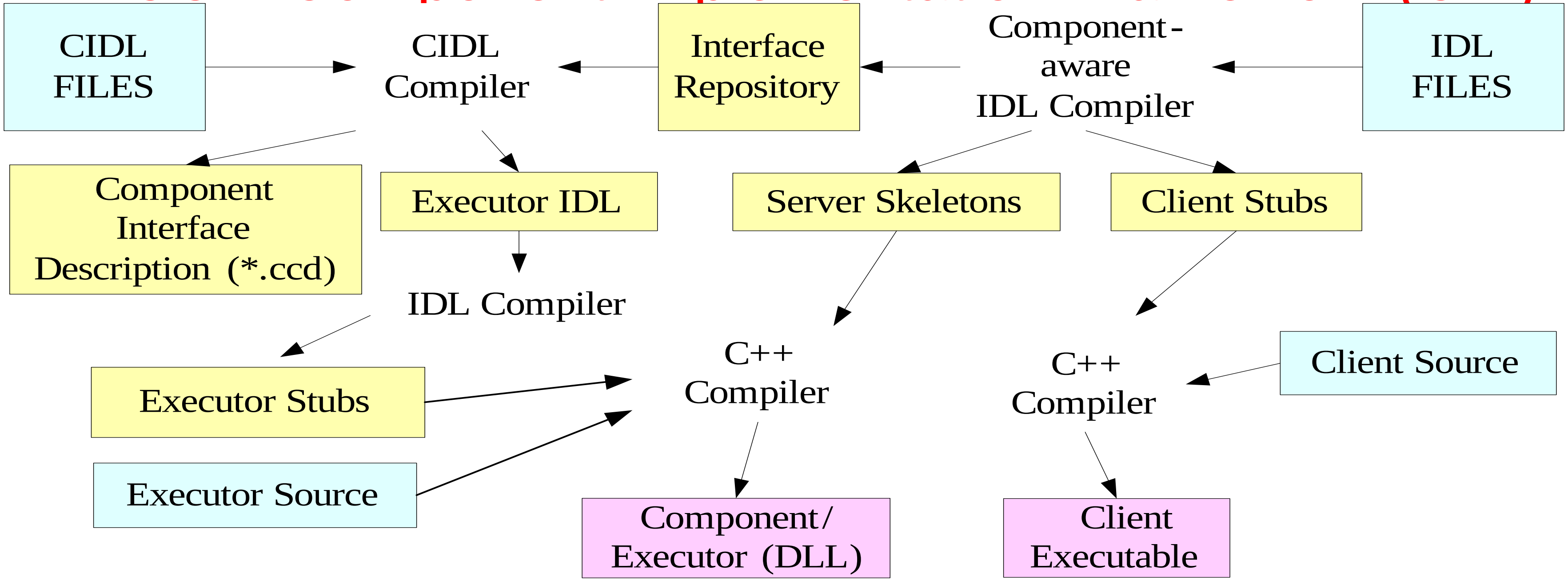
## Approach: Generate as Much Code as Possible from Declarative Specs



**Generated Component & Home Servants**

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

# CCM Component Implementation Framework (CIF)

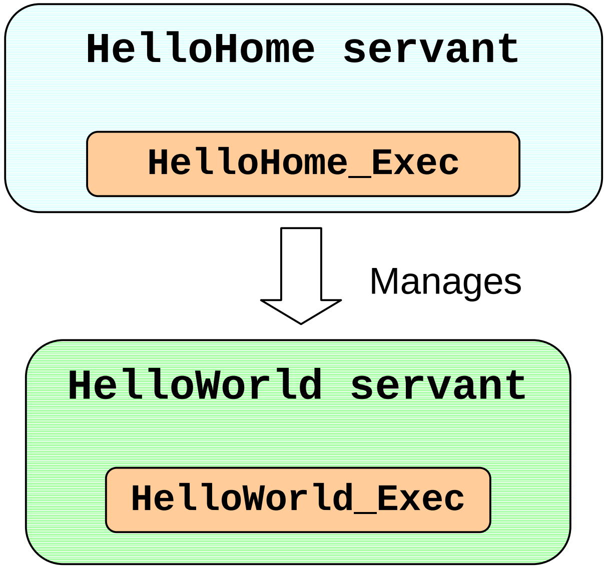


- Defines programming model rules & tools for developing component implementations
  - i.e., specifies how components should be implemented via *executors*

- Simplifies component implementation
  - Developers only implement business logic *not* activation, identification, port management, introspection, etc.
- Automates the generation of much component implementation “glue” code

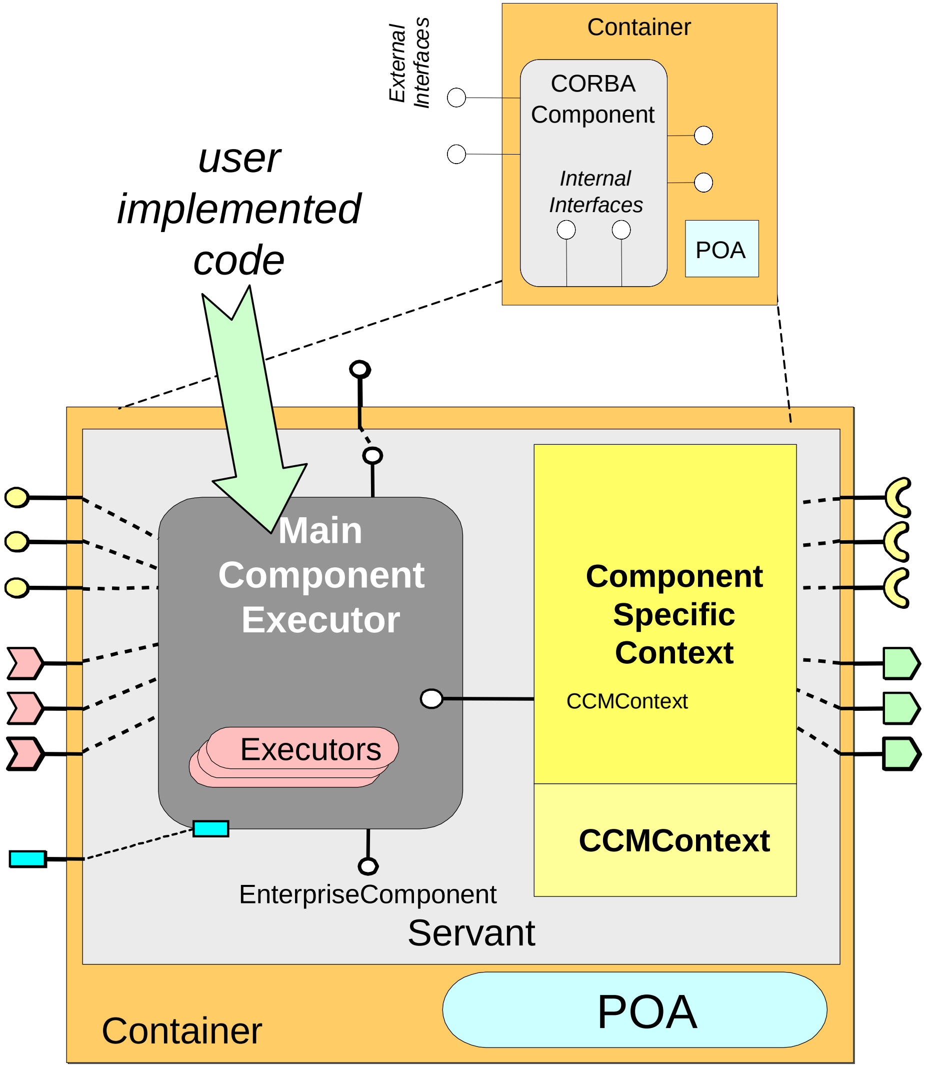
# CCM Executors & Home Executors

- Server-side programming artifacts that implement a component’s (or component home’s) behavior
  - Local CORBA objects with interfaces defined by a local server-side OMG IDL mapping
- Component executors can either be
  - *Monolithic*, where all component attributes, supported interfaces, facet operations, & event sinks implemented by a single class, or
  - *Segmented*, where component features split into several classes
- Home executors are always monolithic

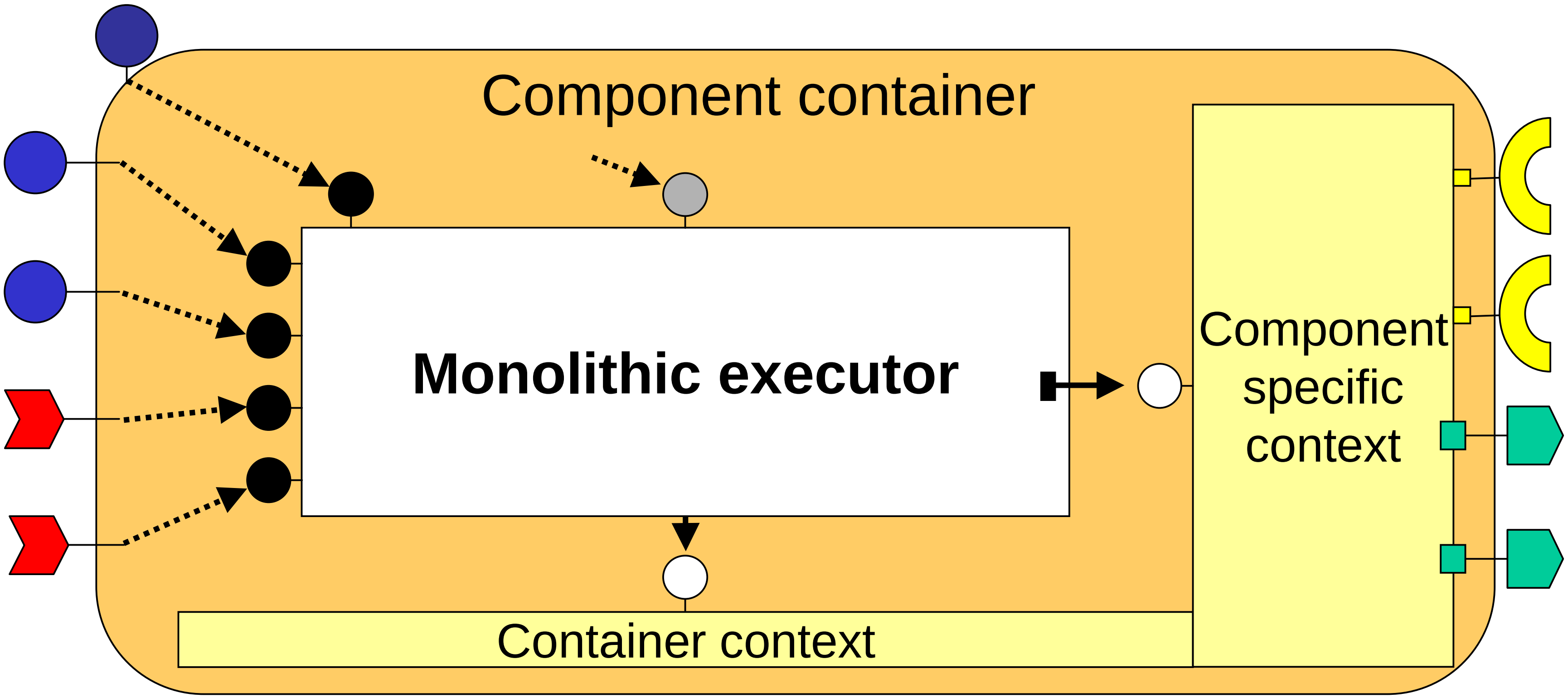


# Executors (& Servants) Are Hosted by Containers

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

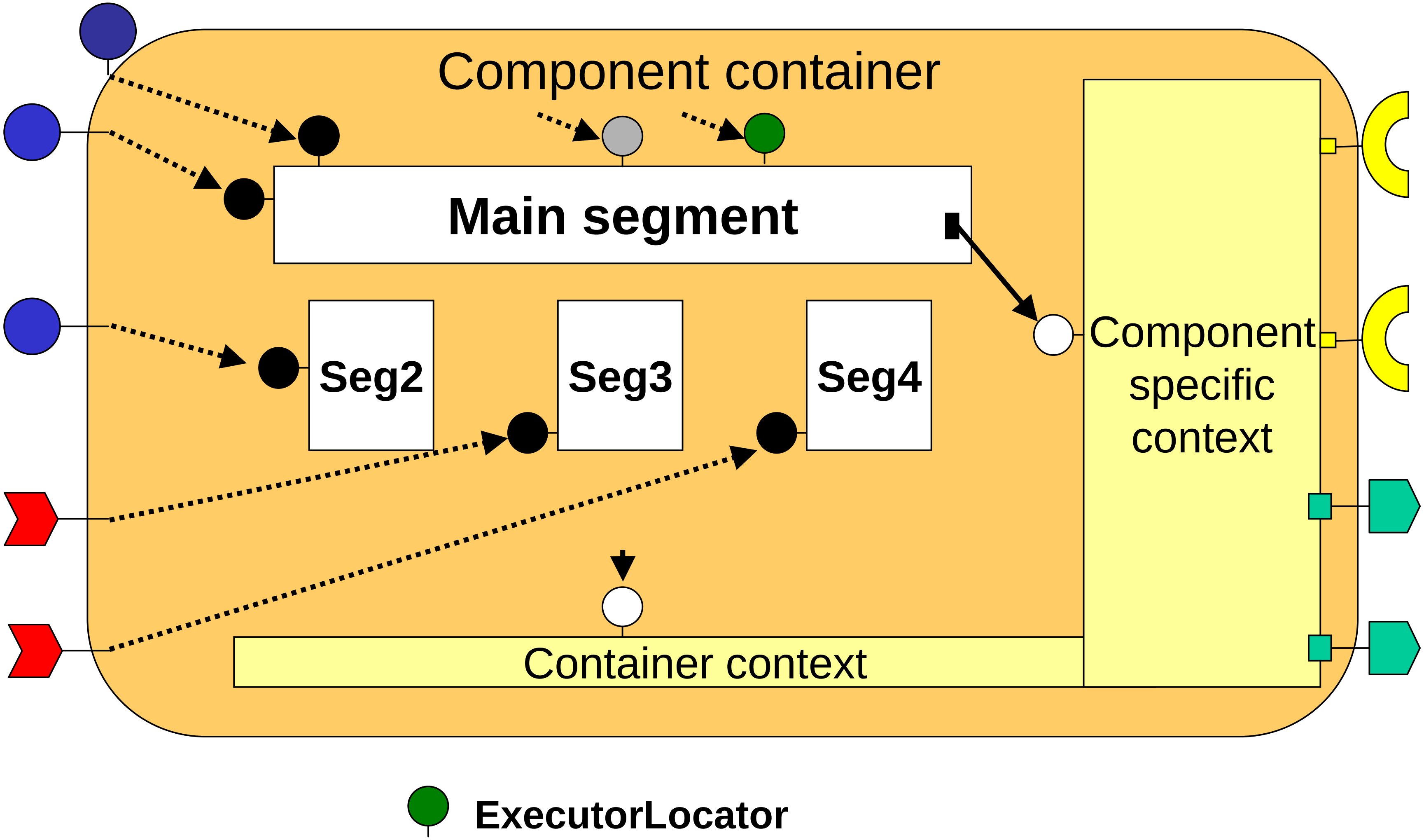


# A Monolithic Component Executor



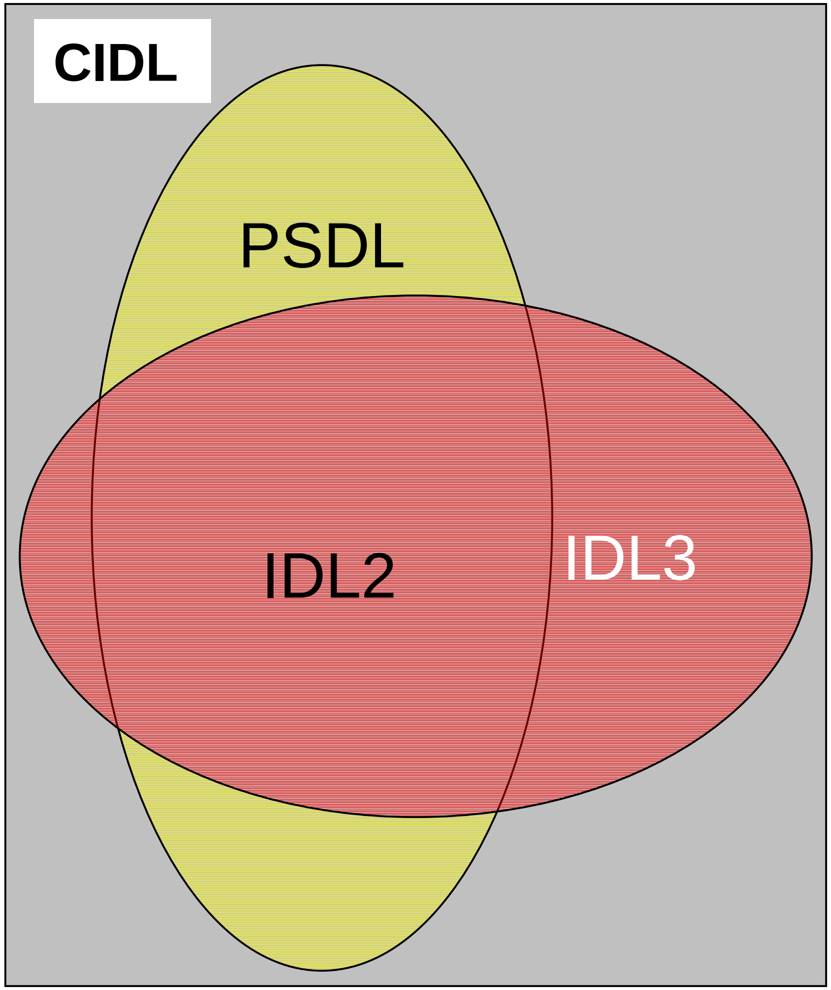
- ● Main component executor interface
- ● Facet or event sink executor interface
- ○ **SessionComponent** or **EntityComponent**
- ○ Component-oriented context interface
- ○ Container-oriented context interface
- → Context use
- ⇢ Container interposition

# A Segmented Component Executor



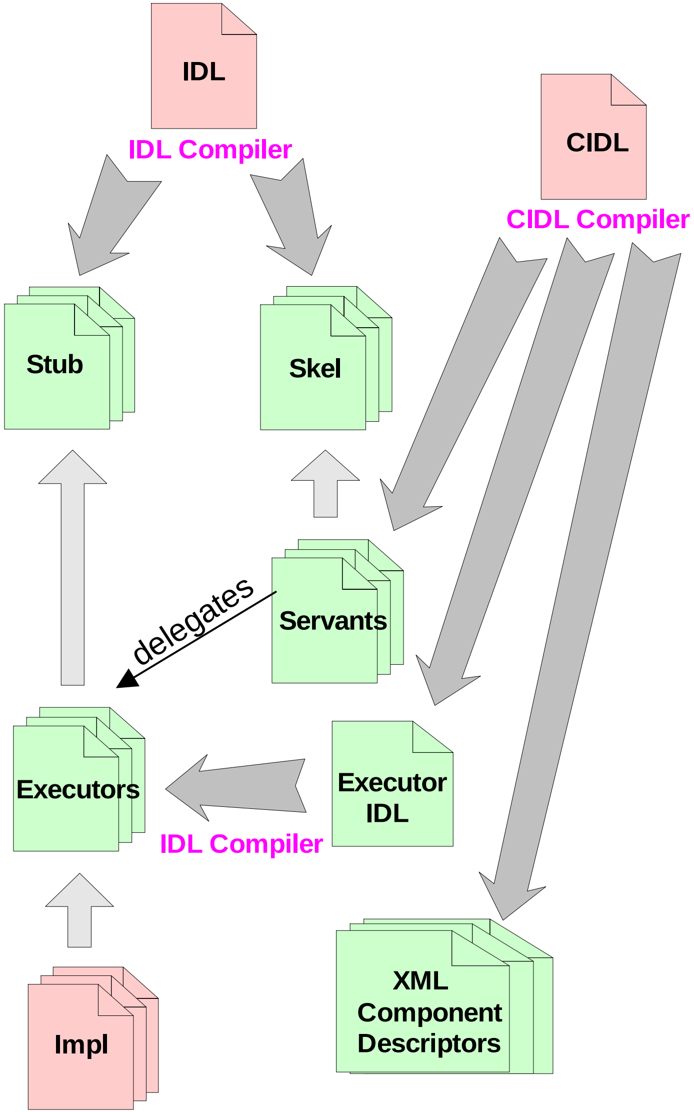
Segmented executors are deprecated in favor of assembly-based components

# Overview of Component Implementation Definition Language (CIDL)



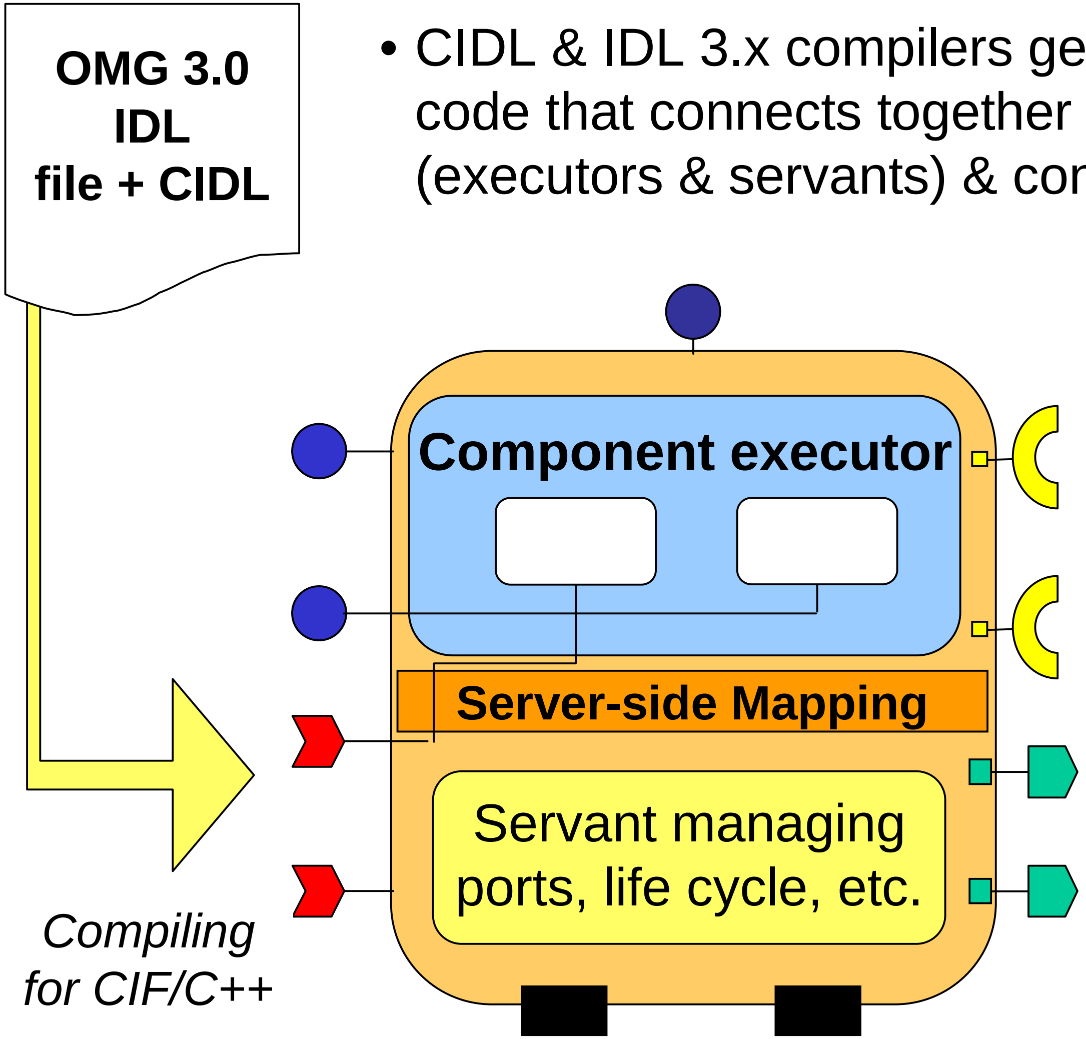
- Describes a component's *composition*
  - Aggregate entity that describes all artifacts required to implement a particular component & its home *executors* with their *interfaces*
- Can also manage component persistence state
  - Via OMG *Persistent State Definition Language* (PSDL)

# Facilitating Component Implementation via CIDL



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

# Connecting Components & Containers with CIDL

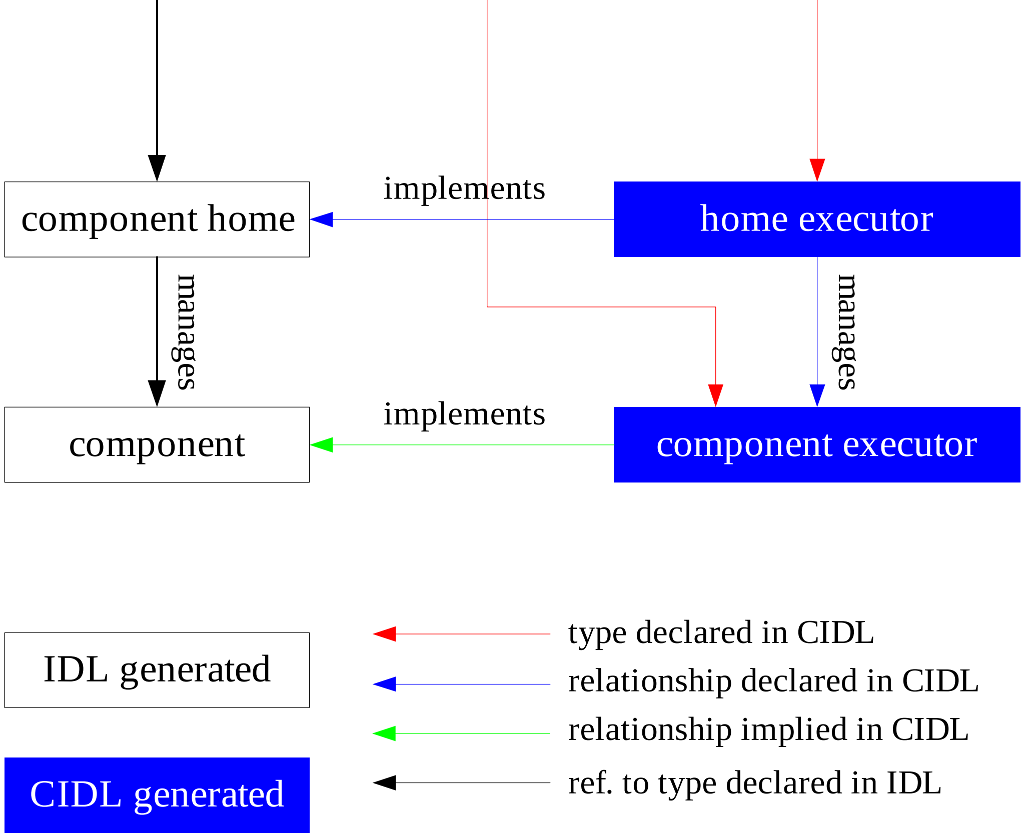


- CIDL & IDL 3.x compilers generate infrastructure “glue” code that connects together component implementations (executors & servants) & containers that hosts them

- Infrastructure code in container intercepts invocations on executors
  - e.g., can be used to manage activation, security, transactions, persistency, & so on
- CCM CIF defines “executor mappings”

# Facilitating Component Composition via CIDL

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



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

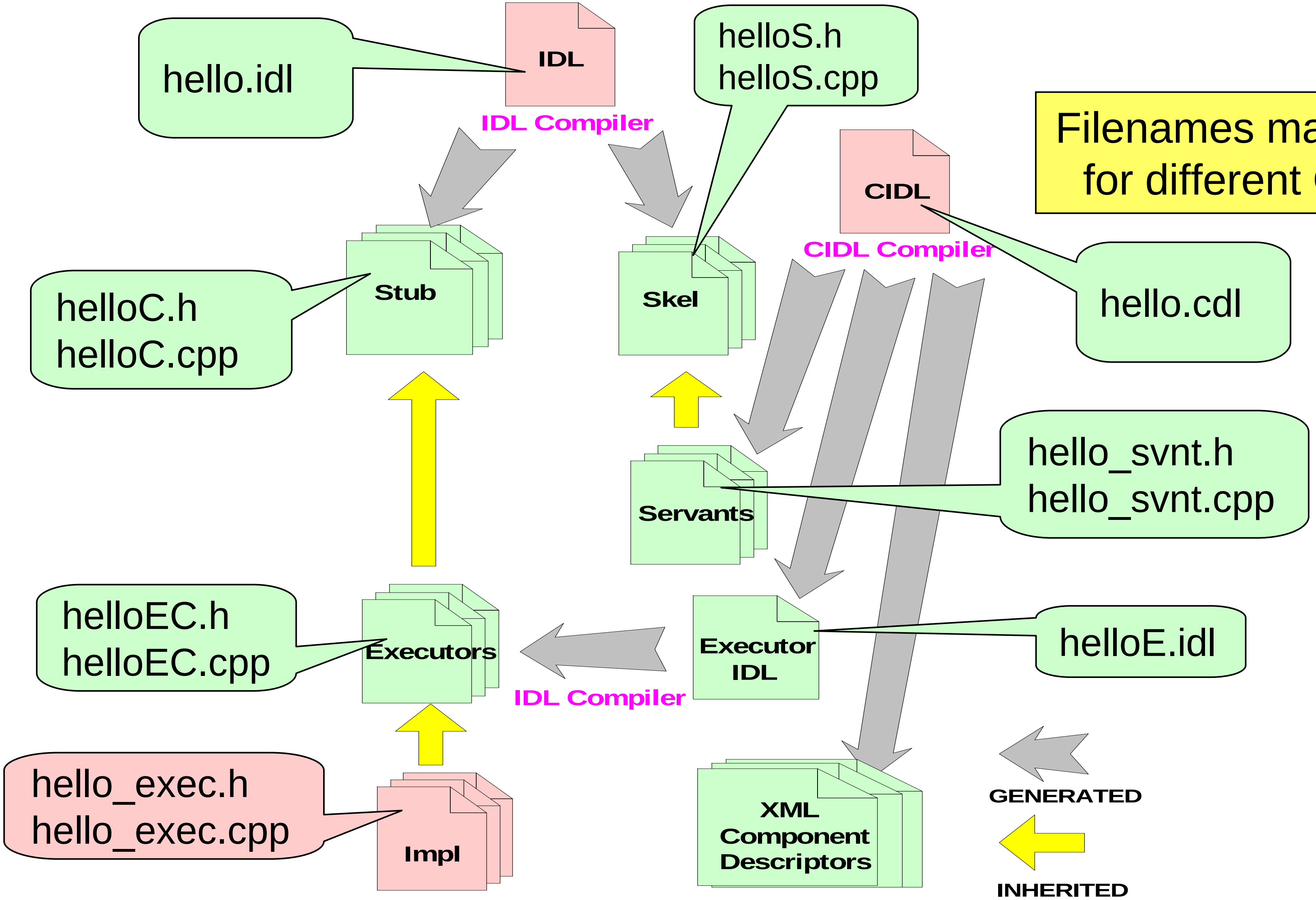
# CCM Component Application Examples

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

# Steps for Developing CCM Applications

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

# Overview of CCM Tool Chain for **HelloWorld** Example



Filenames may differ for different ORBs

# HelloWorld IDL 3 File & Generated Stub/Skel Code

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

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

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

```
// helloC.h – Stub file
class HelloHomeImplicit
 : public virtual ::Components::KeylessCCMHome {};
```

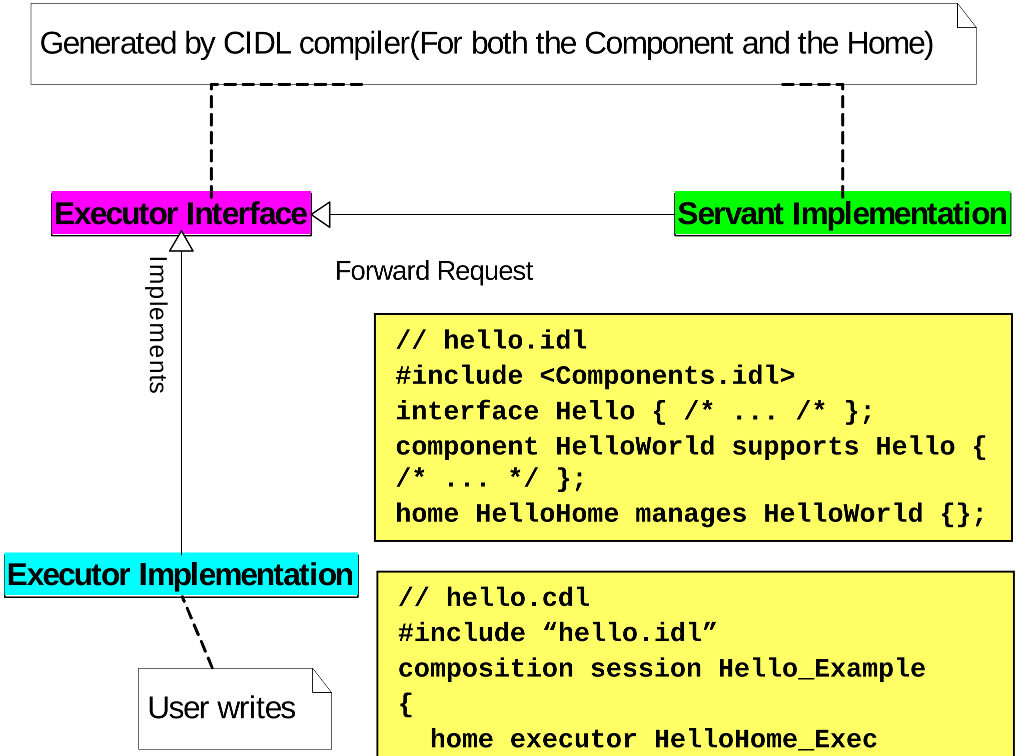
```
// helloC.h – Stub file
class HelloHomeExplicit
    : public virtual ::Components::CCMHome {};
```

```
// helloC.h – Stub file
class HelloHome
 : public virtual HelloHomeExplicit,
   public virtual HelloHomeImplicit {};
```

```
// helloS.h – Skeleton/Servant file
class POA_Hello
 : public virtual PortableServer::ServantBase {};
```

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

# **HelloWorld** CIDL & Generated Servant Code



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

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

Servant code also contains generated component-specific context classes

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

# HelloWorld CIDL-Generated Servants (hello_svnt.*)

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

*Compiling for CIF/C++*

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

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

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

# `HelloWorld` CIDL-Generated Servant Details (1/7)

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

interface Hello
{};

component HelloWorld supports Hello
{};

home HelloHome manages HelloWorld
{};
```

```
// hello.cdl
#include “hello.idl”

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

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

namespace Hello_Example
{
  class HelloWorld_Servant;

  class HelloWorld_Context
    : public virtual CCM_HelloWorld_Context {
    friend class HelloWorld_Servant;

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

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

# HelloWorld CIDL-Generated Servant Details (2/7)

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

interface Hello {};

interface Goodbye {};

eventtype MsgTrigger {};

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

home HelloHome manages HelloWorld {};
```

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

namespace Hello_Example {
  class HelloWorld_Servant;

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

    virtual Goodbye_ptr get_connection_GetGoodbye ();

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

    virtual Goodbye_ptr disconnect_GetGoodbye ();

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

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

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

# HelloWorld CIDL-Generated Servant Details (3/7)

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

interface Hello
{};

component HelloWorld supports Hello
{};

home HelloHome manages HelloWorld
{};
```

```
// hello.cdl
#include “hello.idl”

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

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

namespace Hello_Example
{
  class HelloWorld_Servant
    : public virtual POA_HelloWorld,
      public virtual PortableServer::RefCountServantBase
  {
    // Operation overrides from base classes –
    // Components::CCMObject, Components::Navigation,
    // Components::Receptacles, and Components::Events
  };
}
```

- Compiler navigates through `implements` & (IDL) `manages`
  - Gets component name
  - Maps name to servant & base class names.
- If no port declarations or supported interface operations
  - No new operations generated in servant class
  - Overrides generated for component middleware base class operations

# HelloWorld CIDL-Generated Servant Details (4/7)

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

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

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

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

home HelloHome manages HelloWorld
{};
```

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

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

- Facet (`provides`) declarations maps to C++ servant class
  - Separate servant class is implementation-specific
  - Helps C++ compiler keep footprint down
- Facet type maps to servant & base class name generation
- Facet interface operations mapped directly to servant class
  - Operation names map directly
  - Operation parameters map with the usual CORBA rules

# HelloWorld CIDL-Generated Servant Details (5/7)

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

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

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

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

home HelloHome manages HelloWorld
{};
```

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

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

    virtual Goodbye_ptr provide_Farewell ();

    virtual char * Message ();

    virtual void Message (const char * Message);

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

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

- **Supported op maps directly to component servant op**
- **Facet type maps to the return type of the accessor op**
- **Facet name maps to component servant accessor op**

- **Attribute maps to get/set ops in the component servant**
- **Event sink (`consumes`) maps to nested class (impl-specific)**
  - **Also maps to accessor op for the event consumer**

# HelloWorld CIDL-Generated Servant Details (6/7)

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

interface Hello
{};

component HelloWorld supports Hello
{};

home HelloHome manages HelloWorld
{};
```

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

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

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

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

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

# HelloWorld CIDL-Generated Servant Details (7/7)

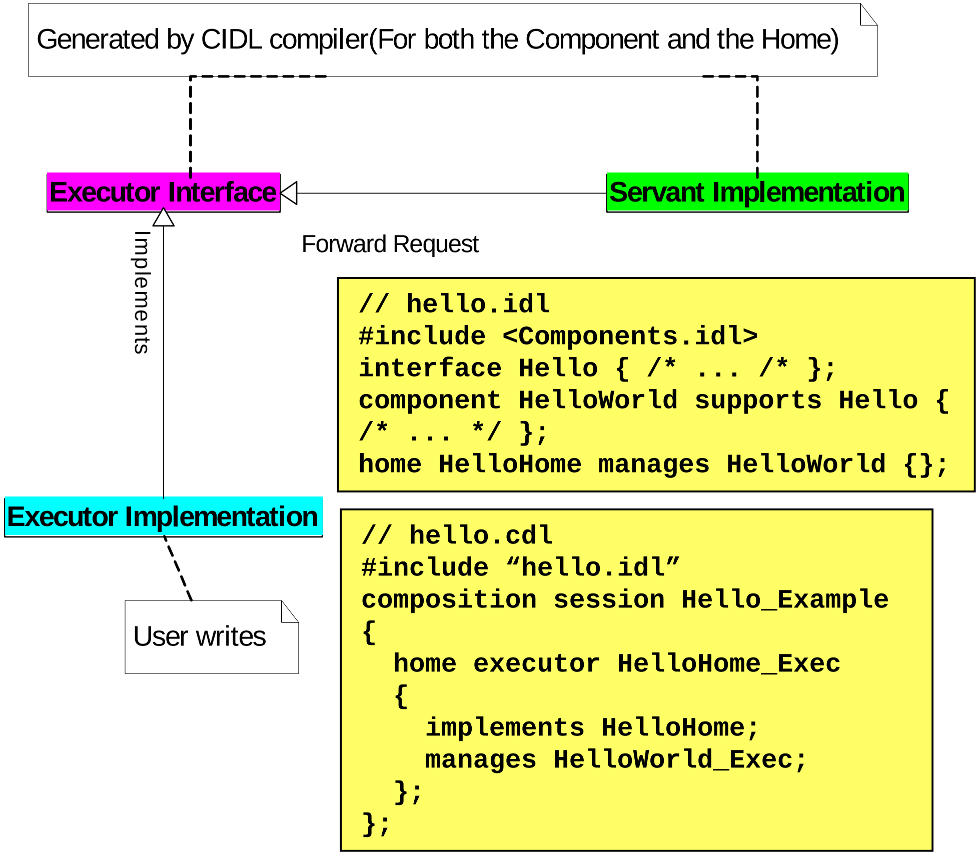
```
// hello.idl
#include <Components.idl>

interface Hello
{};

component HelloWorld supports Hello
{
  attribute string Message;
};

home HelloHome manages HelloWorld
{
  void UtilityOp ();

  factory Generate (in string msg);

  finder Lookup (in long key);

  attribute long DefaultKey;
};
```

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

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

namespace Hello_Example {
  class HelloHome_Servant
    : public virtual POA_HelloHome,
      public virtual PortableServer::RefCountServantBase{
  public:
    virtual void UtilityOp ();

    virtual HelloWorld_ptr Generate (const char * msg);

    virtual HelloWorld_ptr Lookup (CORBA::Long key);

    virtual CORBA::Long DefaultKey ();

    virtual void DefaultKey (CORBA::Long DefaultKey);
  };
}
```

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

# `HelloWorld` CIDL & Generated Executor Code



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

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

Component application developers only implement executor interfaces

- Executor interfaces are IDL or C++/Java code
  - Must be implemented by component developers
- Generated code has interfaces for
  - Implicit & explicit homes
  - Main home executor
  - Main and/or monolithic component executors
  - Facet & consumer executor
  - Component context
- All executor interfaces are “locality constrained”
  - i.e., use IDL keyword **`local`**

# HelloWorld CIDL-Generated Executor IDL (helloE.idl)

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

Component Executor Interface

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

Component Context Interface

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

Implicit Home interface

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

Explicit Home interface

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

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

Main Component Interface

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

Main Home Interface

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

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

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

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

interface Hello
{};

component HelloWorld supports Hello
{};

home HelloHome manages HelloWorld
{};
```

```
// hello.cdl
#include “hello.idl”

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

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

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

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

local interface CCM_HelloWorld_Context
  :  Components::SessionContext {};

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

local interface CCM_HelloHomeExplicit
  : Components::HomeExecutorBase {};

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

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

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

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

interface Hello
{};

component HelloWorld supports Hello
{};

home HelloHome manages HelloWorld
{};
```

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

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

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

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

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

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

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

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

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

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

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

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

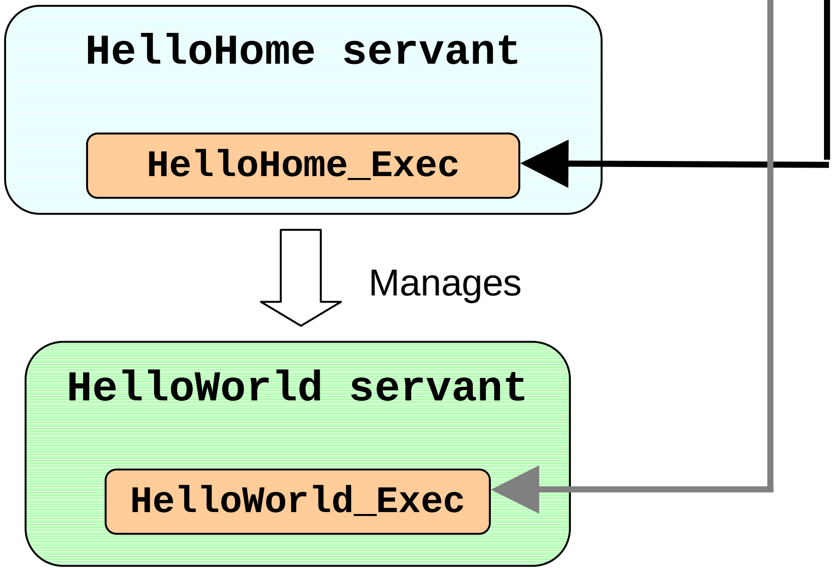
- Facet (**provides**) type maps to local interface, base class, & return type of accessor operation

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

local interface CCM_Goodbye : Goodbye {};

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

local interface CCM_HelloWorld_Context
  : Components::SessionContext {};

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

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

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

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

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

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



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

# HelloWorld Component Executors

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

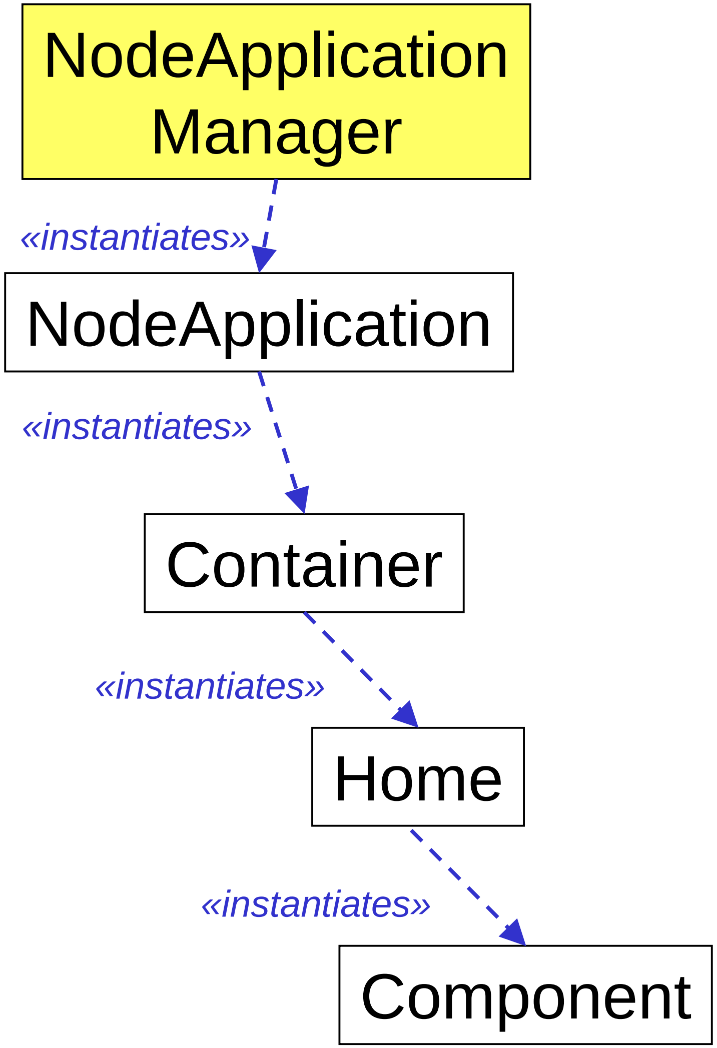
- **HelloWorld_Exec_Impl** executor implements **HelloWorld** component behavior
- **HelloHome_Exec_Impl** executor implements lifecycle management strategy of **HelloWorld** component

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

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

};
```

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

**$CIAO_ROOT/docs/tutorial/Hello/**

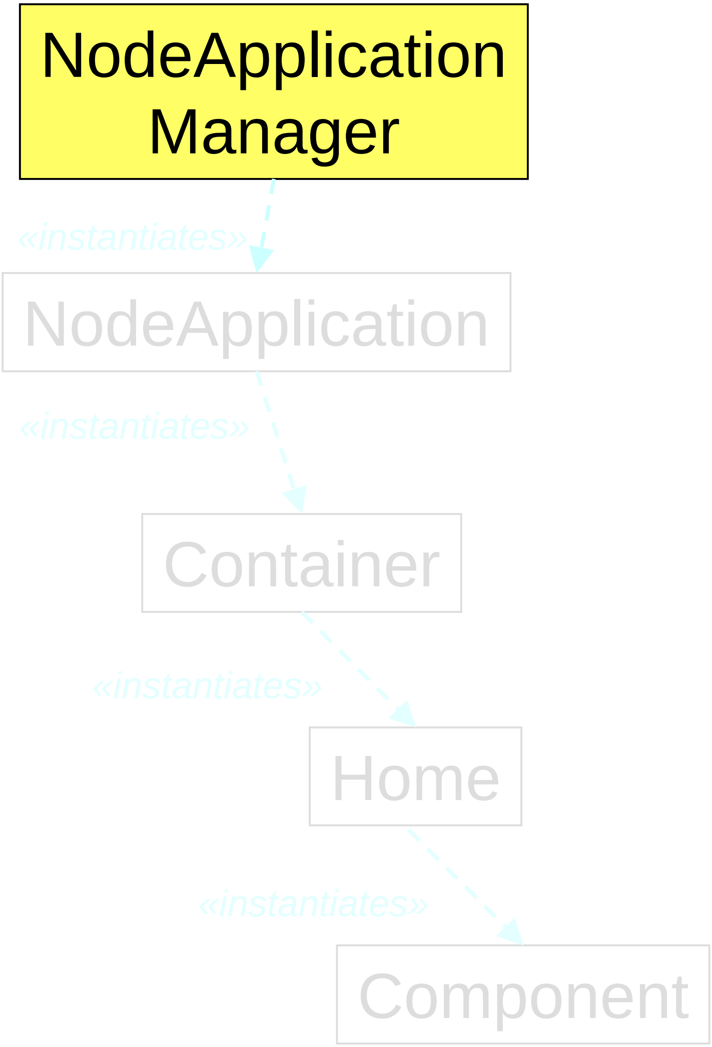
# Deployment & Configuration Process



Canonical steps in the application deployment & configuration process (performed by CCM Deployment & Configuration engine):

- Create the *NodeApplication* environment within which containers reside
- Create *containers* for the components
- Create & register *homes* for components
- Create & register the *components* themselves
- Establish *connections* between components

# Deployment & Configuration Process – Step 1



NodeApplication Manager

«instantiates»

NodeApplication

«instantiates»

Container

«instantiates»

Home

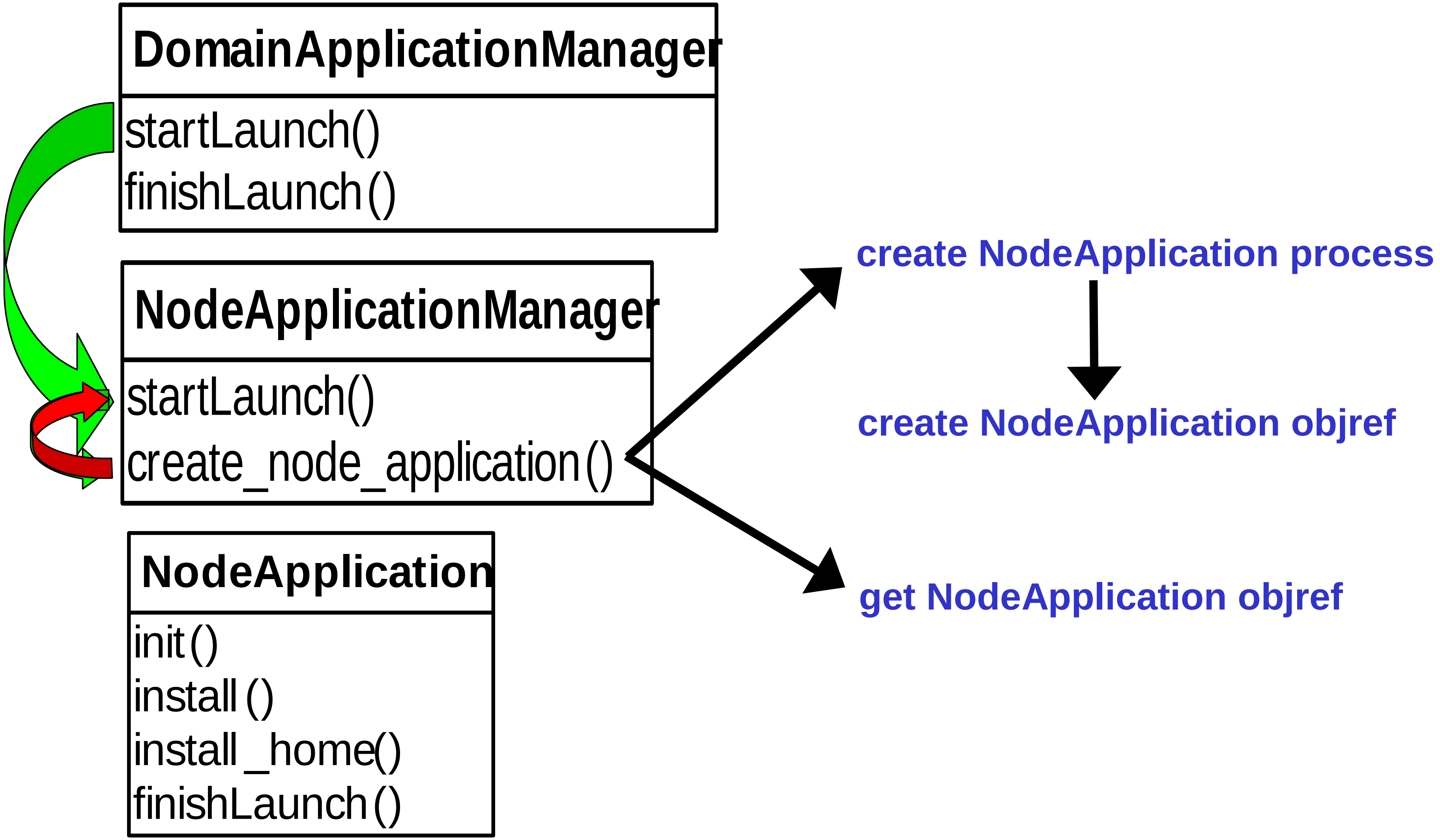
«instantiates»

Component

Canonical steps in the application deployment & configuration process (performed by CCM Deployment & Configuration engine):

- Create the *NodeApplication* environment within which containers reside
- Create *containers* for the components
- Create & register *homes* for components
- Create & register the *components* themselves
- Establish *connections* between components

# Creating a NodeApplication



**DomainApplicationManager**

startLaunch()
finishLaunch()

**NodeApplicationManager**

startLaunch()
create_node_application()

**NodeApplication**

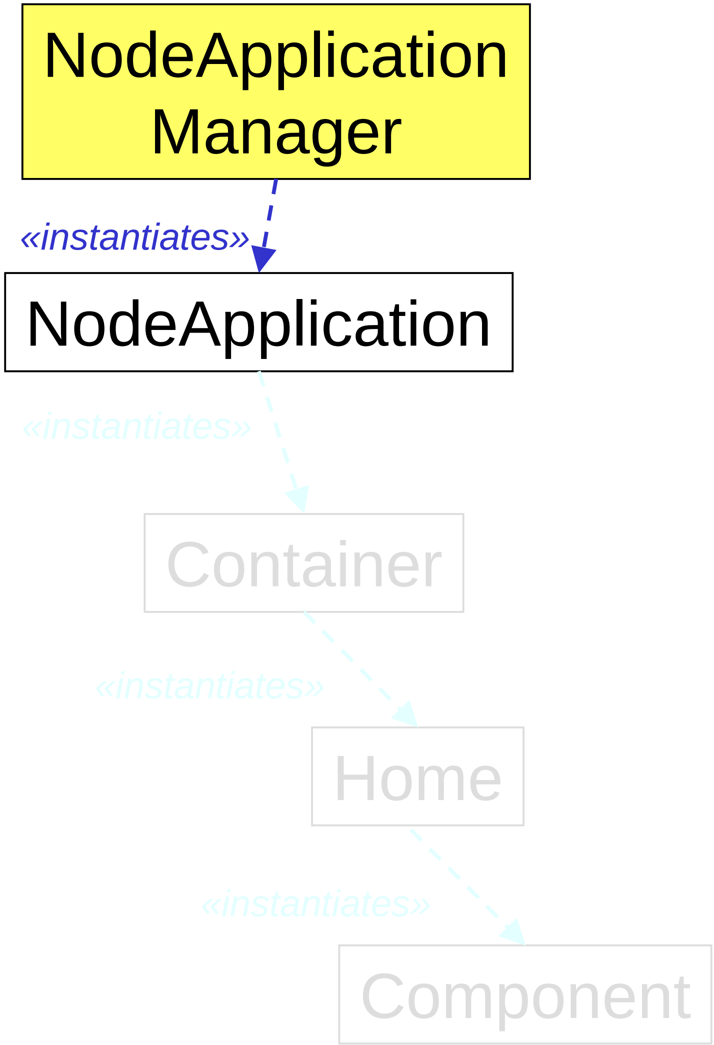
init()
install()
install_home()
finishLaunch()

**create NodeApplication process**

**create NodeApplication objref**

**get NodeApplication objref**

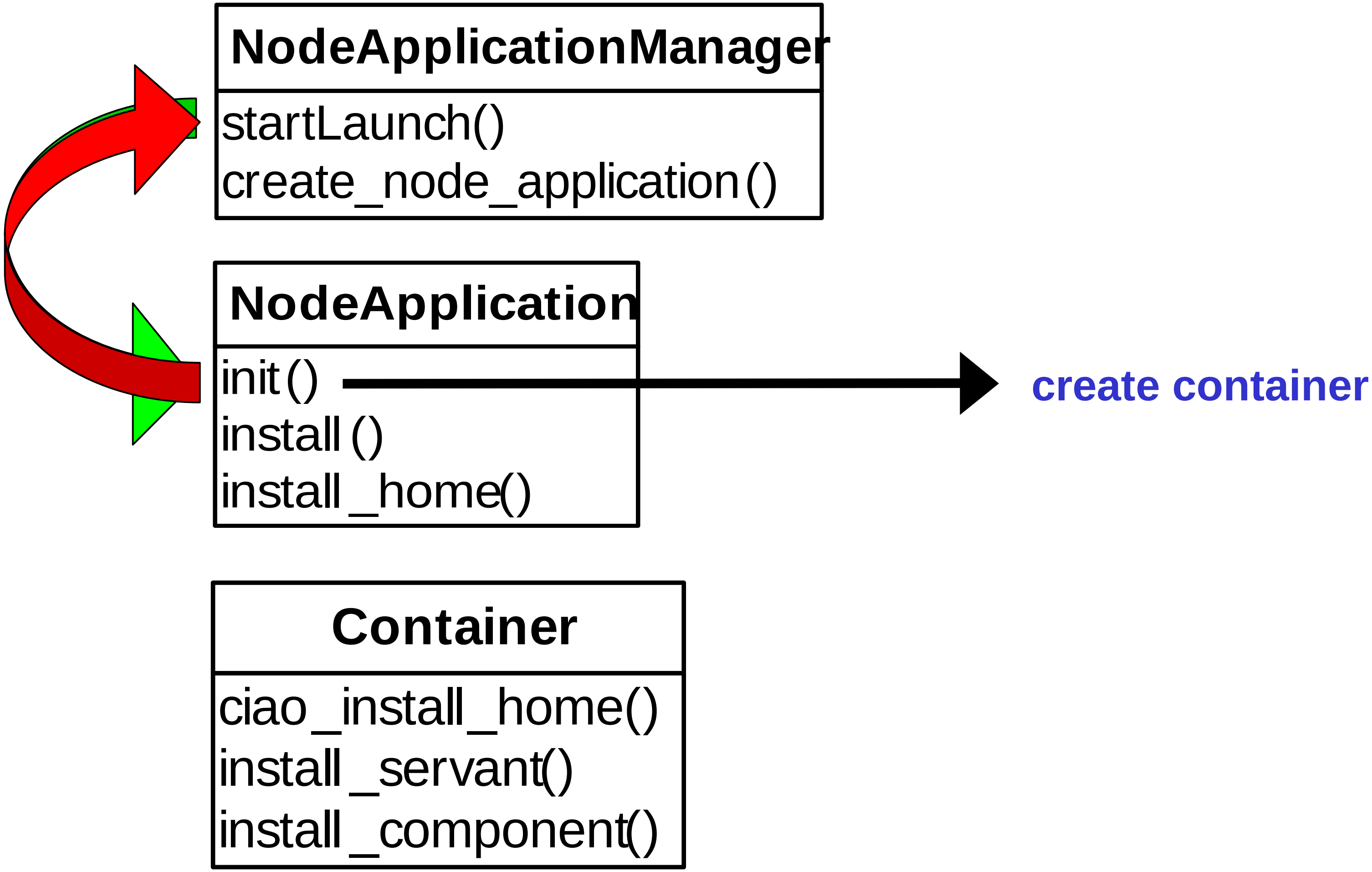
# Deployment & Configuration Process – Step 2



Canonical steps in the application deployment & configuration process (performed by CCM Deployment & Configuration engine):

- Create the *NodeApplication* environment within which containers reside
- Create *containers* for the components
- Create & register *homes* for components
- Create & register the *components* themselves
- Establish *connections* between components

# Creating a Container



**NodeApplicationManager**

startLaunch()
create_node_application()

**NodeApplication**

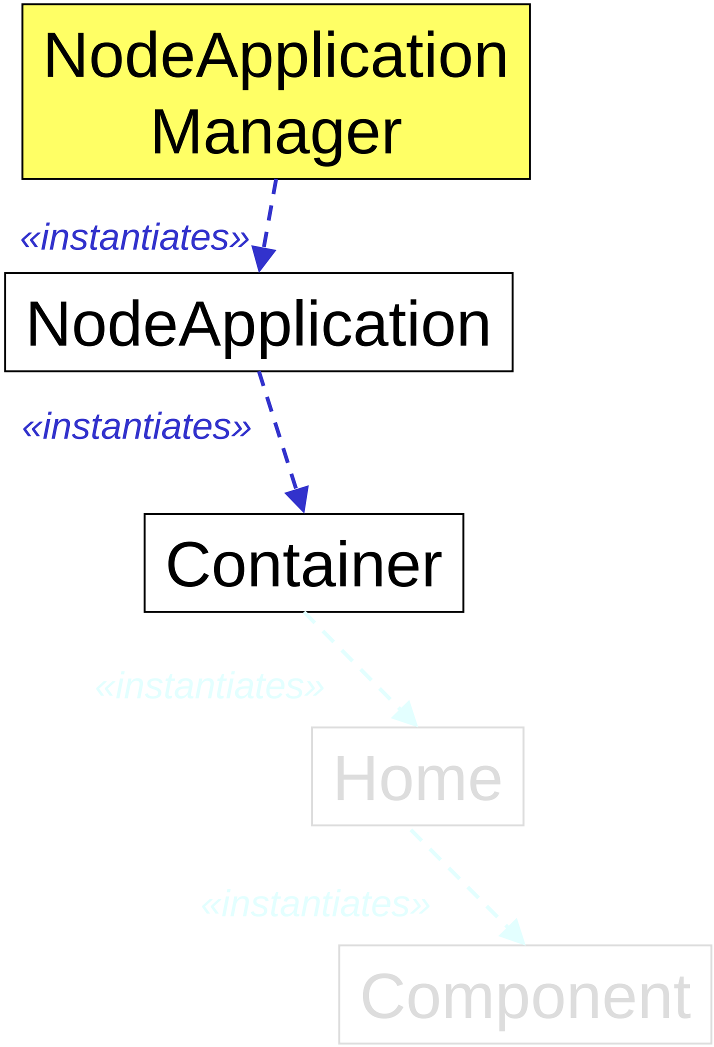
init()
install()
install_home()

**create container**

**Container**

ciao_install_home()
install_servant()
install_component()

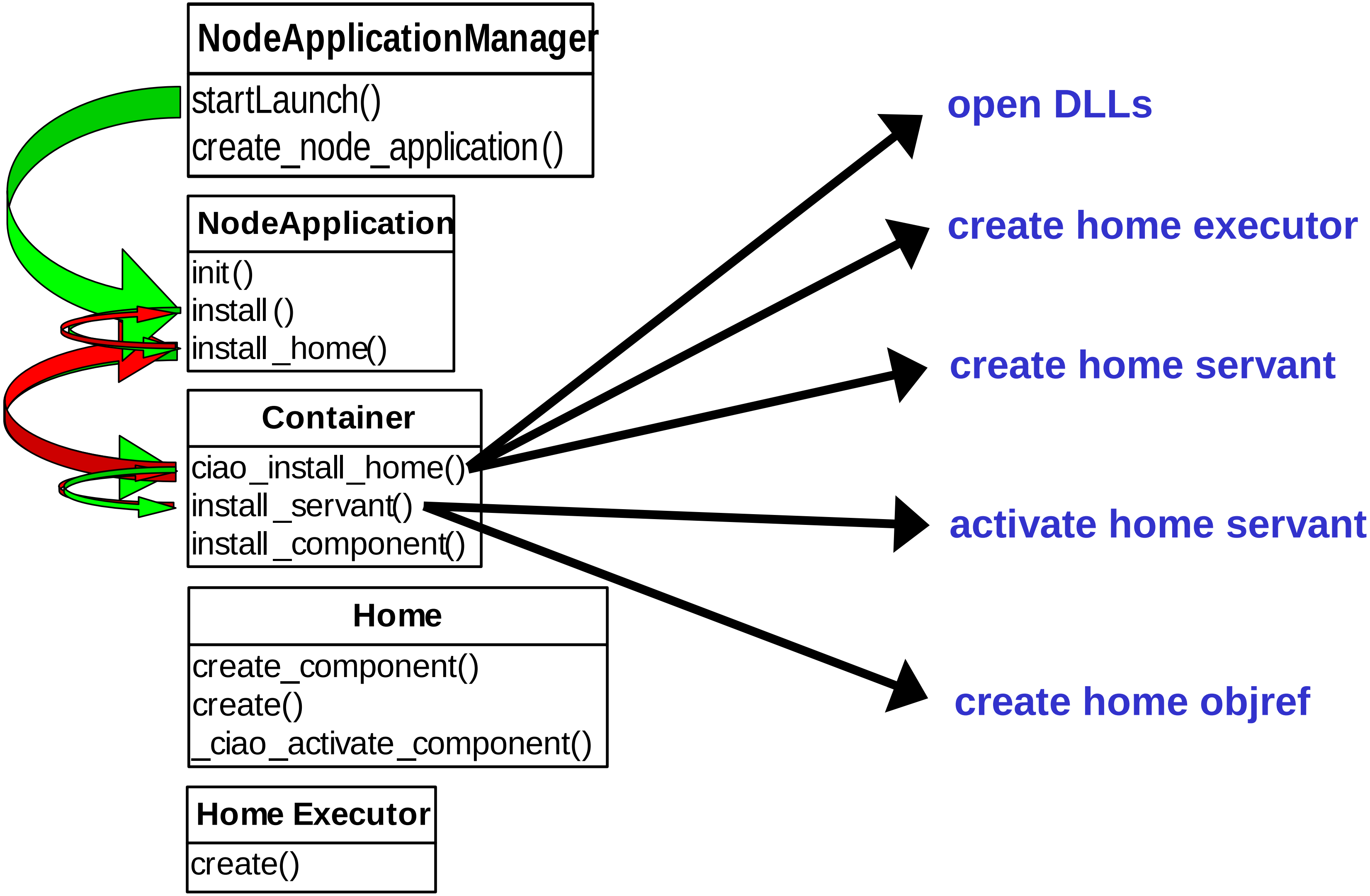
# Deployment & Configuration Process – Step 3



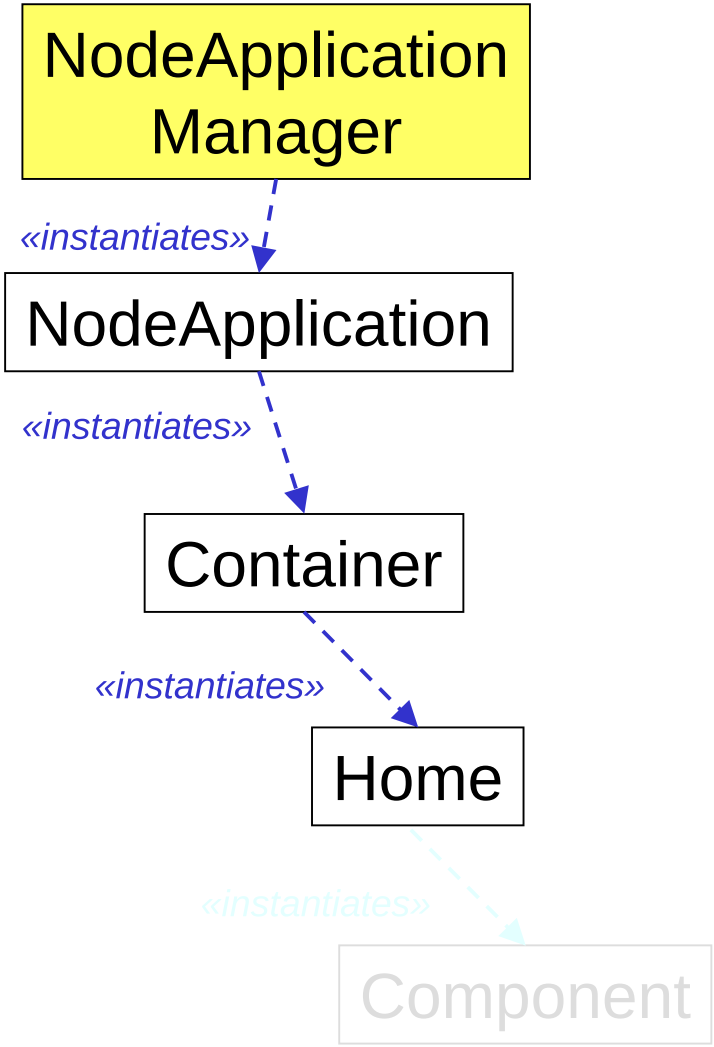
Canonical steps in the application deployment & configuration process (performed by CCM Deployment & Configuration engine):

- Create the *NodeApplication* environment within which containers reside
- Create *containers* for the components
- Create & register *homes* for components
- Create & register the *components* themselves
- Establish *connections* between components

# Creating a Home Executor & Home Servant



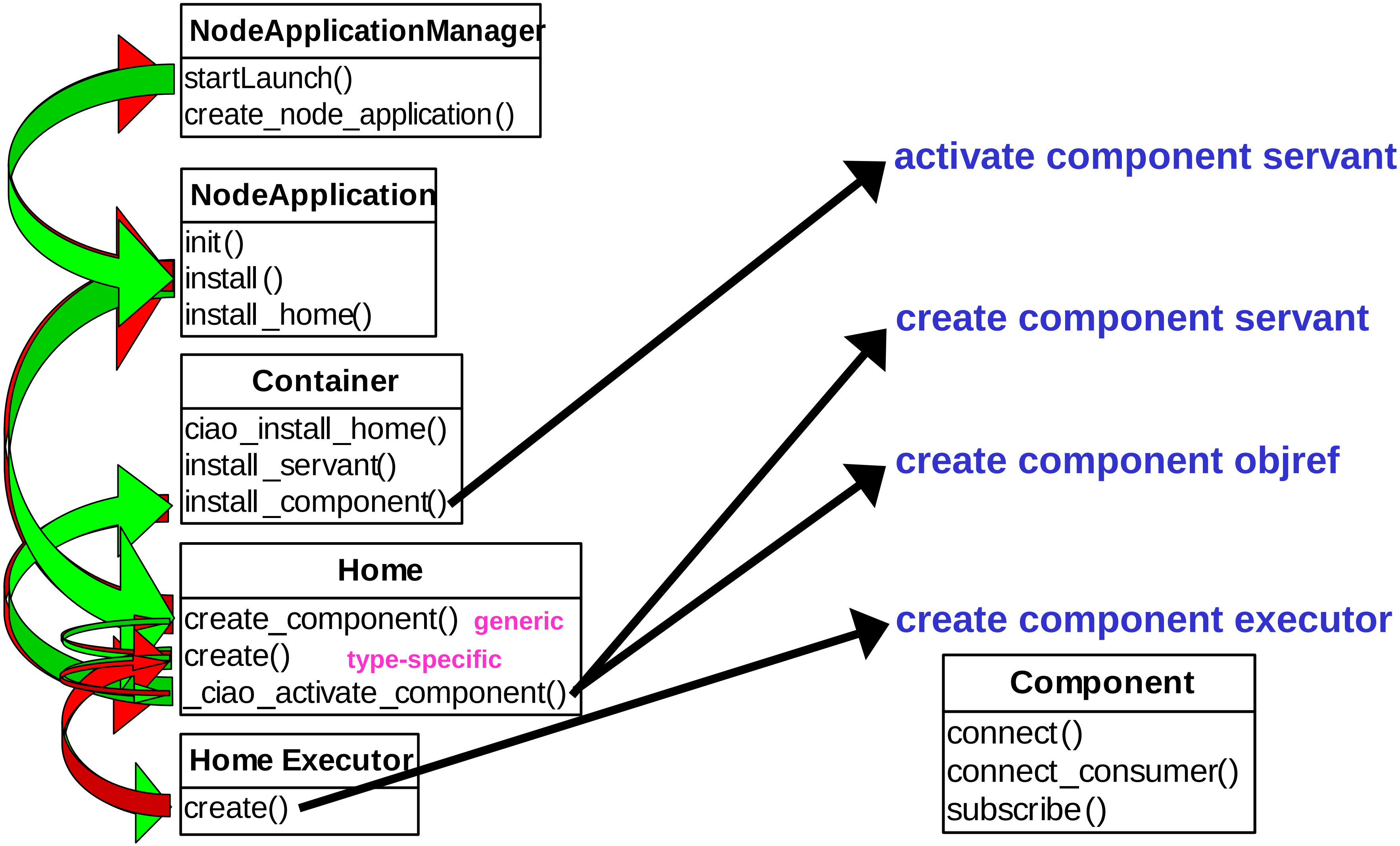
# Deployment & Configuration Process – Step 4



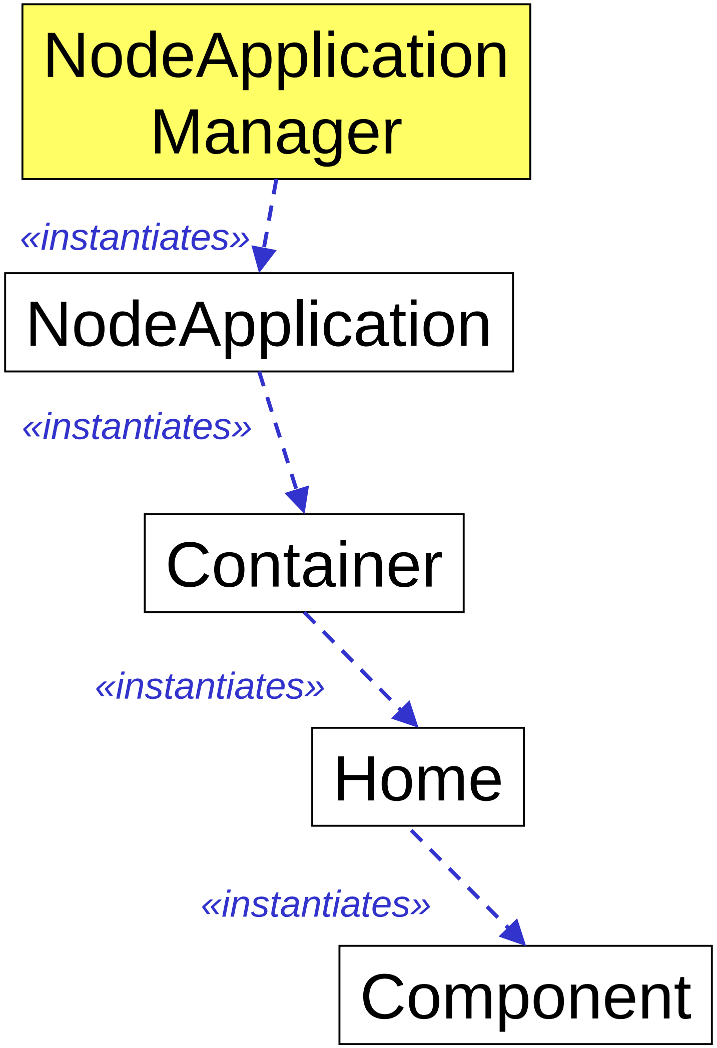
Canonical steps in the application deployment & configuration process (performed by CCM Deployment & Configuration engine):

- Create the *NodeApplication* environment within which containers reside
- Create *containers* for the components
- Create & register *homes* for components
- Create & register the *components* themselves
- Establish *connections* between components

# Creating a Component



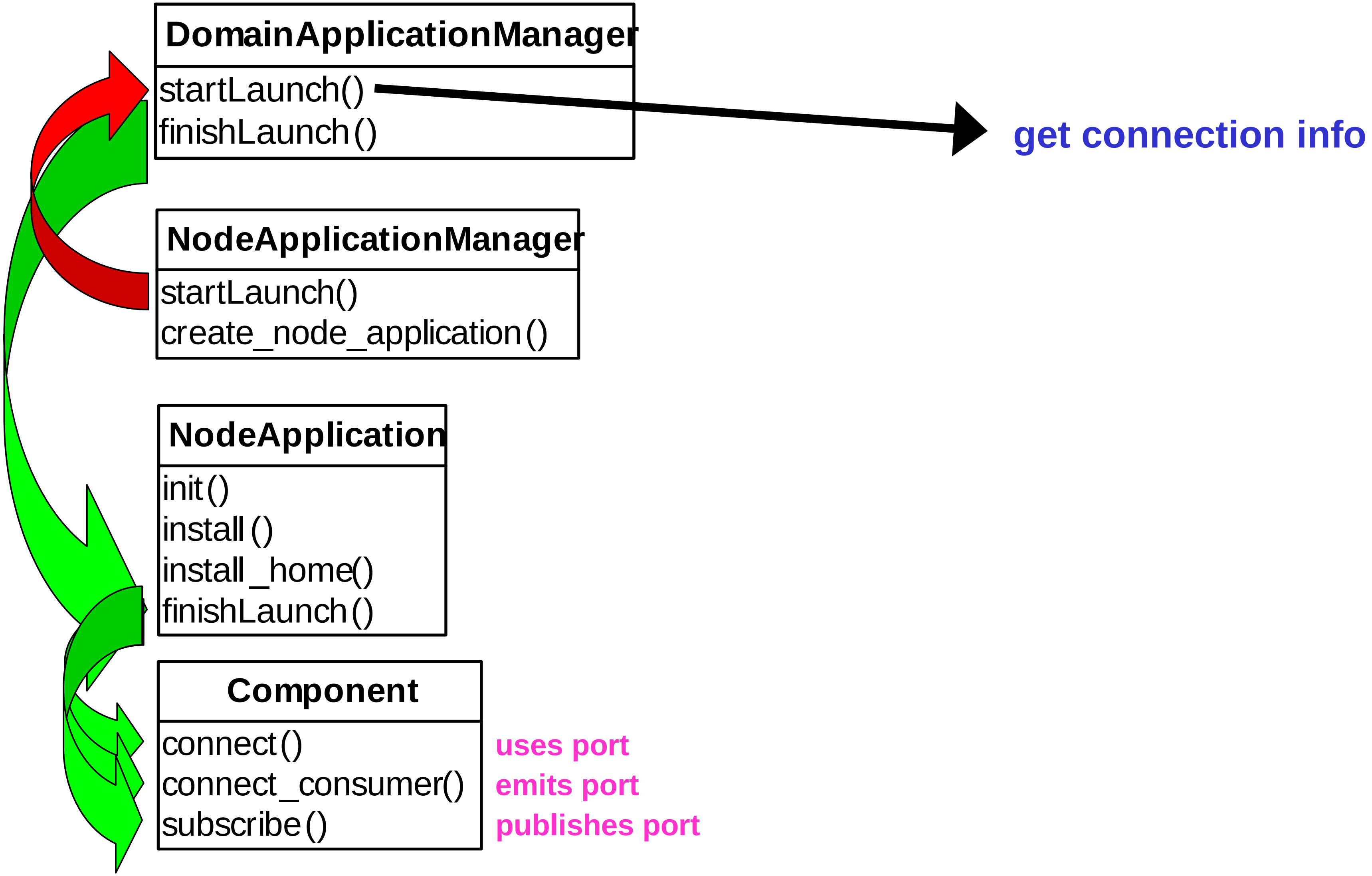
# Deployment & Configuration Process – Step 5



Canonical steps in the application deployment & configuration process (performed by CCM Deployment & Configuration engine):

- Create the *NodeApplication* environment within which containers reside
- Create *containers* for the components
- Create & register *homes* for components
- Create & register the *components* themselves
- Establish *connections* between components

# Establishing Connections



**DomainApplicationManager**

startLaunch()
finishLaunch()

**get connection info**

**NodeApplicationManager**

startLaunch()
create_node_application()

**NodeApplication**

init()
install()
install_home()
finishLaunch()

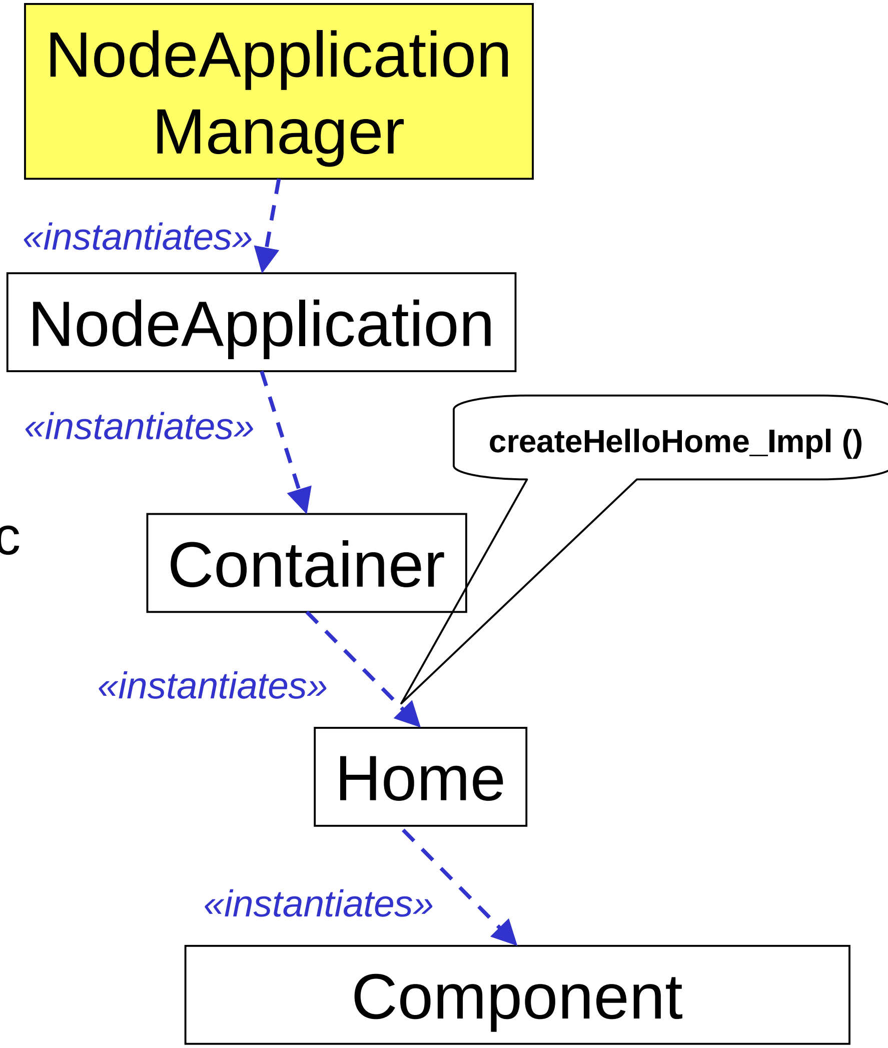
**Component**

connect() **uses port**
connect_consumer() **emits port**
subscribe() **publishes port**

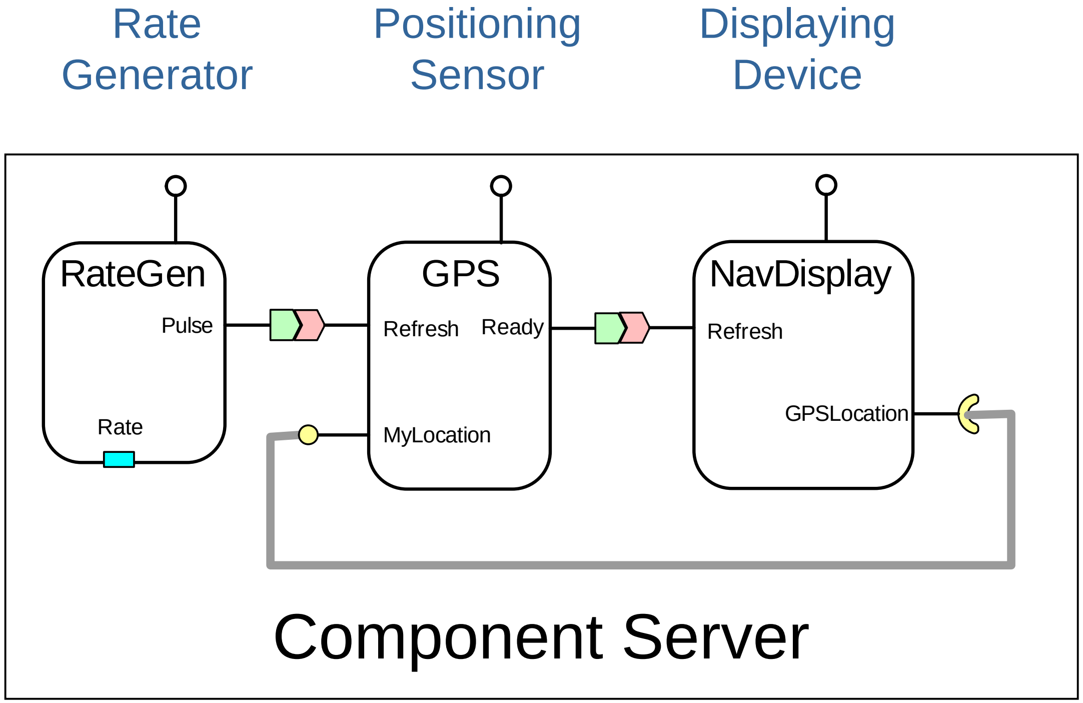
# HelloWorld Component Entry Point Example

```
extern "C" {
Components::HomeExecutorBase_ptr
createHelloHome_Impl (void)
{
  return new
    HelloHome_Exec_Impl;
}
}
```

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



# Implementing Heads Up Display (HUD) Example Executors



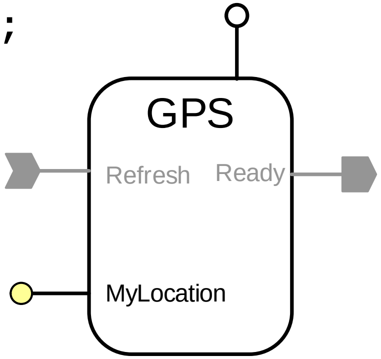
- Component developers must implement
  - Executors for “provided” ports that are invoked by its clients
    - Facets
    - Event sinks
  - Executors that invoke operations on the component’s “required” ports
    - Receptacles
    - Event sources

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

# Implementing HUD Facet Local Interface

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

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

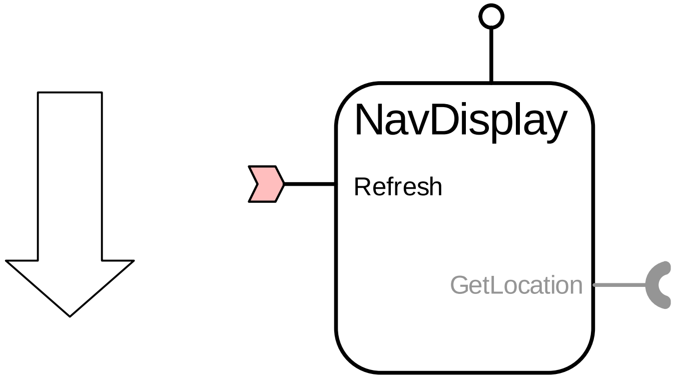


```
// Executor IDL generated by CIDL compiler
local interface CCM_position : position {};
local interface GPS_Exec :
  CCM_GPS,
  Components::SessionComponent
{
  CCM_position get_MyLocation ();
};
```

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

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

# HUD Component Event Sinks

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



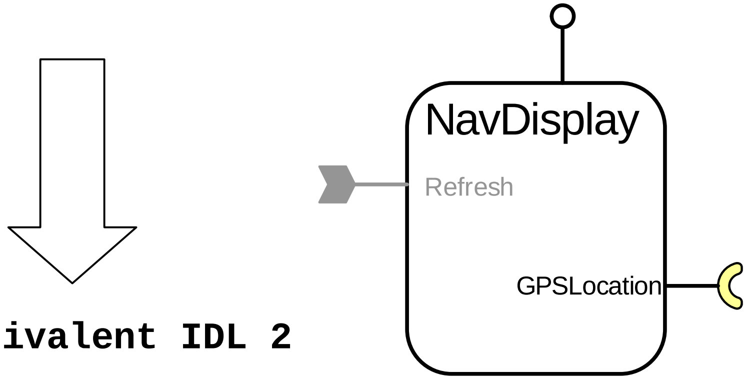
```
// Equivalent IDL 2
interface NavDisplay :
  Components::CCMObject
{
  …
  tickConsumer get_consumer_Refresh ();
  …
};
```

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

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

# Using HUD Receptacle Connections

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

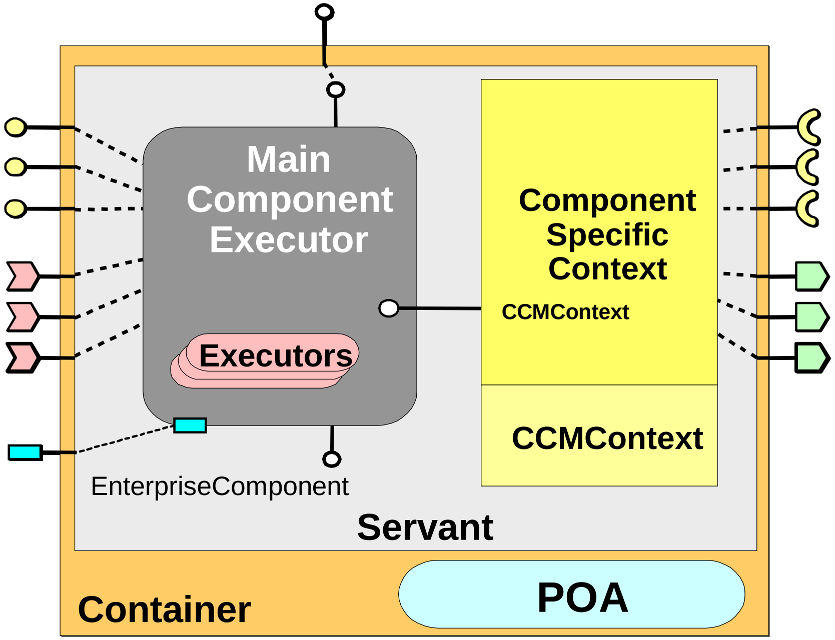


```
// Equivalent IDL 2
interface NavDisplay :
  Components::CCMObject
{
 …
 void connect_GPSLocation (in position c);
 position disconnect_GPSLocation();
 position get_connection_GPSLocation ();
 …
};
```

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

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

# Initializing HUD Component-specific Context



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

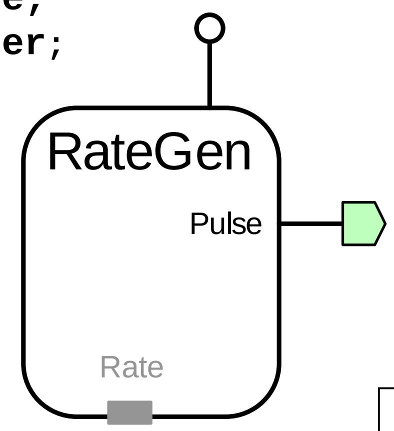
```
class NavDisplay_Exec_Impl :
  public virtual NavDisplay_Exec,
  public virtual CORBA::LocalObject {
private:
  CCM_NavDisplay_Context_var context_;
public:
  …
  // Called back by container
  void set_session_context
    (Components::SessionContext_ptr c) {
    this->context_ =
      CCM_NavDisplay_Context::_narrow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

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

# Pushing Events from a HUD Component

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

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



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

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

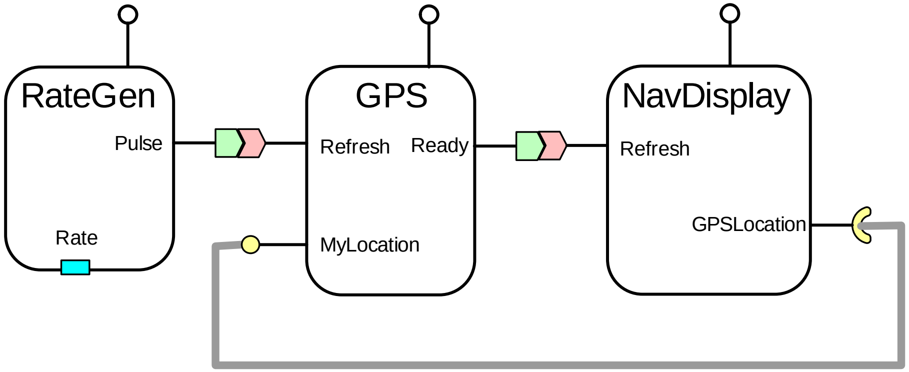
# Summary of Server OMG IDL Mapping Rules

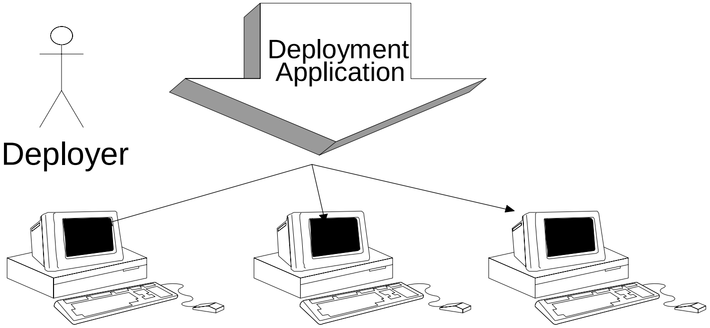
- A **`component`** type is mapped to three **`local`** interfaces that correspond to different component roles/ports
  - The *component executor interface*
    - Inherits from **`Components::EnterpriseComponent`** & provides operations for attributes, supported interfaces, & receiving events
  - A *facet executor interface*
    - Operations to obtain facet object references
  - The *component-specific context interface*
    - Operations to publish events & access component receptacles

- A **`home`** type is mapped to four **`local`** interfaces
  - An *explicit executor* interface for user-defined operations
    - Inherits from **`Components::HomeExecutorBase`**
  - An *implicit executor* interface for **`create()`** operation
  - A *main executor* interface inheriting from both previous interfaces
  - A *composition executor* interface inheriting from the main executor interface

# Component Packaging, Assembly, & Deployment

# Overview of Configuration & Deployment Process





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

# Component Configuration Problem

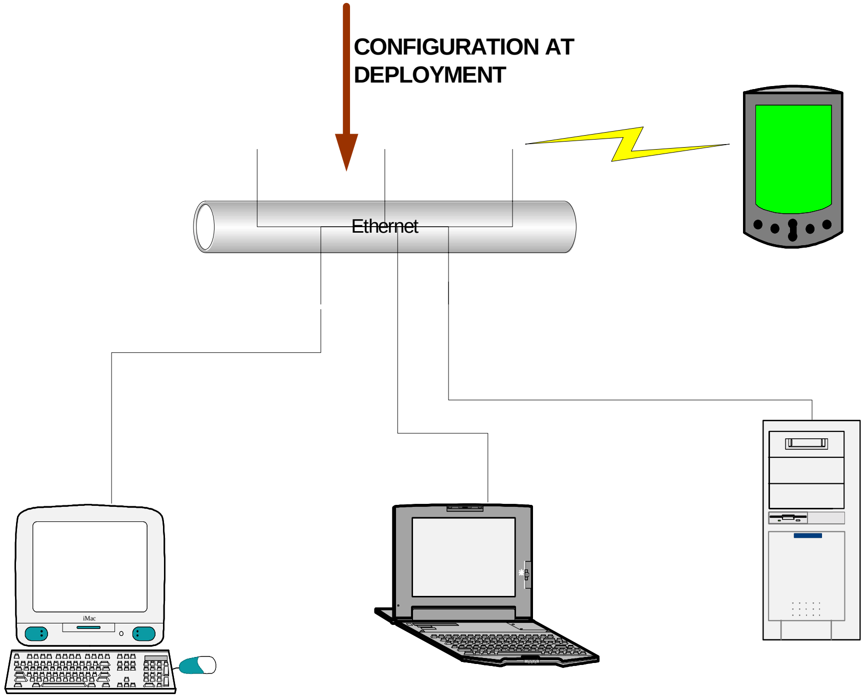
Component middleware & applications are characterized by a large *configuration space* that maps known variations in the *application requirements space* to known variations in the *solution space*

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

# CCM Configuration Concept & Solution

Concept:

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



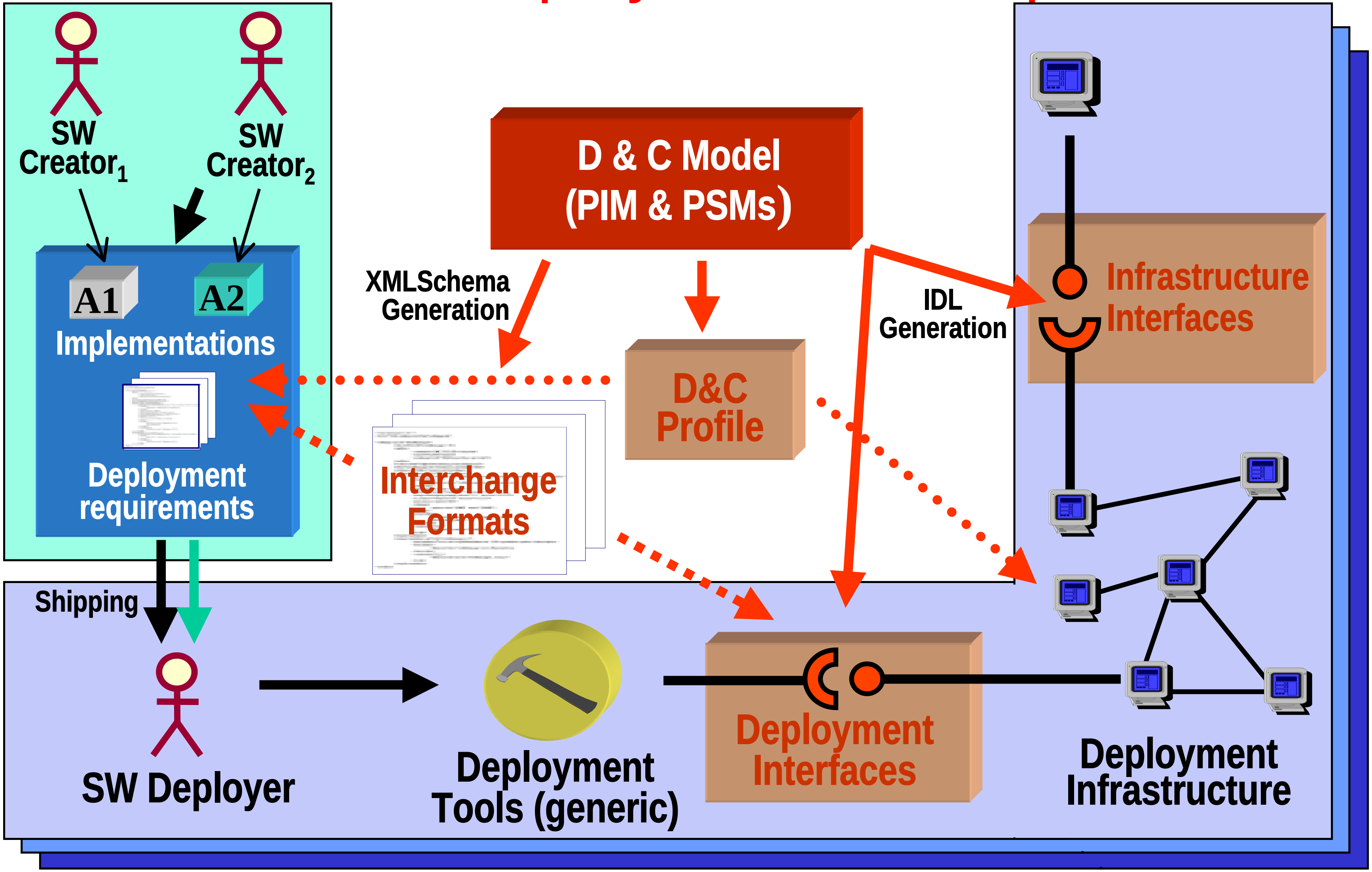
Solution:

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

# Component Deployment Problem

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

# CCM Deployment Concept



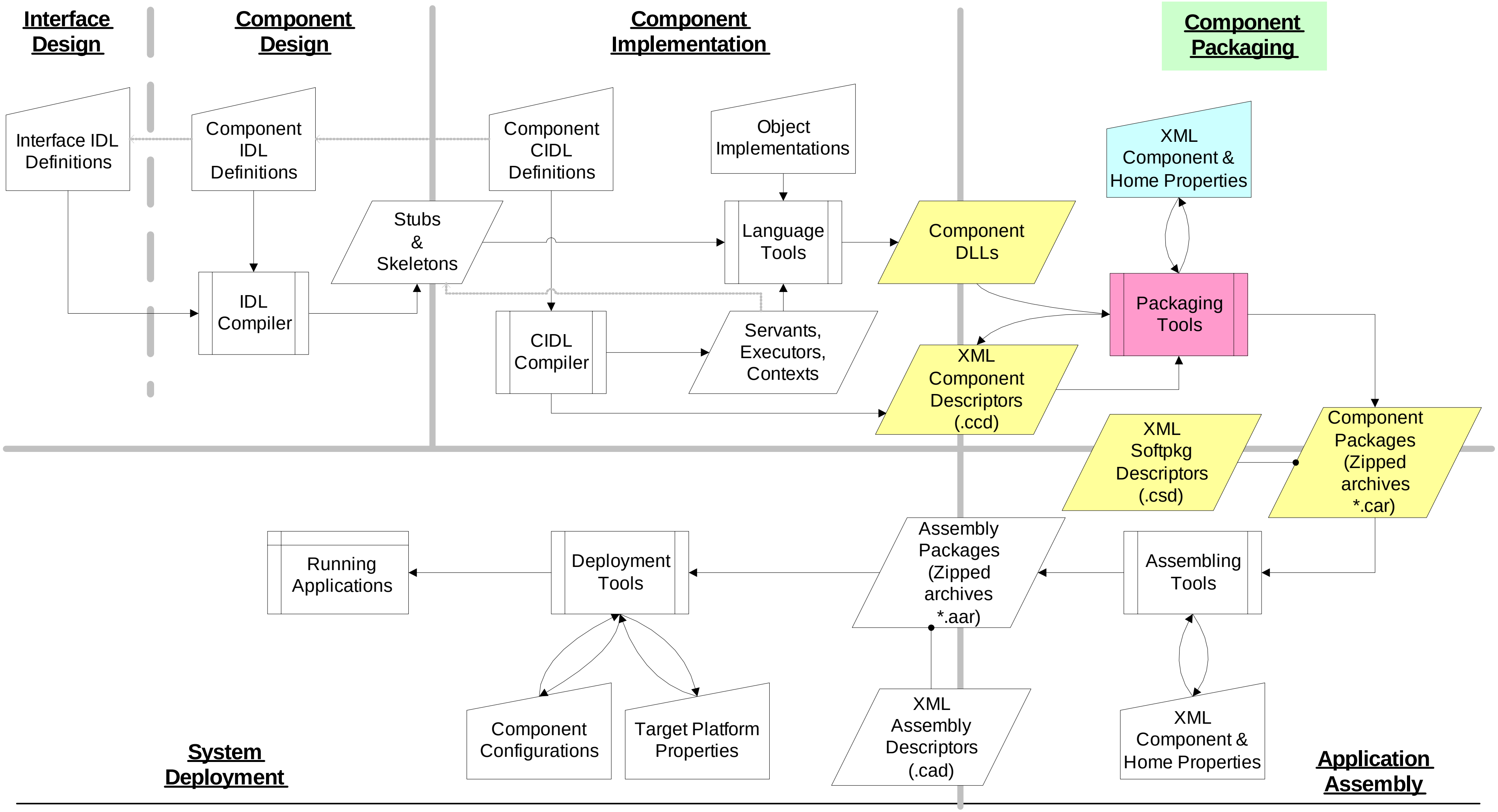
# CCM Deployment Solution

- **Well-defined exchange format**
  - – Defines what a software vendor delivers
  - – Requires “off-line” data format that can be stored in files
- **Well-defined interfaces**
  - – Infrastructure to install, configure, & deploy software
  - – Requires “on-line” data format that can be passed to/from interfaces
- **Well-defined software metadata model**
  - – Annotate software & hardware with interoperable, vendor-independent, deployment-relevant information
  - – Generate “on-line” & “off-line” data formats from model

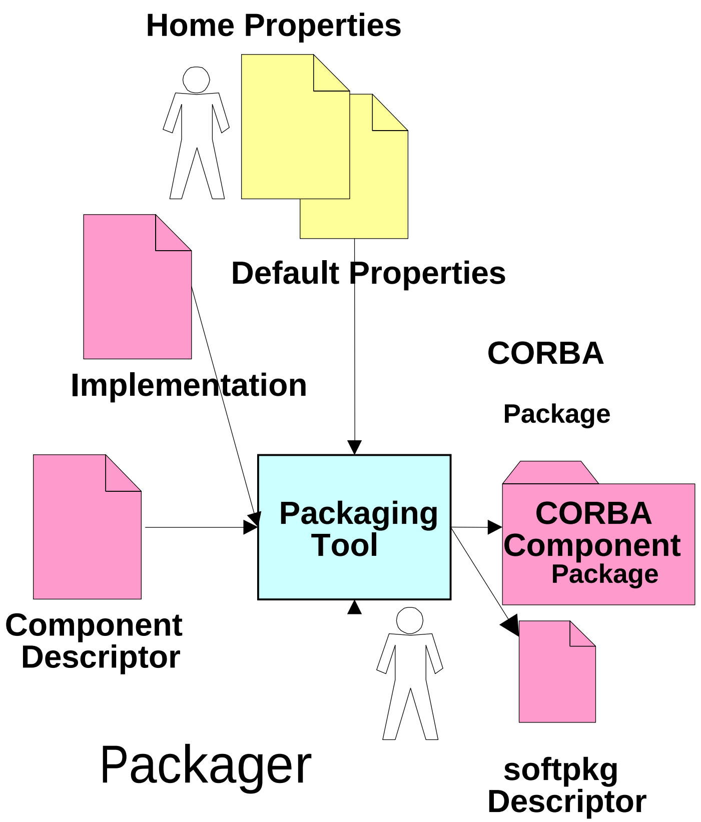
# Old OMG Packaging & Deployment Specification

# Component Packaging Stage

Packaging: bundling a component implementation with associate metadata


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
XML Component & Home Properties
Stubs & Skeletons
Language Tools
Component DLLs
IDL Compiler
Packaging Tools
CIDL Compiler
Servants, Executors, Contexts
XML Component Descriptors (.ccd)
XML Softpkg Descriptors (.csd)
Component Packages (Zipped archives *.car)
Running Applications
Deployment Tools
Assembly Packages (Zipped archives *.aar)
Assembling Tools
Component Configurations
Target Platform Properties
XML Assembly Descriptors (.cad)
XML Component & Home Properties
System Deployment
Application Assembly

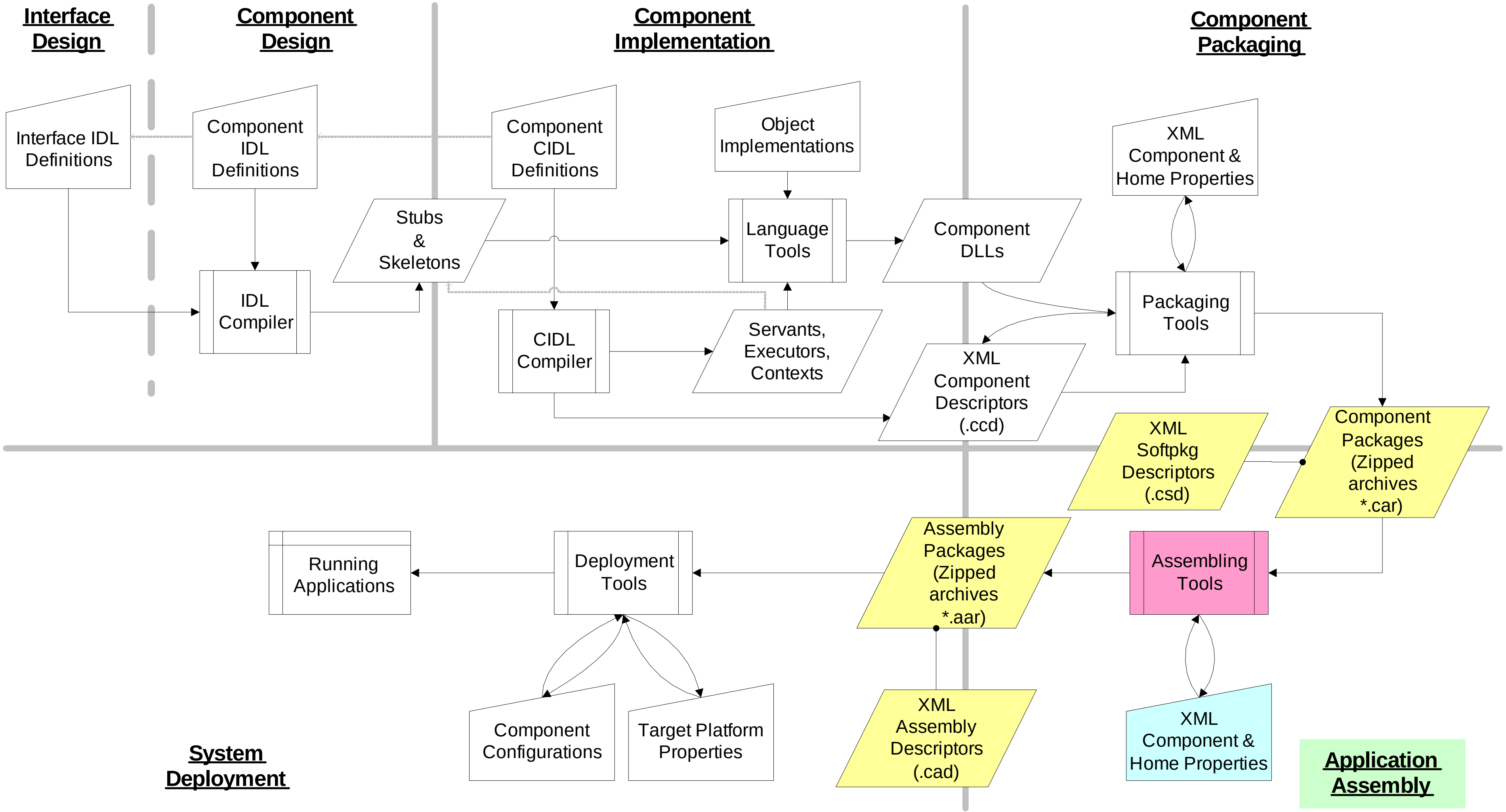
# Component Packages



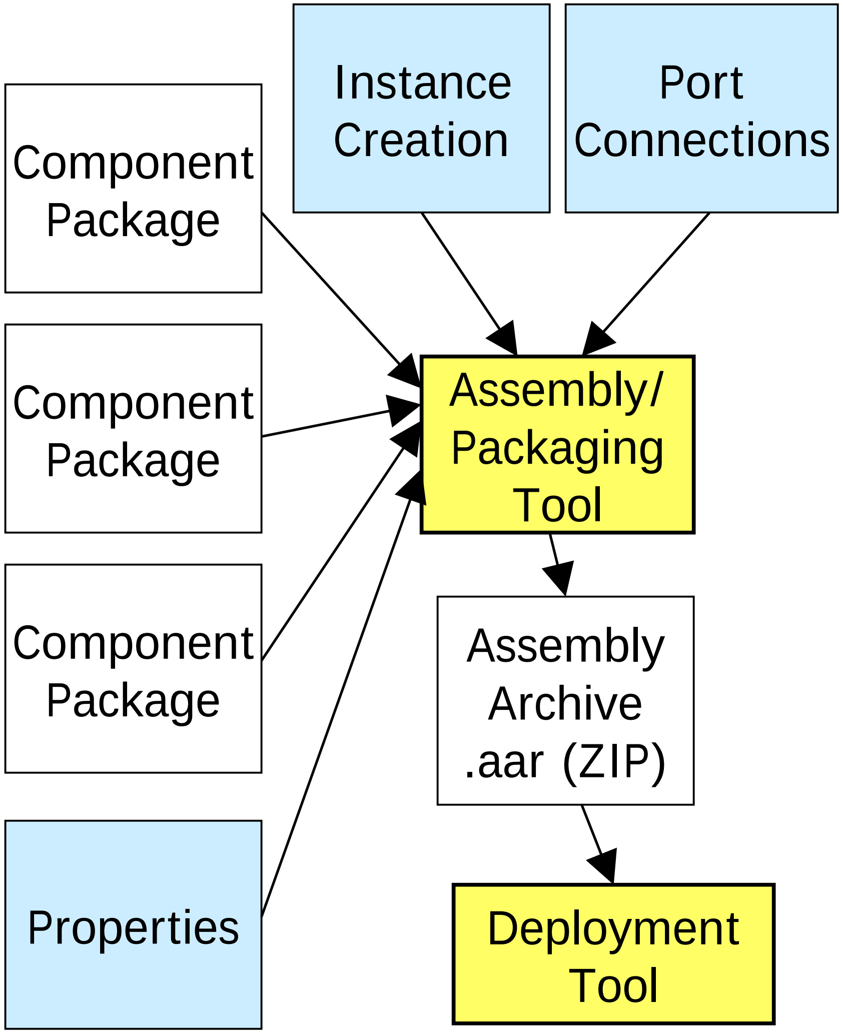
- Goals
  - Configure components, containers, servers
  - Extract these aspects into metadata
- That's a lot of stuff to be bundled together & moved around
- "Classic" CORBA: No standard means of configuration, distribution, & deployment
- Packaging of components
  - Components are packaged into a self-descriptive package as a compressed archive
- XML descriptors provide metadata that describe
  - The content of a package
  - The capability of components
  - The dependencies to other software artifacts
    - e.g., Other components, 3rd party DLLs, & Value factories

# Application Assembling Stage

**Assembly: component packages & metadata that specify composition of application**

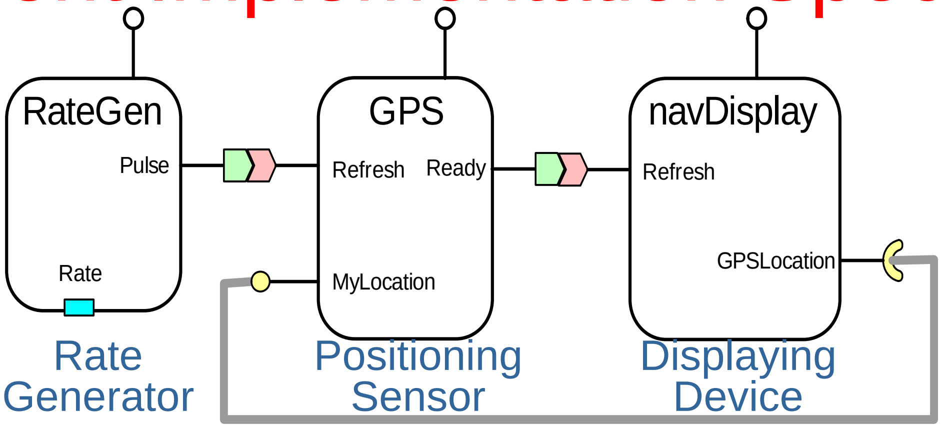

Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
XML Component & Home Properties
Stubs & Skeletons
Language Tools
Component DLLs
IDL Compiler
CIDL Compiler
Servants, Executors, Contexts
Packaging Tools
XML Component Descriptors (.ccd)
XML Softpkg Descriptors (.csd)
Component Packages (Zipped archives *.car)
Running Applications
Deployment Tools
Assembly Packages (Zipped archives *.aar)
Assembling Tools
Component Configurations
Target Platform Properties
XML Assembly Descriptors (.cad)
XML Component & Home Properties
System Deployment
Application Assembly

# Component Assembling



- Goals
  - Configure components, containers, servers, and applications
  - Extract these aspects into metadata
  - Provide higher level of modeling
- "Classic" CORBA: No standard means of
  - Configuration
  - Distribution
  - Deployment
- An assembly descriptor specifies:
  - Component implementations
  - Component/home instantiations
  - Interconnections

# Component Implementation Specifications



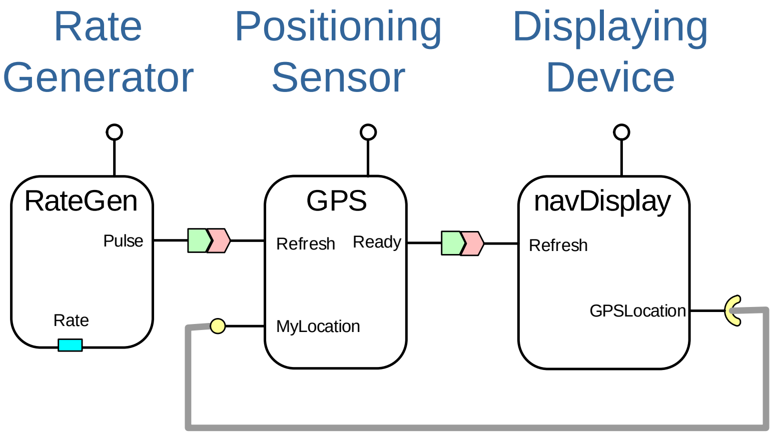
```
<!– Assembly descriptors associate components with implementations -->
<!- in software packages defined by softpkg descriptors (*.csd) files -->
<componentfiles>
    <componentfile id=“com-RateGen">
      <fileinarchive name=“RateGen.csd"/>
    </componentfile>

    <componentfile id=“com-GPS">
      <fileinarchive name=“GPS.csd"/>
    </componentfile>

    <componentfile id=“com-Display">
      <fileinarchive name=“NavDisplay.csd"/>
    </componentfile>

 </componentfiles>
```

# Component Home/Instances Installation Specifications

```
<!– Instantiating component homes/instances -->

<partitioning>
  <hostcollocation>
    ...

    <homeplacement id=“a_RateGenHome">
      <componentfileref idref=“com-RateGen"/>
      <componentinstantiation id=“a_RateGen">
        <componentproperties>
          <fileinarchive name=“NavRateGen.cpf"/>
        </componentproperties>
      </componentinstantiation>
    </homeplacement>
    ...
    <destination>A_Remote_Host</destination>
  </hostcollocation>
</partitioning>
```

- An assembly descriptor specifies how & where homes & components should be instantiated
- A component property file (.cpf) can be associated with a home or a component instantiation to override default component properties

# Interconnection Specification

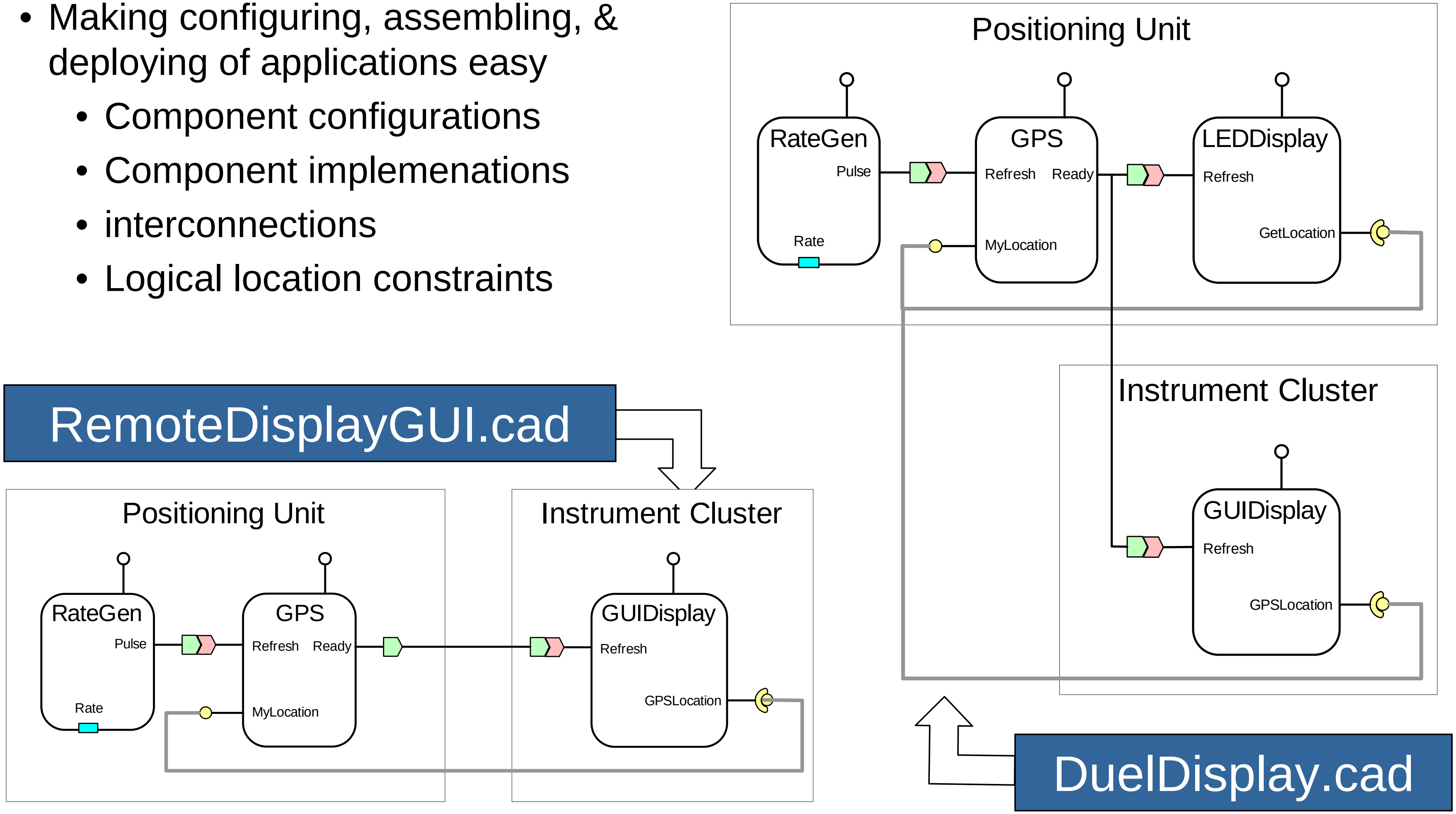


```
<connections>
  ...
  <connectinterface>
    <usesport>
      <usesidentifier>GPSPosition</usesidentifier>
      <componentinstantiationref idref=“a_NavDisplay"/>
    </usesport>
    <providesport>
      <providesidentifier>
         MyLocation
      </providesidentifier>
      <componentinstantiationref idref=“a_GPS"/>
    </providesport>
  </connectinterface>
  <connectevent>
    <consumesport>
      <consumesidentifier>Refresh</consumesidentifier>
      <componentinstantiationref idref=“a_GPS"/>
    </consumesport>
    <publishesport>
      <publishesidentifier>
         Pulse
      </publishesidentifier>
      <componentinstantiationref
             idref=“a_RateGen"/>
    </publishesport>
  </connectevent>
  ...
</connections>
```

- Assembly descriptors also specify how component instances are connected together

# Two Deployment Examples

- Making configuring, assembling, & deploying of applications easy
  - Component configurations
  - Component implemenations
  - interconnections
  - Logical location constraints

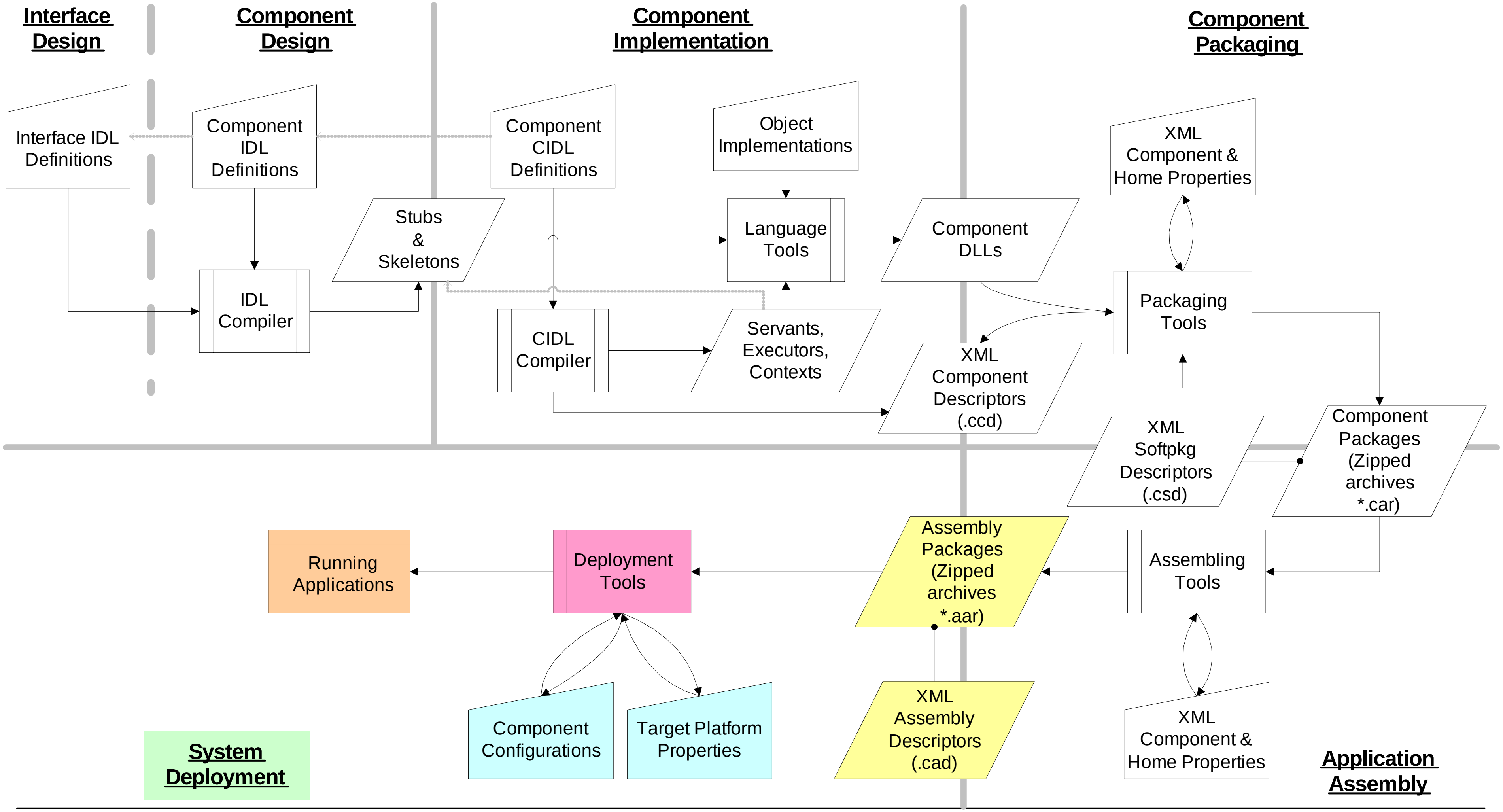


RemoteDisplayGUI.cad

DuelDisplay.cad

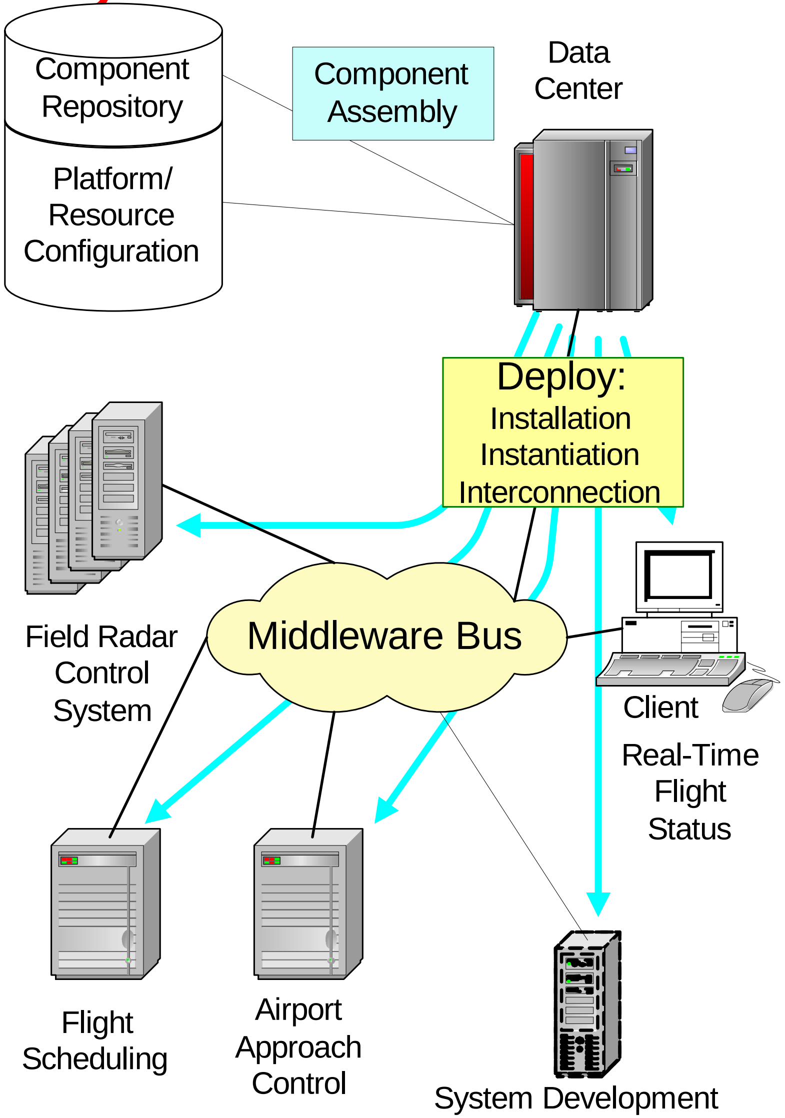
# Deployment Stage

Deployment: Realization of a single component or an assembly specification


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
XML Component & Home Properties
Stubs & Skeletons
Language Tools
Component DLLs
IDL Compiler
CIDL Compiler
Servants, Executors, Contexts
Packaging Tools
XML Component Descriptors (.ccd)
XML Softpkg Descriptors (.csd)
Component Packages (Zipped archives *.car)
Running Applications
Deployment Tools
Assembly Packages (Zipped archives *.aar)
Assembling Tools
Component Configurations
Target Platform Properties
XML Assembly Descriptors (.cad)
XML Component & Home Properties
System Deployment
Application Assembly

# Application Deployment

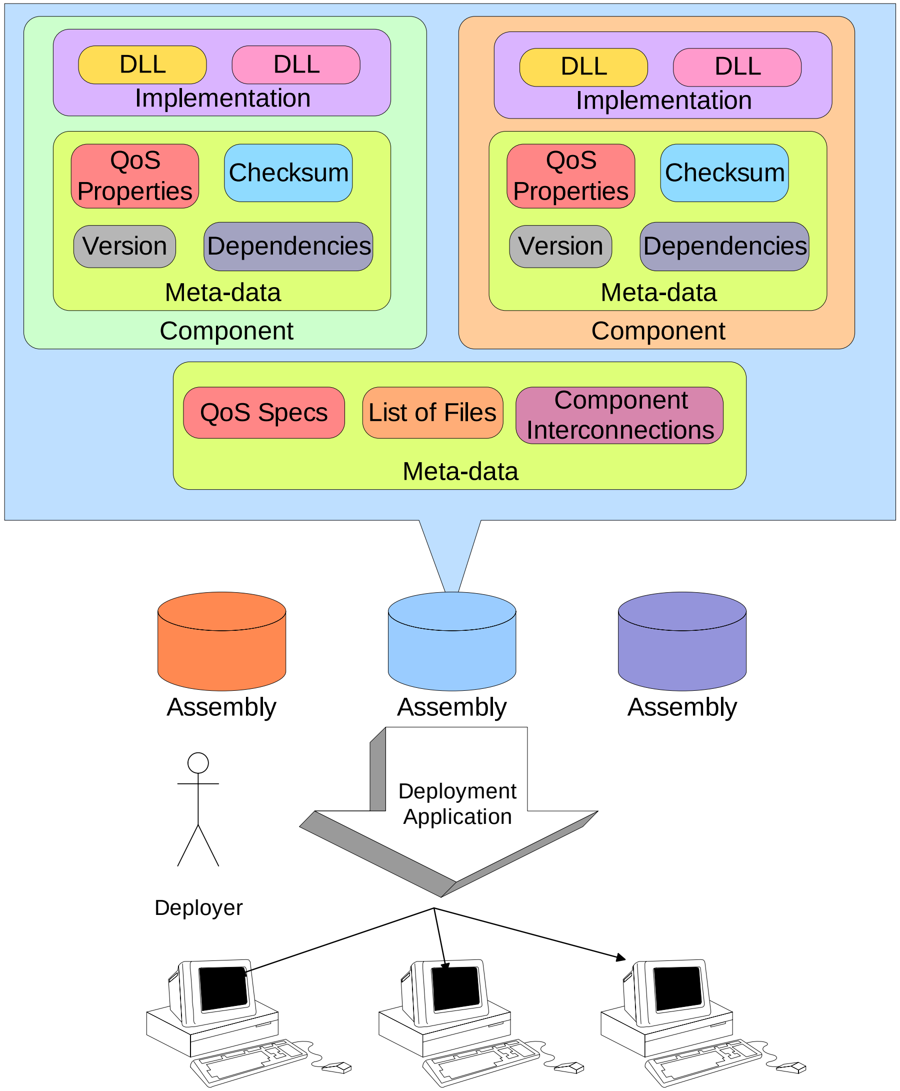
- Deployment tools
  - Have knowledge of target platforms
  - Map locations in assembly to physical nodes
  - Manage available resources for applications
  - Use standard CCM interfaces defined in module **Components::Deployment** to realize an assembly



# New OMG Deployment & Configuration Specification

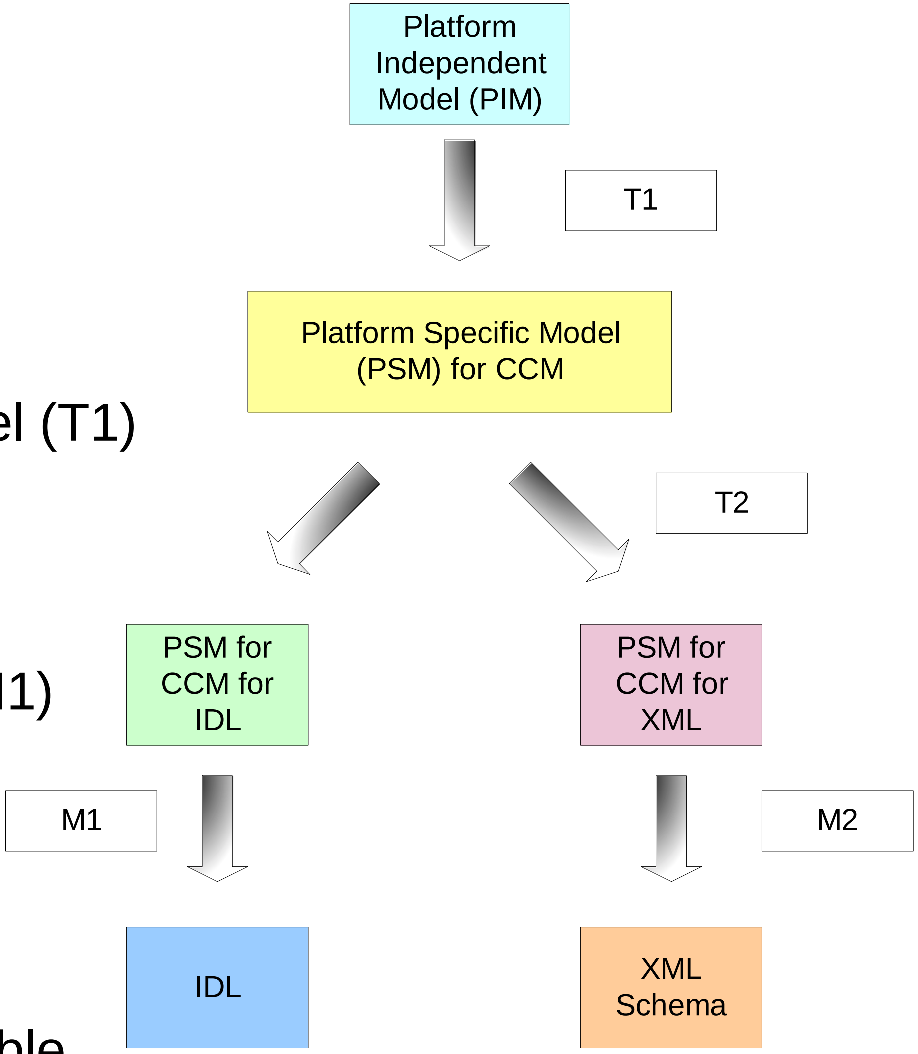
# CCM Deployment & Configuration (D&C) Spec

- "D&C" spec was adopted by OMG in 2003
- Intended to replace *Packaging & Deployment* chapter of CCM (CORBA 3.0) specification
- Supports …
  - Hierarchical assemblies
  - Resource management
  - QoS characteristics
  - Automated deployment
  - Vendor-independent deployment infrastructure



# D&C & Model-Driven Architecture

- D&C is specified using a platform-independent model
  - Defines "deployment" model
  - Independent of CORBA & CCM (specified in UML)
- Can be 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 & M2


Platform Independent Model (PIM)
T1
Platform Specific Model (PSM) for CCM
T2
PSM for CCM for IDL
PSM for CCM for XML
M1
M2
IDL
XML Schema

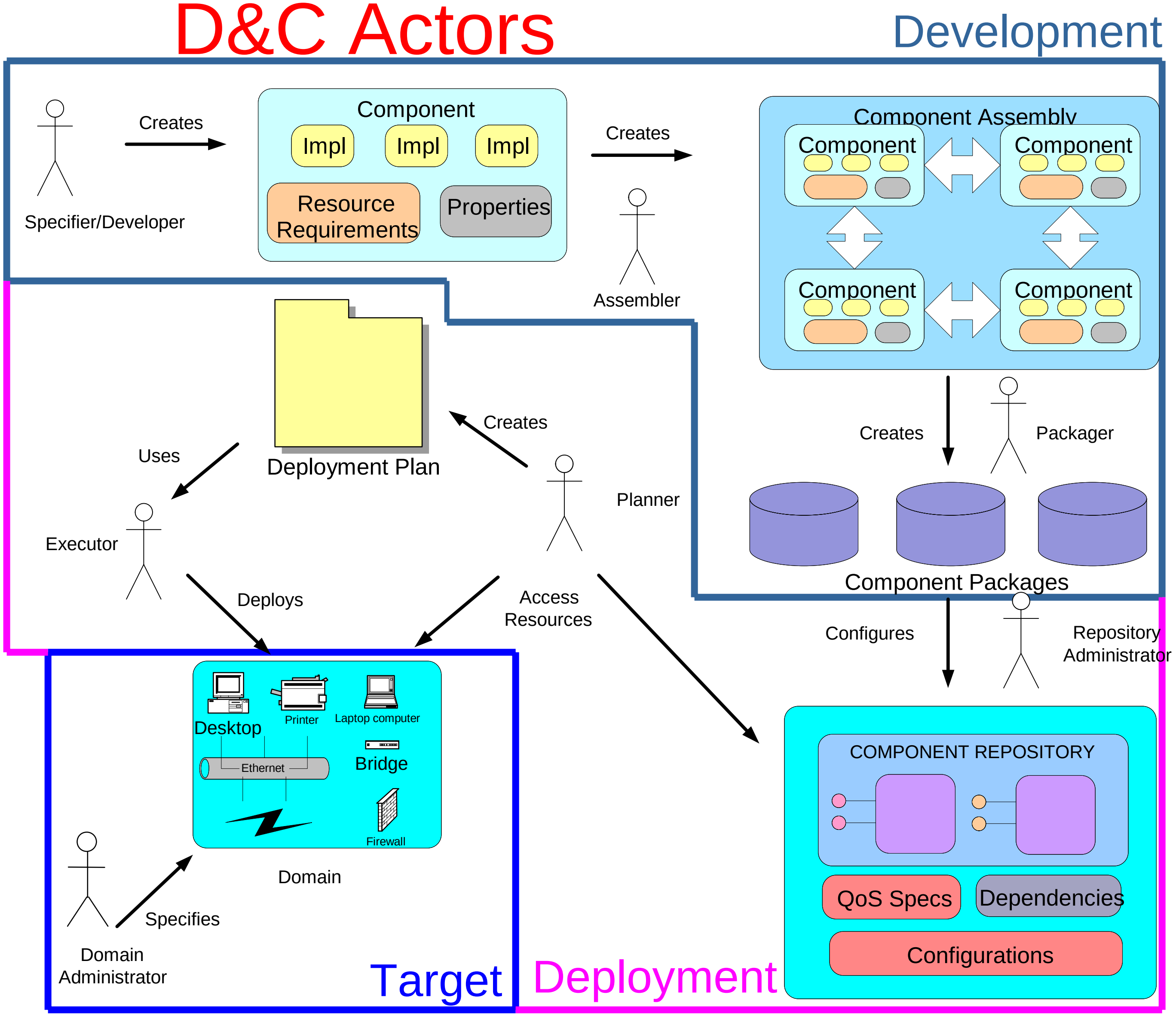
# Deployment & Configuration “Segments”

| PIM | Data Model | Run-time Model |
|---|---|---|
| Component Software | Metadata to describe component-based applications & their requirements | Repository Manager interfaces to browse, store, & retrieve such metadata |
| Target | Metadata to describe heterogeneous distributed systems & their capabilities | Target Manager interfaces to collect & retrieve such metadata & commit resources |
| Execution | Metadata to describe a specific deployment plan for an application into a distributed system | Execution Manager interfaces to prepare environment, execute deployment plan on target, manage lifecycle |

- Data model
  - Metadata, usually in XML format
- Run-time model
  - Deployment interfaces (similar to CORBA services)

# D&C Actors

- Different stages & different actors
  - **Development**
    - *Specifier/ Developer*
    - *Assembler*
    - *Packager*
  - **Target**
    - *Domain Administrator*
  - **Deployment**
    - *Repository Administrator*
    - *Planner*
    - *Executor*
- Actors are abstract
  - Usually humans & software tools



**These actors & stages are simply making explicit existing processes**

# CCM Development & Deployment Phases

***Specification & Implementation***

– Defining, partitioning, & implementation application functionality as standalone components

***Packaging***

– Bundling a suite of software binary modules & metadata representing application components

***Installation***

– Populating a repository with the packages required by the application

***Configuration***

– Configuring the packages with the appropriate parameters to satisfy the functional & systemic requirements of an application without constraining to any physical resources

***Planning***

– Making appropriate deployment decisions including identifying the entities, such as CPUs, of the target environment where the packages will be deployed

***Preparation***

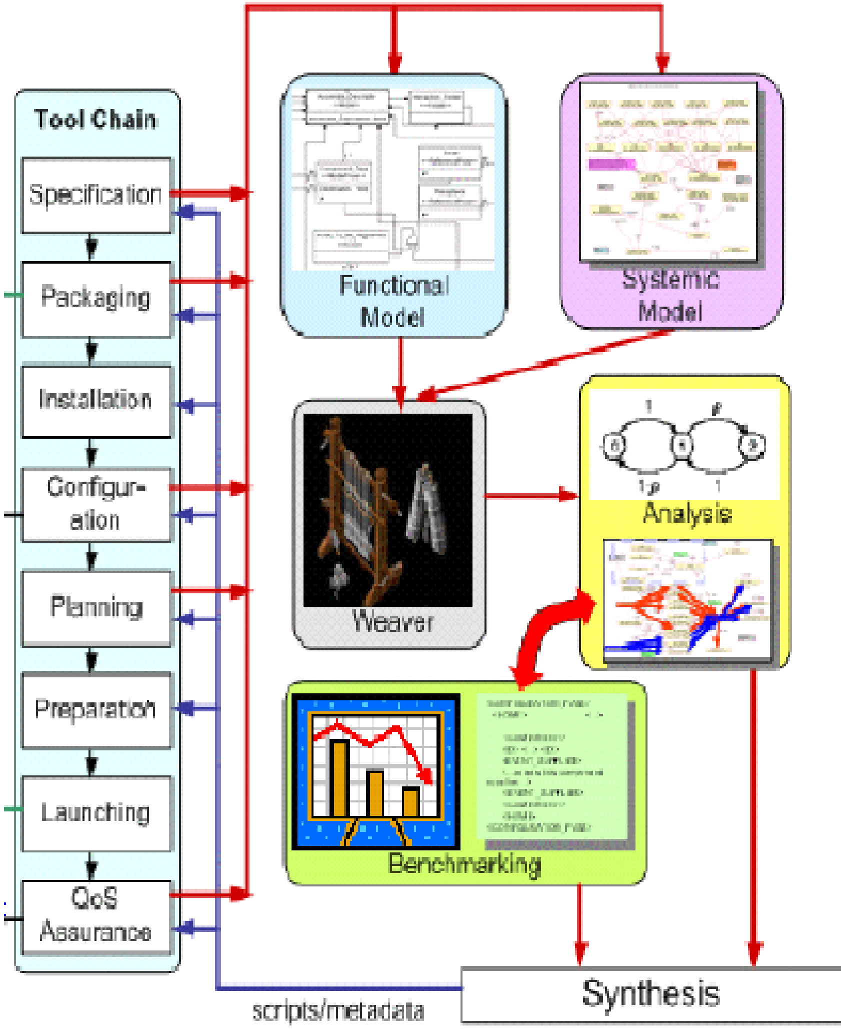
– Moving the binaries to the identified entities of the target environment

***Launching***

– Triggering the installed binaries & bringing the application to a ready state

***QoS Assurance & Adaptation***

– Runtime reconfiguration & resource management to maintain end-to-end QoS

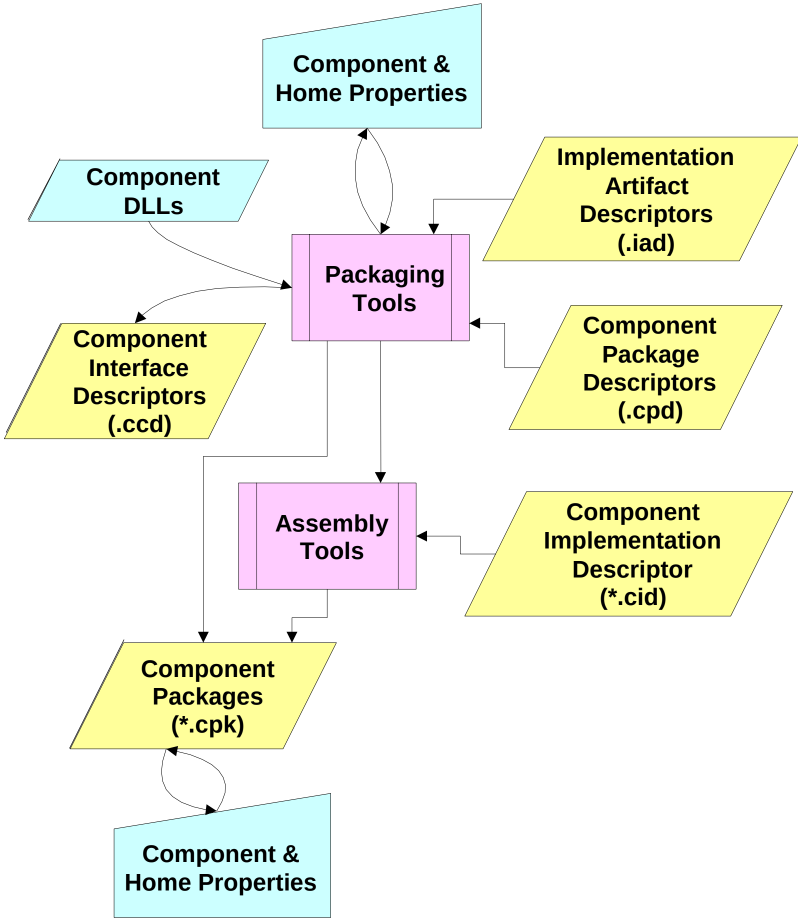


OMG Deployment & Configuration (D&C) specification (ptc/05-01-07)

# Metadata Produced/Used by D&C Tools (1/2)

- **Component Interface Descriptor (.ccd)**
  - –Describes the interface, ports, & properties of one component
- **Implementation Artifact Descriptor (.iad)**
  - –Describes the implementation artifacts (e.g., DLLs, OS, etc.) of one component
- **Component Package Descriptor (.cpd)**
  - –Describes multiple alternative implementations of one component
- **Package Configuration Descriptor (.pcd)**
  - –Describes a configuration of a component package
- **Top-level Package Descriptor (package.tpd)**
  - –Describes the top-level component package in a package (.cpk)
- **Component Implementation Descriptor (.cid)**
  - –Describes a specific implementation of a component interface
  - –Implementation can be either monolithic- or assembly-based
  - –Contains subcomponent instantiations in case of assembly based implementations
  - –Contains interconnection information between components
- **Component Packages (.cpk)**
  - –A component package can contain a single component
  - –A component package can also contain an assembly

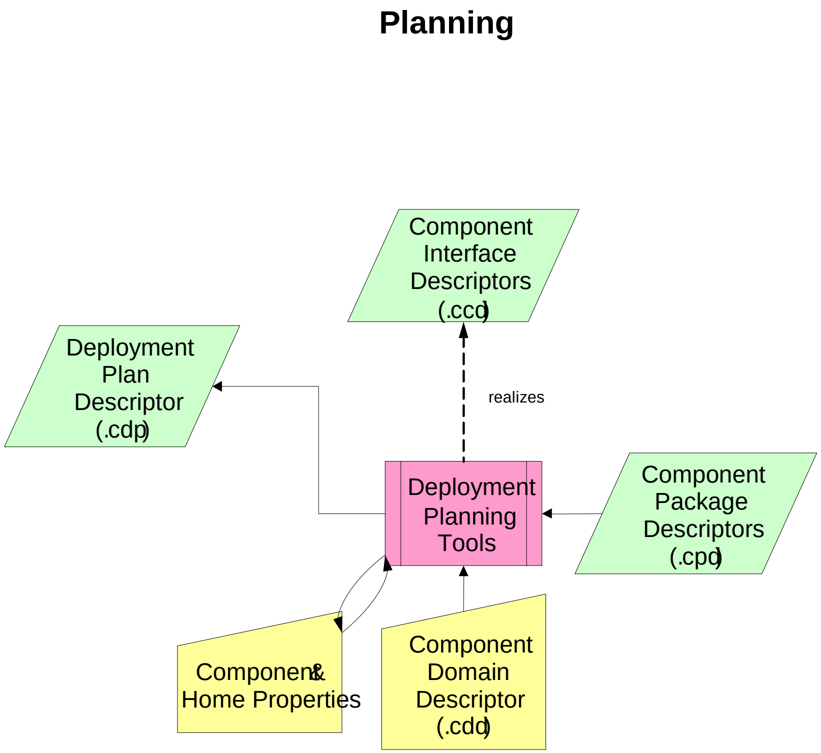
**Component Packaging & Assembly**


Component & Home Properties
Component DLLs
Implementation Artifact Descriptors (.iad)
Packaging Tools
Component Interface Descriptors (.ccd)
Component Package Descriptors (.cpd)
Assembly Tools
Component Implementation Descriptor (*.cid)
Component Packages (*.cpk)
Component & Home Properties

**These files could be coalesced into a smaller # (i.e., 1 file) in actual systems**

# Metadata Produced/Used by D&C Tools (2/2)

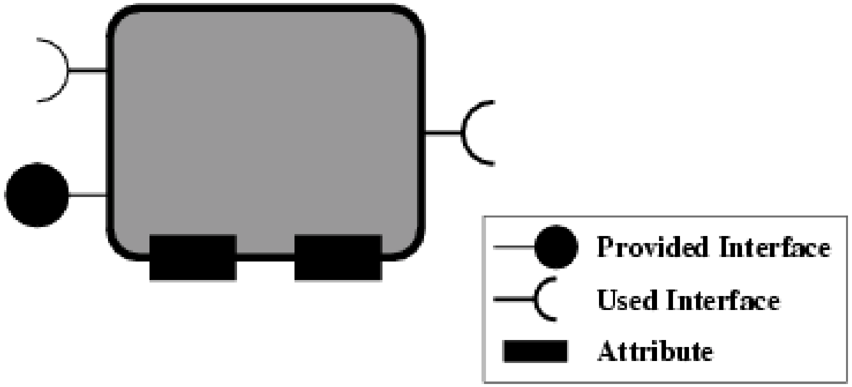
- **Component Domain Descriptor (.cdd)**
  - –Describes the target domain resources (e.g., nodes, interconnects, bridges and shared resources)
- **Deployment Plan Descriptor (.cdp)**
  - –Describes the mapping of a configured application into a domain, this includes mapping monolithic implementations to nodes, and requirements to resources.


Planning
Component Interface Descriptors (.ccd)
Deployment Plan Descriptor (.cdp)
realizes
Deployment Planning Tools
Component Package Descriptors (.cpd)
Component & Home Properties
Component Domain Descriptor (.cdd)

**These files could be coalesced into a smaller # (i.e., 1 file) in actual systems**

# Component-based Software: Component

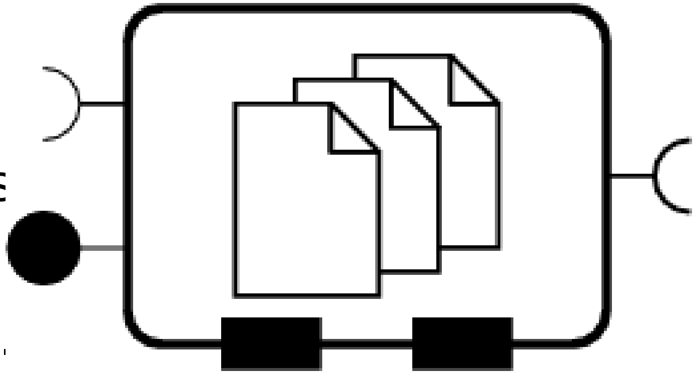
- **Component**
  - Modular
  - Encapsulates its contents
  - Replaceable “black box”, conformance defined by interface compatibility



- **Component Interface**
  - “Ports” consist of provided interfaces (facets) & required (used) interfaces (receptacles)
  - Attributes
- **Component Implementation**
  - “Monolithic” (i.e., executable software) or
  - “Assembly-based” (a set of interconnected subcomponents)

# Monolithic Component Implementation

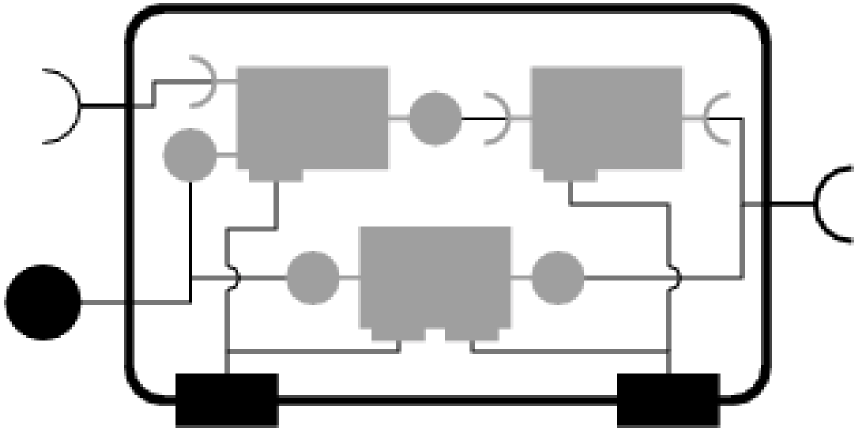
- **Monolithic Implementation**
  - Executable piece of software
    - One or more “implementation artifacts (e.g., .exe, .so, .o, .class)
    - Zero or more supporting artifacts (e.g. configuration files)
  - May have hardware or software requirements/constraints
    - Specific CPU (e.g., x86, PPC, SPARC)
    - Specific OS (e.g., Windows, VxWorks, Linux, Solaris)
    - Hardware devices (e.g., GPS sensor)



- Described by *.ccd, *.iad, & *.cid files

# Assembly-based Component Implementation

- Set of interconnected (sub)components
- Hardware & software independent
  - Reuses subcomponents as “black boxes,” independent of their implementation



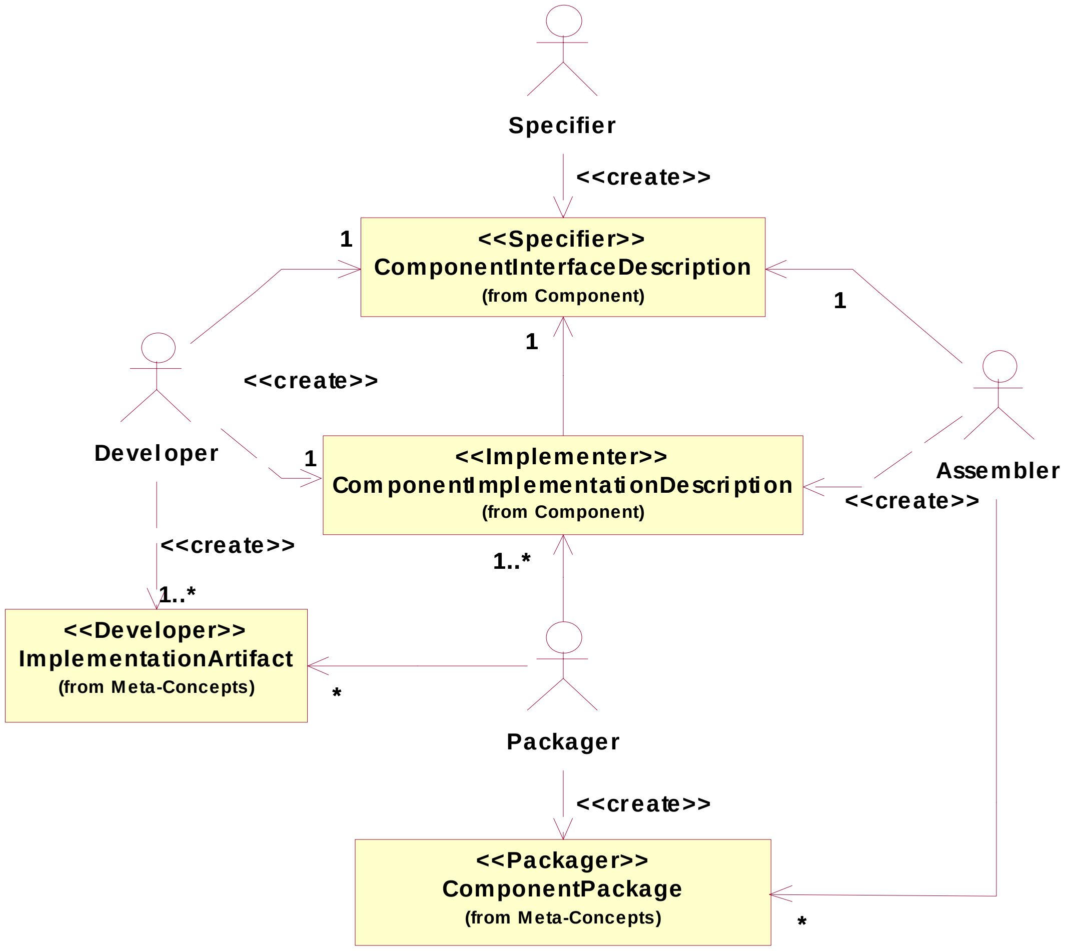
- Implements a specific (virtual) component interface
  - i.e., *external* ports & attributes are “mapped” to *internal* subcomponents
- Assemblies are fully reusable
  - Can be “standalone” applications or reusable components

- Assemblies are hierarchical
  - i.e., can be used in an encompassing assembly
  - Note recursion here…
- Described by *.ccd & *.cid files

# Component Package

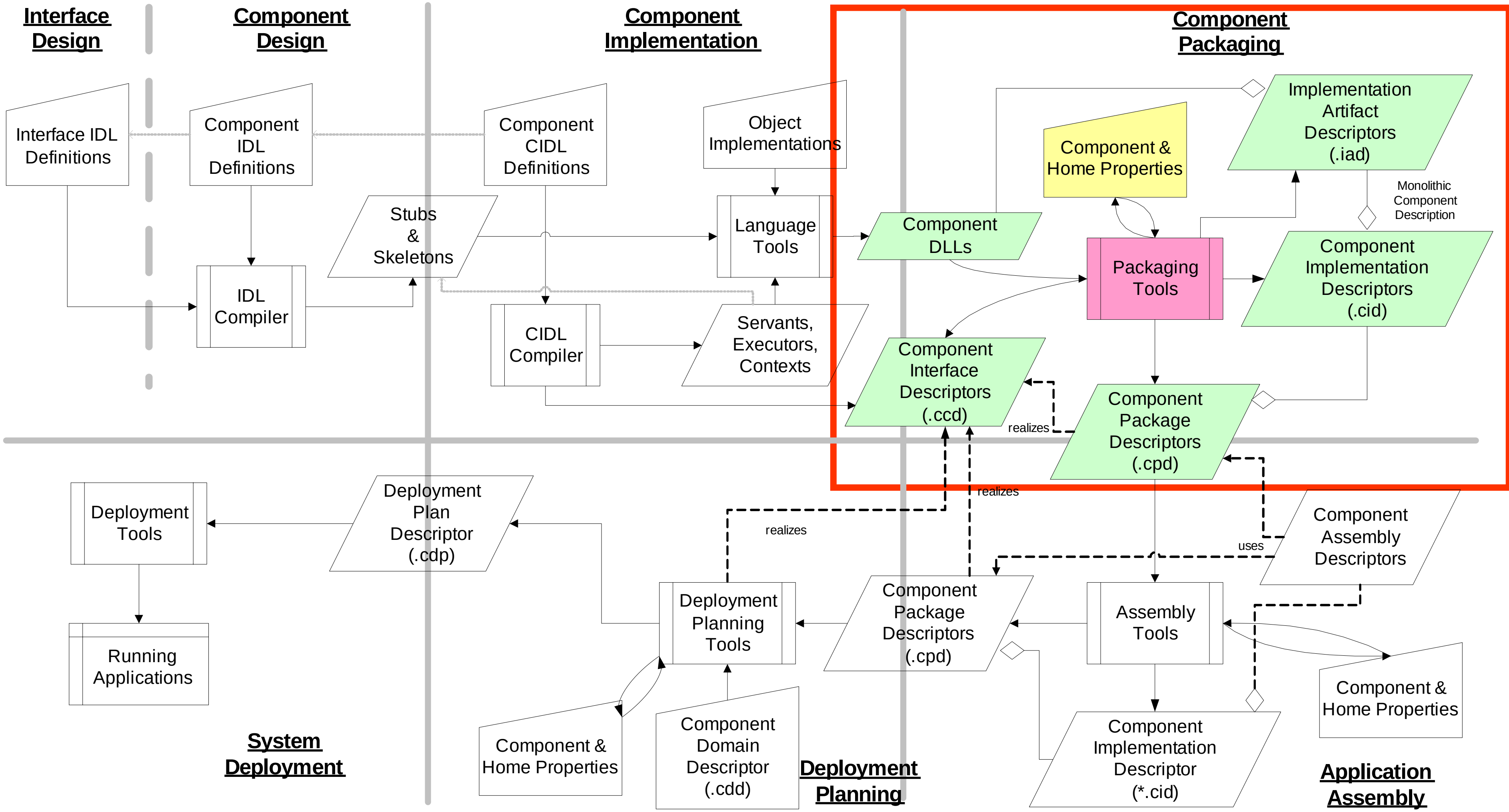
- **Component Package**
  - A set of alternative, replaceable implementations of the same component interface
    - e.g., implementations for Windows, Linux, and/or JVM
  - Can be a mix of monolithic & assembly-based implementations
    - e.g., a parallel, scalable implementation for a Solaris symmetric multiprocessor or a single monolithic Java component
  - Implementations may have different “quality of service” (QoS)
    - e.g., latency, resolution, security
  - “Best” implementation is chosen at deployment time by *Planner*
    - Based on available hardware & QoS requirements

# CCM Development Actors



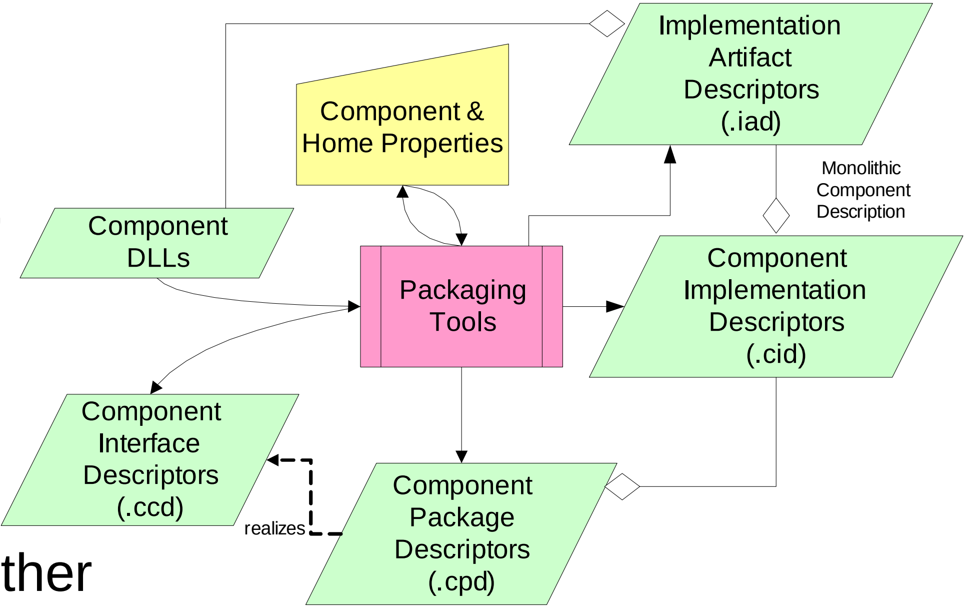
# Component Packaging

Goal: Associate a component implementation with its meta-data


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Stubs & Skeletons
Language Tools
IDL Compiler
CIDL Compiler
Servants, Executors, Contexts
Component & Home Properties
Implementation Artifact Descriptors (.iad)
Monolithic Component Description
Component DLLs
Packaging Tools
Component Implementation Descriptors (.cid)
Component Interface Descriptors (.ccd)
Component Package Descriptors (.cpd)
realizes
realizes
realizes
uses
Deployment Tools
Deployment Plan Descriptor (.cdp)
Running Applications
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component Assembly Descriptors
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
Component & Home Properties
System Deployment
Deployment Planning
Application Assembly

# Component Packaging Tools

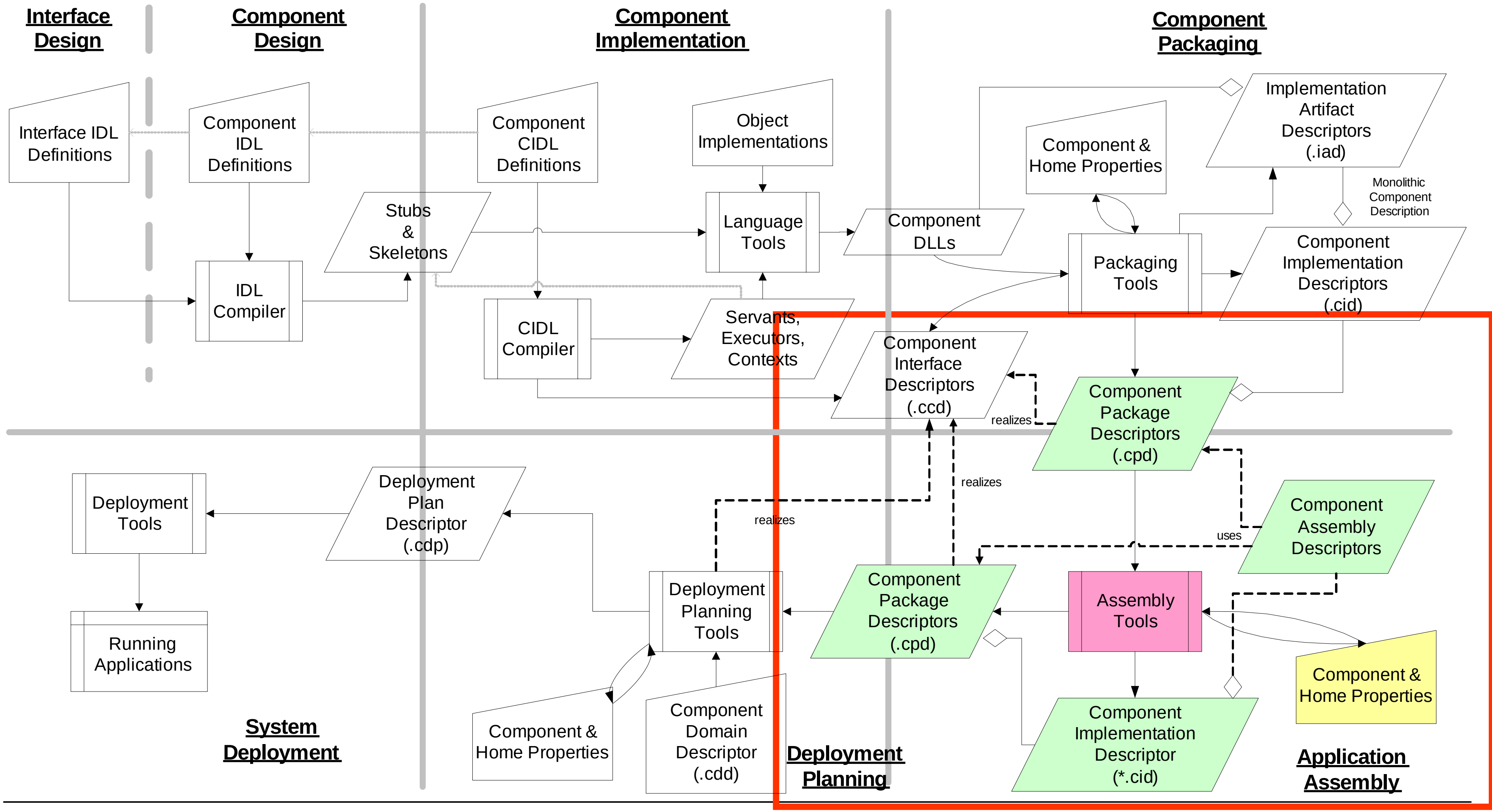
- Goals
  - Extract systemic properties into metadata
  - Configure components, containers, target environment, & applications
  - Provide abstraction of *physical* information, e.g., OS version, location of DLLs, etc.


Implementation Artifact Descriptors (.iad)
Component & Home Properties
Monolithic Component Description
Component DLLs
Packaging Tools
Component Implementation Descriptors (.cid)
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)

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

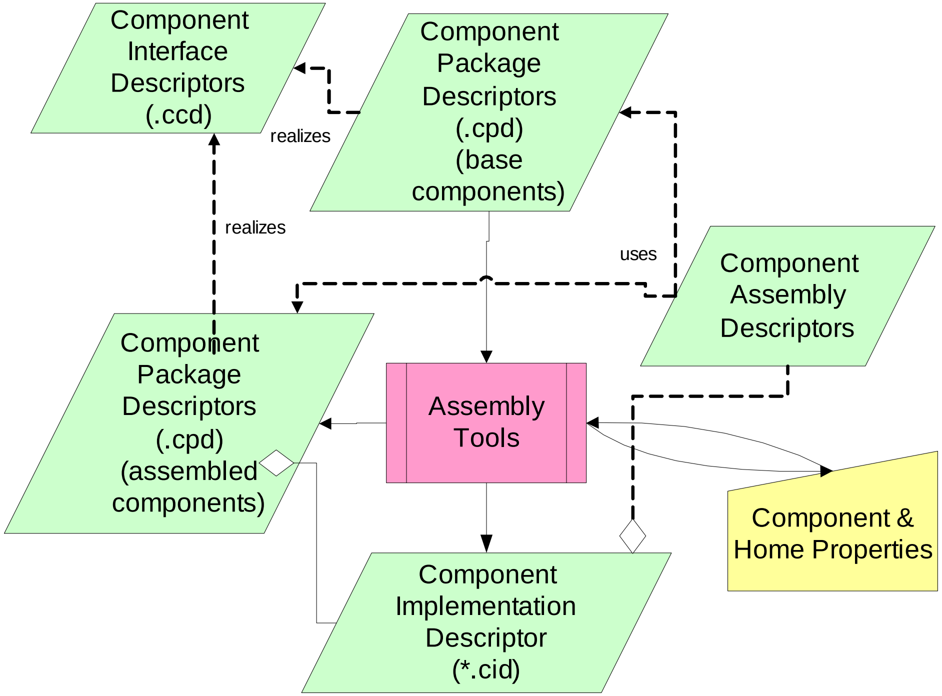
# Application Assembly

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


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Stubs & Skeletons
Language Tools
IDL Compiler
CIDL Compiler
Servants, Executors, Contexts
Component DLLs
Component & Home Properties
Implementation Artifact Descriptors (.iad)
Monolithic Component Description
Packaging Tools
Component Implementation Descriptors (.cid)
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
realizes
realizes
uses
Component Assembly Descriptors
Deployment Tools
Deployment Plan Descriptor (.cdp)
Running Applications
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component & Home Properties
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
System Deployment
Deployment Planning
Application Assembly

# Application Assembly Tools

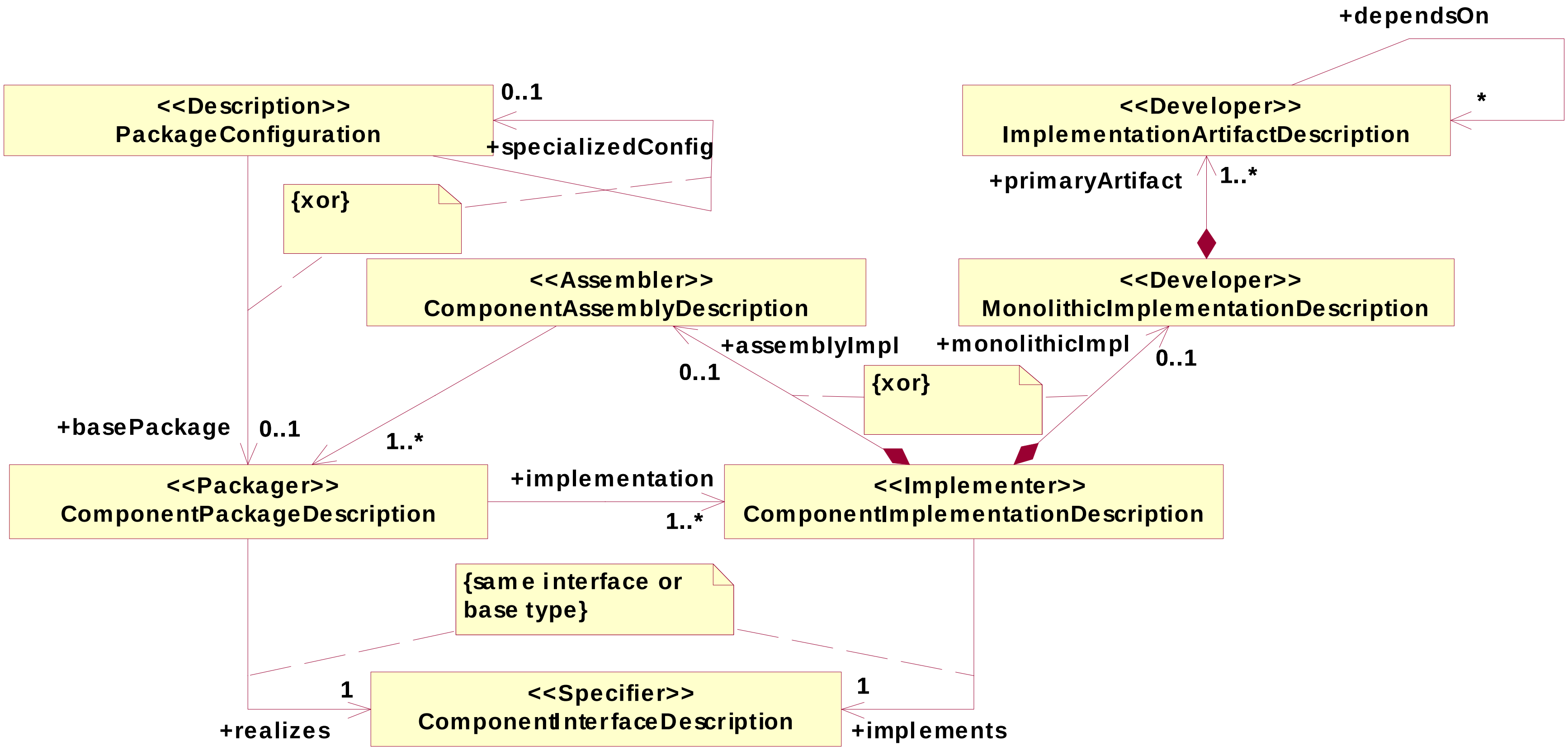
- Goals
  - Compose higher level components from set of (sub)components
  - Store composition & connection information as metadata
  - Provide abstraction of *logical* information, e.g., interconnections
- Component assembly description specifies:
  - Subcomponent packages
  - Subcomponent instantiation & configuration
  - Interconnections
  - Mapping of ports & properties to subcomponents


Component Interface Descriptors (.ccd)
Component Package Descriptors (.cpd) (base components)
realizes
realizes
uses
Component Assembly Descriptors
Component Package Descriptors (.cpd) (assembled components)
Assembly Tools
Component & Home Properties
Component Implementation Descriptor (*.cid)

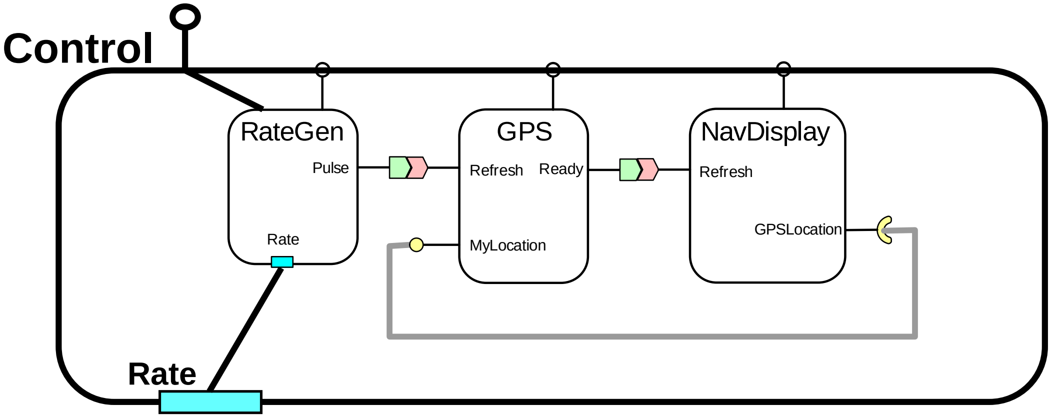
- “Pure metadata” construct (no directly executable code, hardware-agnostic)

# Component Data Model Overview

We'll show XML snippets for each of these component data model elements


+dependsOn
<<Description>> PackageConfiguration
0..1
+specializedConfig
<<Developer>> ImplementationArtifactDescription
*
+primaryArtifact
1..*
{xor}
<<Assembler>> ComponentAssemblyDescription
<<Developer>> MonolithicImplementationDescription
+assemblyImpl
+monolithicImpl
0..1
0..1
{xor}
+basePackage
0..1
1..*
<<Packager>> ComponentPackageDescription
+implementation
1..*
<<Implementer>> ComponentImplementationDescription
{same interface or base type}
1
1
<<Specifier>> ComponentInterfaceDescription
+realizes
+implements

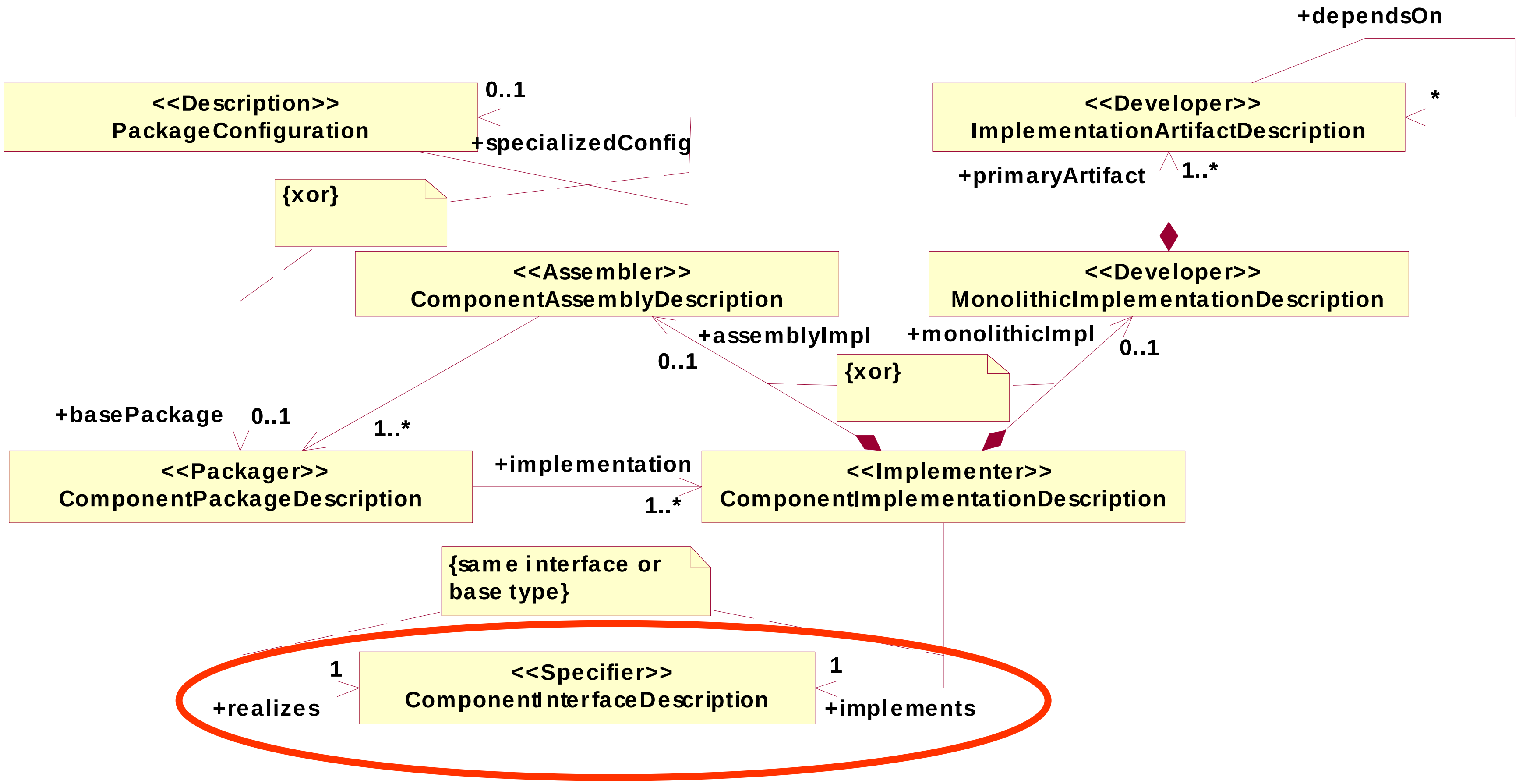
# Example CCM DRE Application



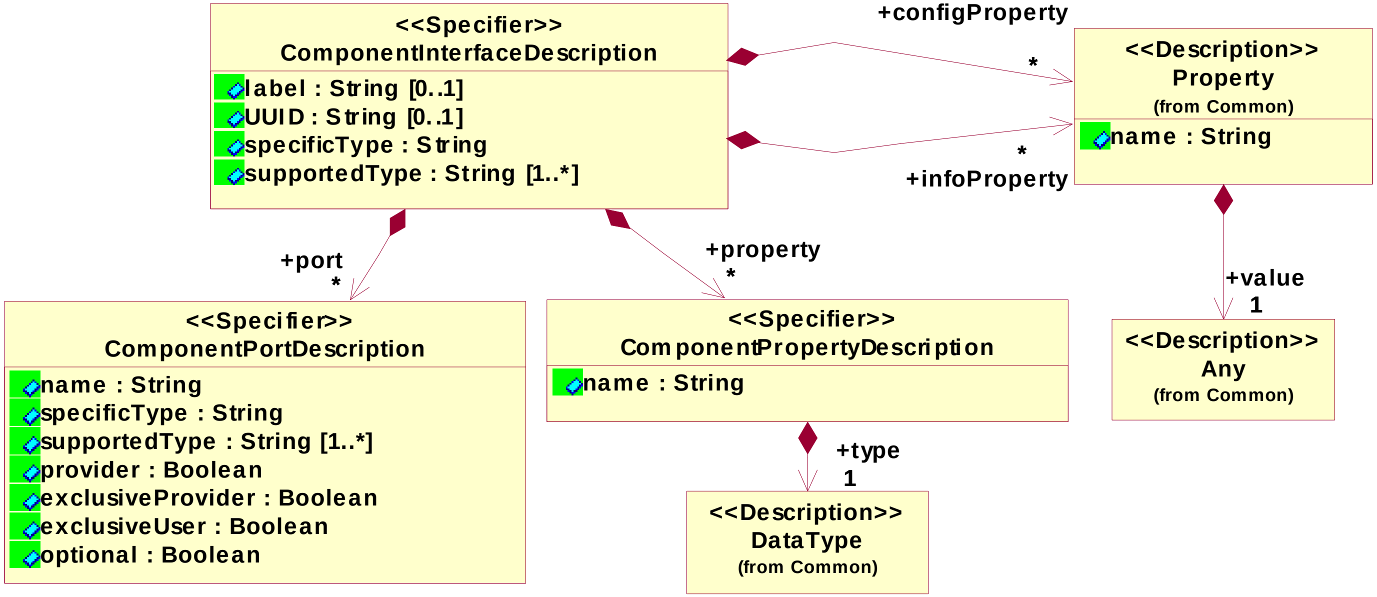
Display component

- The **Display** component is an assembly of three (sub)components
- **RateGen**, **GPS**, & **NavDisplay** implemented monolithically (for this example)
- **GPS** component requires a particular type of GPS device
- Two alternative implementations for **NavDisplay**
  - Text-based & GUI versions

# Component Interface Description



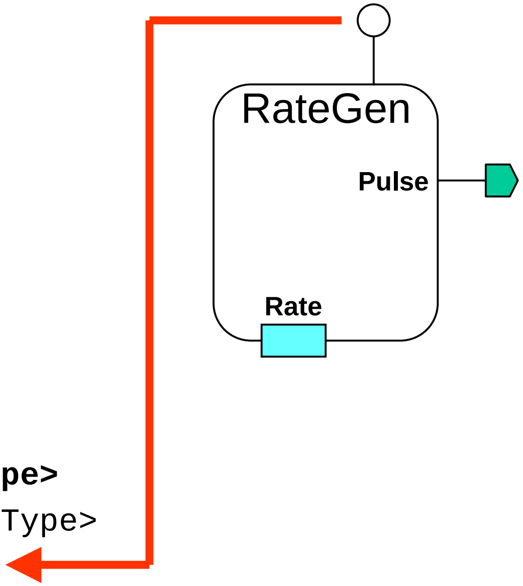
# Component Interface Description



- Metadata used by *Specifiers* to describe component interface (*.ccd file)
  - Identifies a component's specific (most-derived) type & supported (inherited) types
  - Describes a component's ports & properties (attributes)
  - Optionally configures default property values

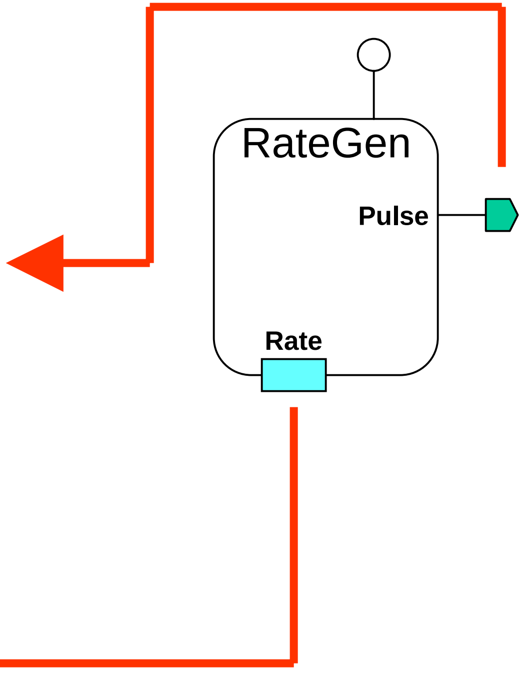
# Component Interface Descriptor for the RateGen Component: RateGen.ccd (1/3)

```
<?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/rate_control:1.0</specificType>
    <supportedType>IDL:HUDisplay/rate_control:1.0</supportedType>
    <provider>true</provider>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>false</exclusiveUser>
    <optional>true</optional>
    <kind>Facet</kind>
  </port>
  [...]
</Deployment:ComponentInterfaceDescription>
```



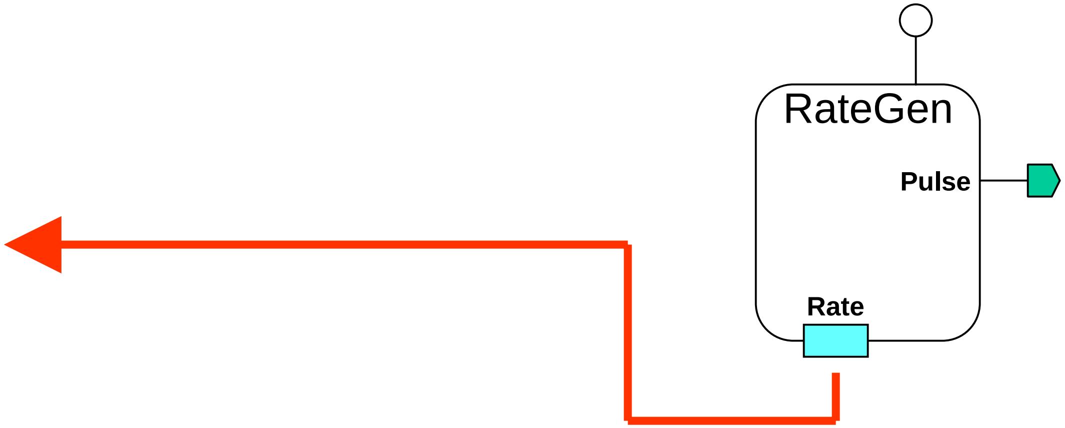
# Component Interface Descriptor for the RateGen Component: RateGen.ccd (2/3)

```
<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 the RateGen Component: RateGen.ccd (3/3)

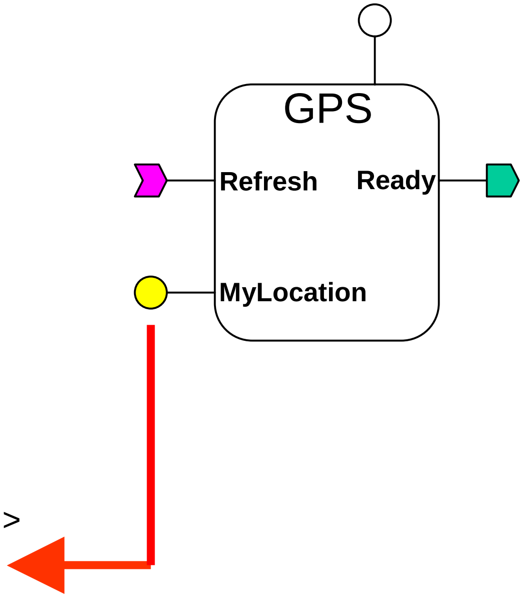
```
<Deployment:ComponentInterfaceDescription>
  [...]
 <configProperty>
    <name>Rate</name>
    <value>
      <type>
        <kind>tk_long</kind>
      </type>
      <value>
        <long>1</long>
      </value>
    </value>
  </configProperty>
</Deployment:ComponentInterfaceDescription>
```



- Note the default value for the **`Rate`** property
  - Can be overridden by implementation, package, assembly, user, or at deployment time

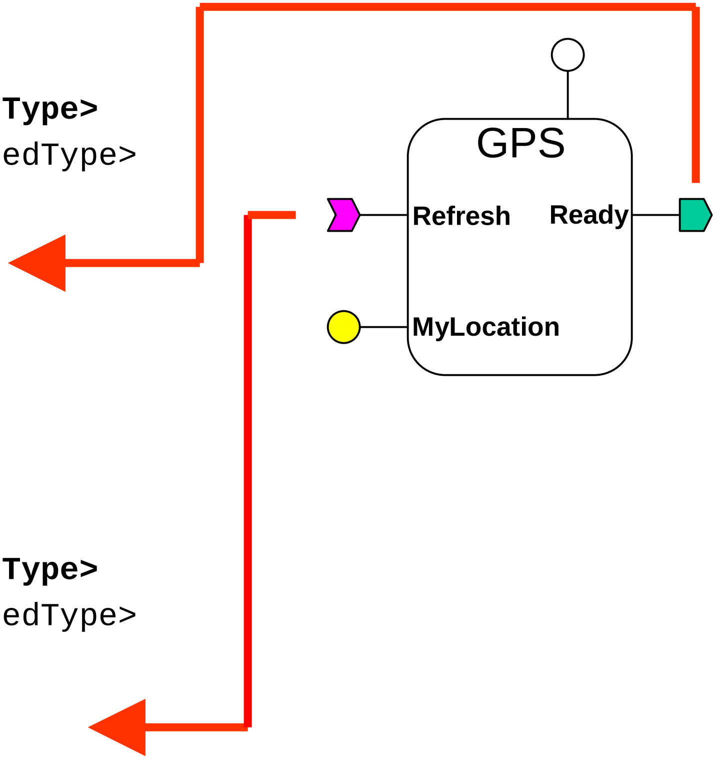
# Component Interface Descriptor for the GPS Component: GPS.ccd (1/2)

```
<?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>Global 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>
```



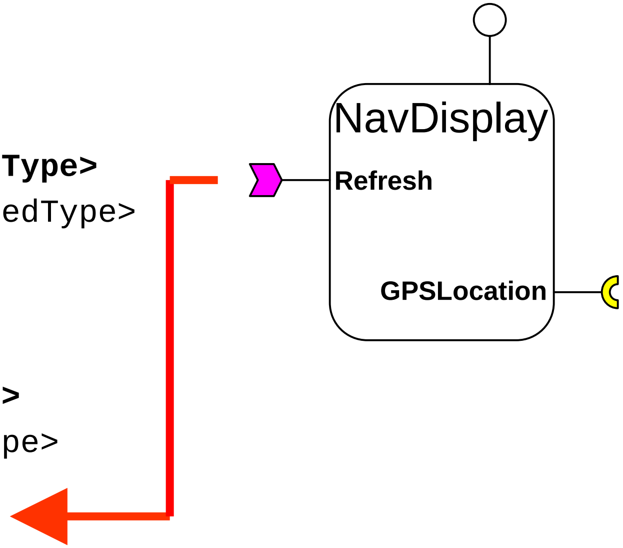
# Component Interface Descriptor for the GPS Component: GPS.ccd (2/2)

```
<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>
```



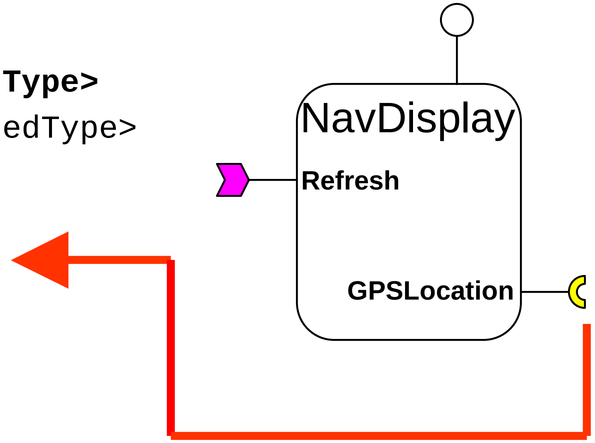
# Component Interface Descriptor for the NavDisplay Component: NavDisplay.ccd (1/2)

```
<Deployment:ComponentInterfaceDescription
    xmlns:Deployment='http://www.omg.org/Deployment'
    xmlns:xmi='http://www.omg.org/XMI'
    >
  <label>Navigation 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>
```

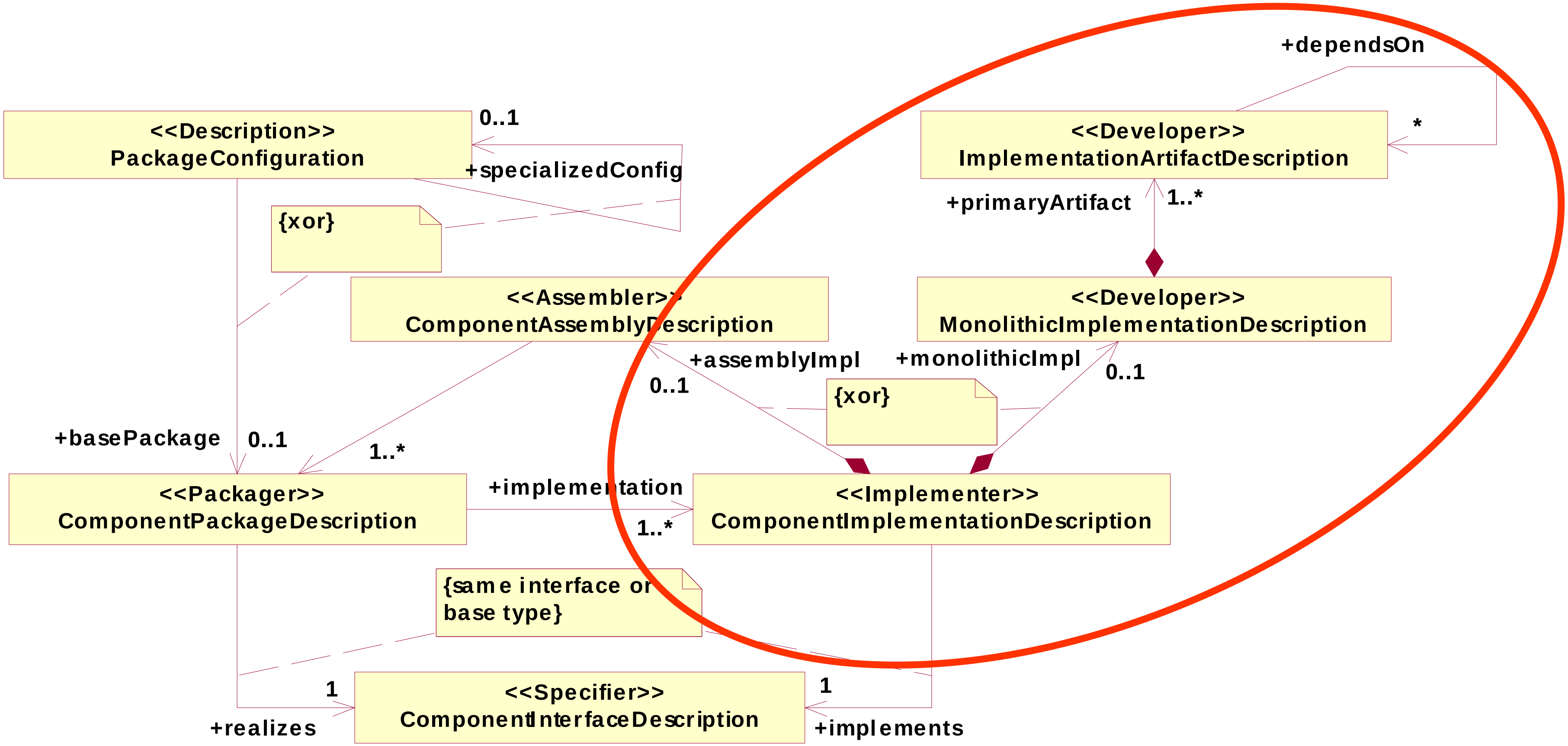


# Component Interface Descriptor for the NavDisplay Component: NavDisplay.ccd (2/2)

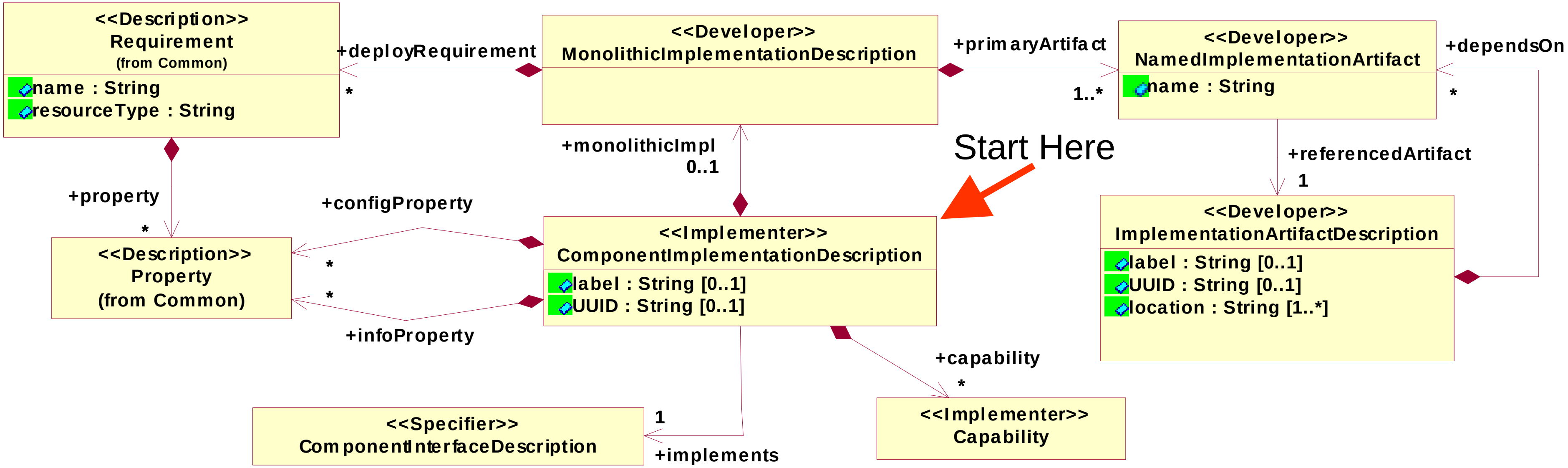
```
<Deployment:ComponentInterfaceDescription>
 [...]
 <port>
    <name>GPSLocation</name>
    <specificType>IDL:HUDisplay/position:1.0</specificType>
    <supportedType>IDL:HUDisplay/position:1.0</supportedType>
    <provider>false</provider>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>true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



# Component Implementation Description for a Monolithic Implementation



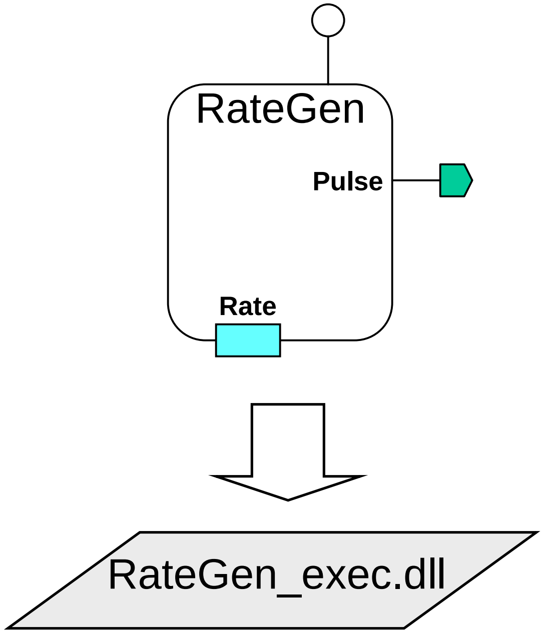
# Component Implementation Description for a Monolithic Implementation



- Metadata used by *Developers* to describe a monolithic component implementation (*.cid file)
  - Contains deployment requirements & QoS capabilities
  - References artifacts by URL, which may have dependencies

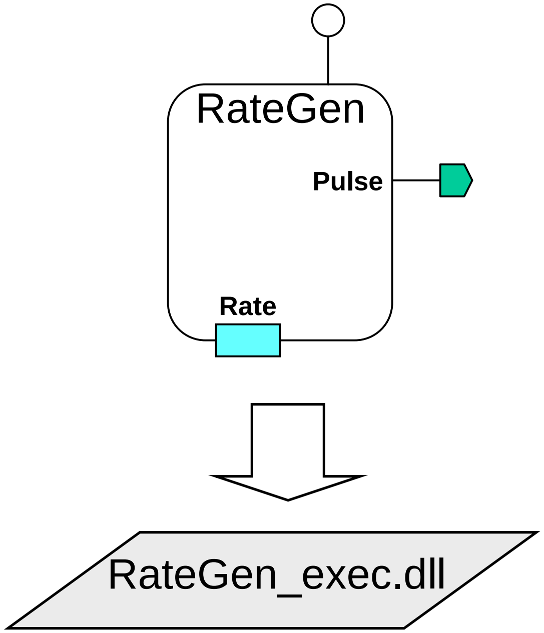
# Component Implementation Descriptor for the RateGen Component: RateGen.cid (1/2)

```
<?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="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>
```



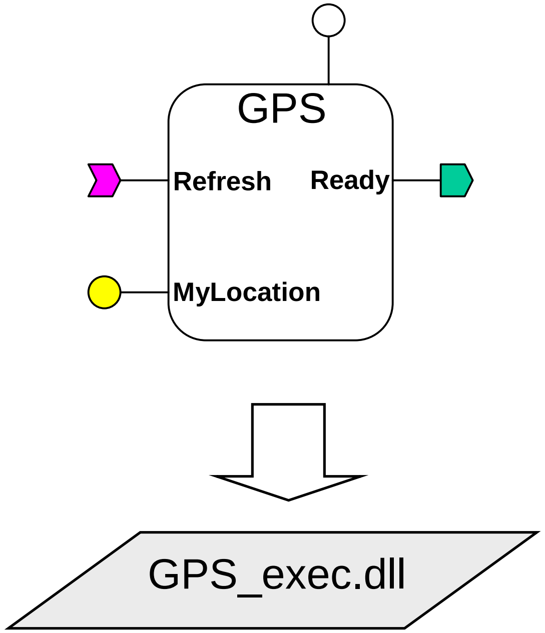
# Component Implementation Descriptor for the RateGen Component: RateGen.cid (2/2)

```
<Deployment:ComponentImplementationDescription>
  <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>
```



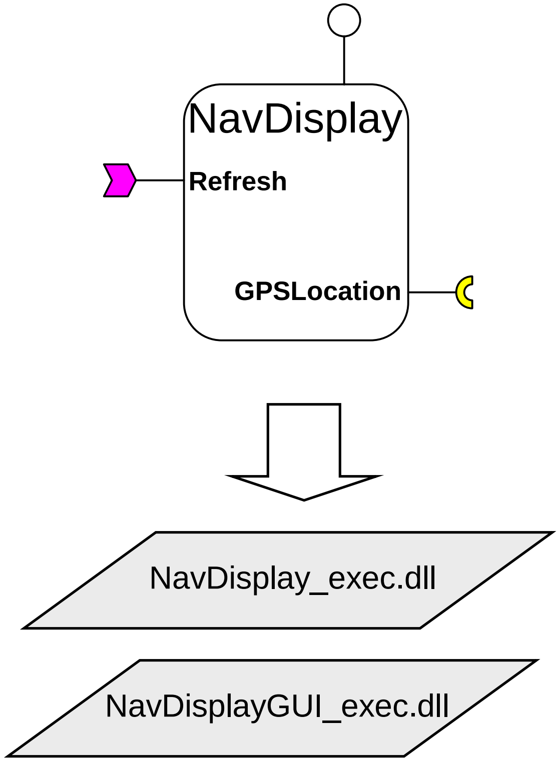
# Component Implementation Descriptor for the GPS Component: GPS.cid (excerpt)

```
<?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>
```

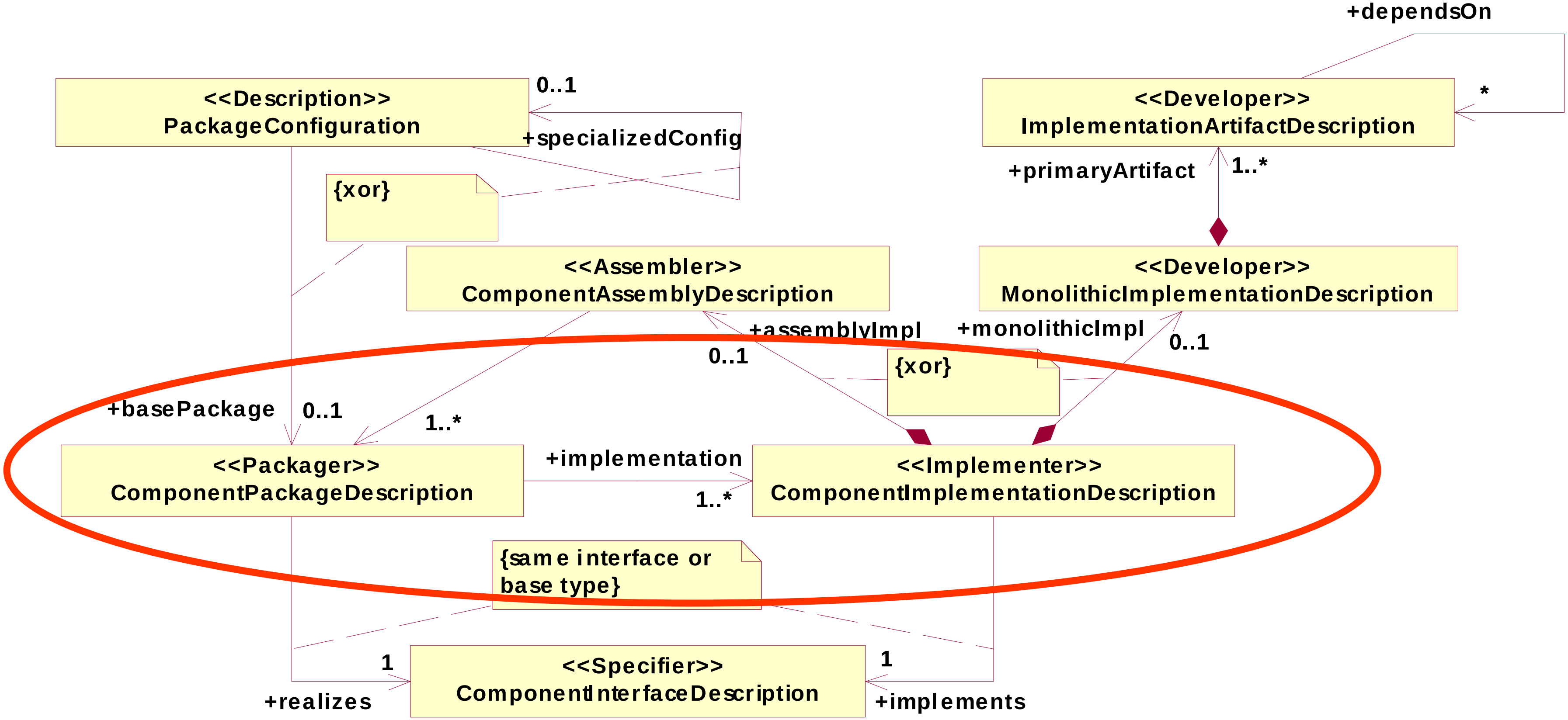


# Two Component Implementation Descriptors for the NavDisplay Component

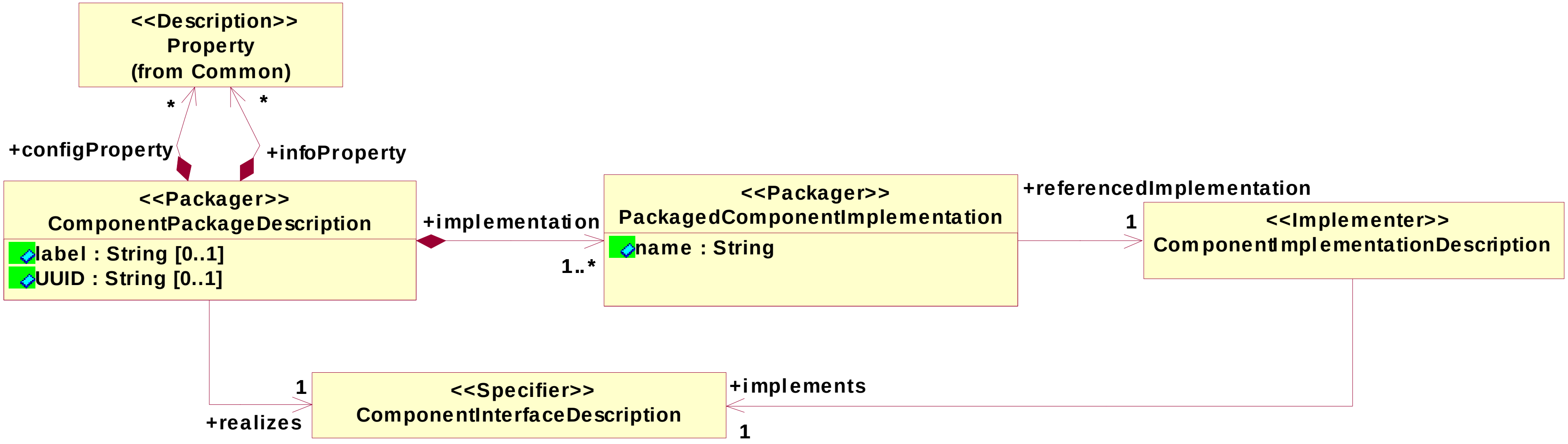
- Two alternative implementations (i.e., text vs. GUI) and thus two Component Implementation Descriptor (*.cid) files:
  - **`NavDisplay.cid`**
    - Text-based implementation
  - **`NavDisplayGUI.cid`**
    - GUI implementation
    - “deployRequirement” on graphical display
- XML code not shown here (but available in CIAO release)



# Component Package Description



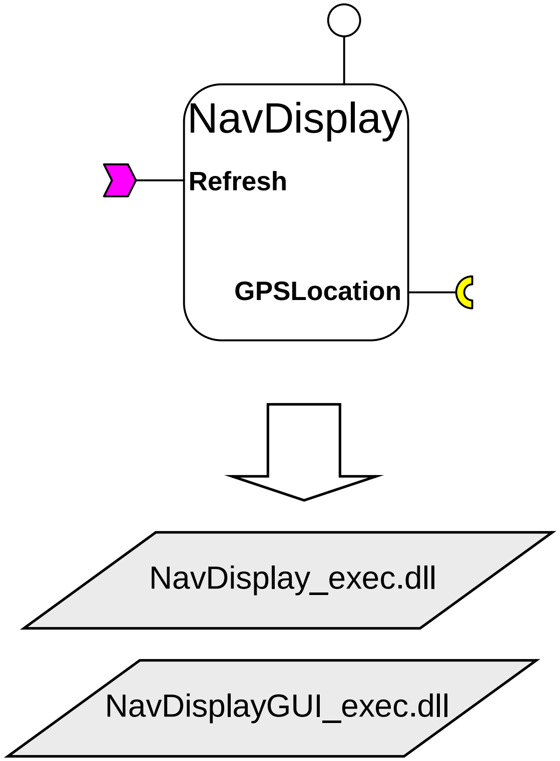
# Component Package Description



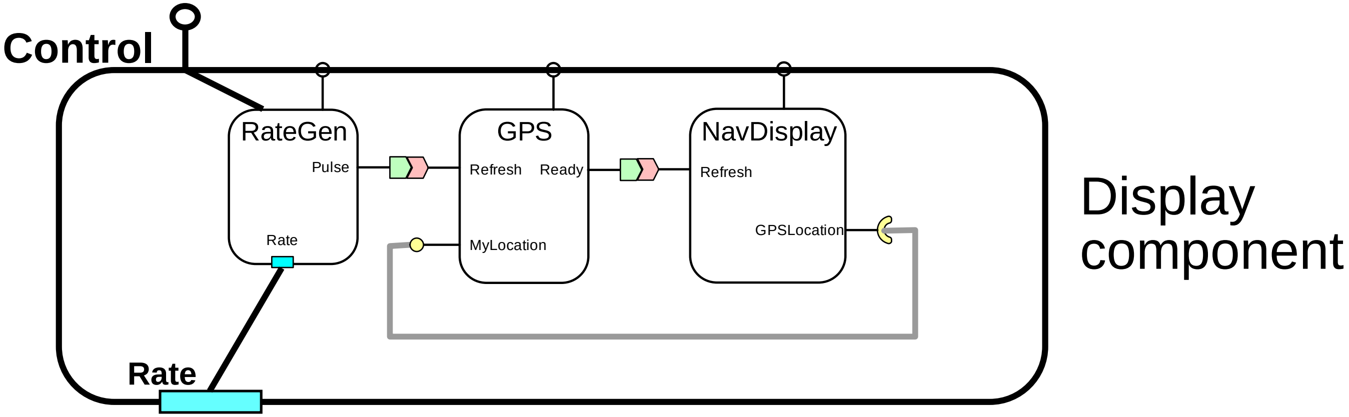
- Metadata used by *Packagers* to describe a set of alternative implementations of the same component (*.cpd files)
  - May redefine (overload) properties

# Component Package Descriptor for the 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>Navigation 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>
```



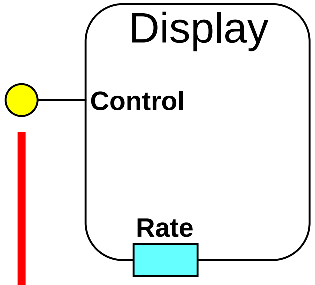
# Display Component Assembly



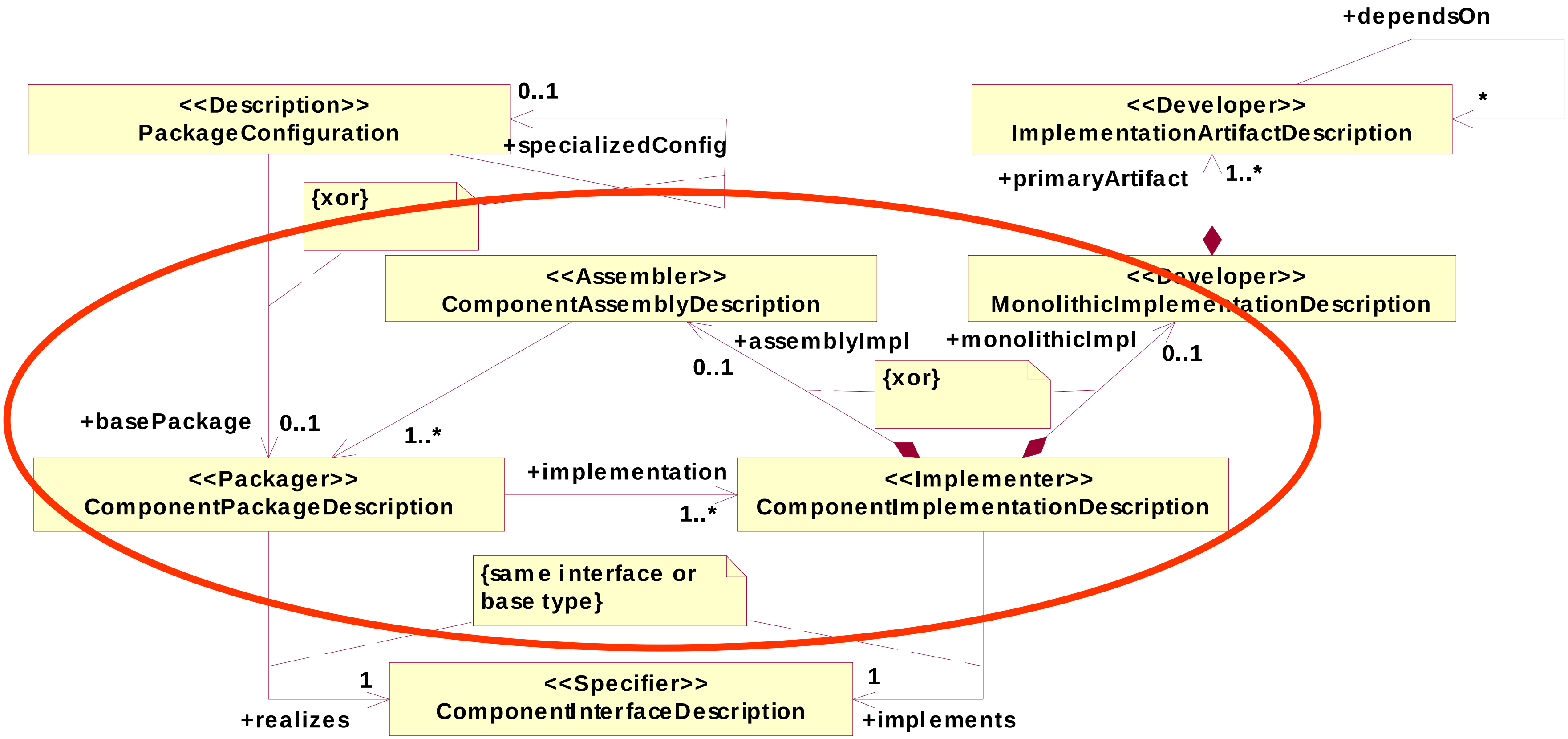
- Recall that the **`Display`** component is an assembly of (sub)components
- We've shown the various D&C XML files for **`Display`**'s three (sub)components
- We now show the assembly for the **`Display`** component itself, which is essentially a façade
- Again, note the recursion, where assembly-based components can be composed of monolithic and/or assembly-based (sub)components…

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

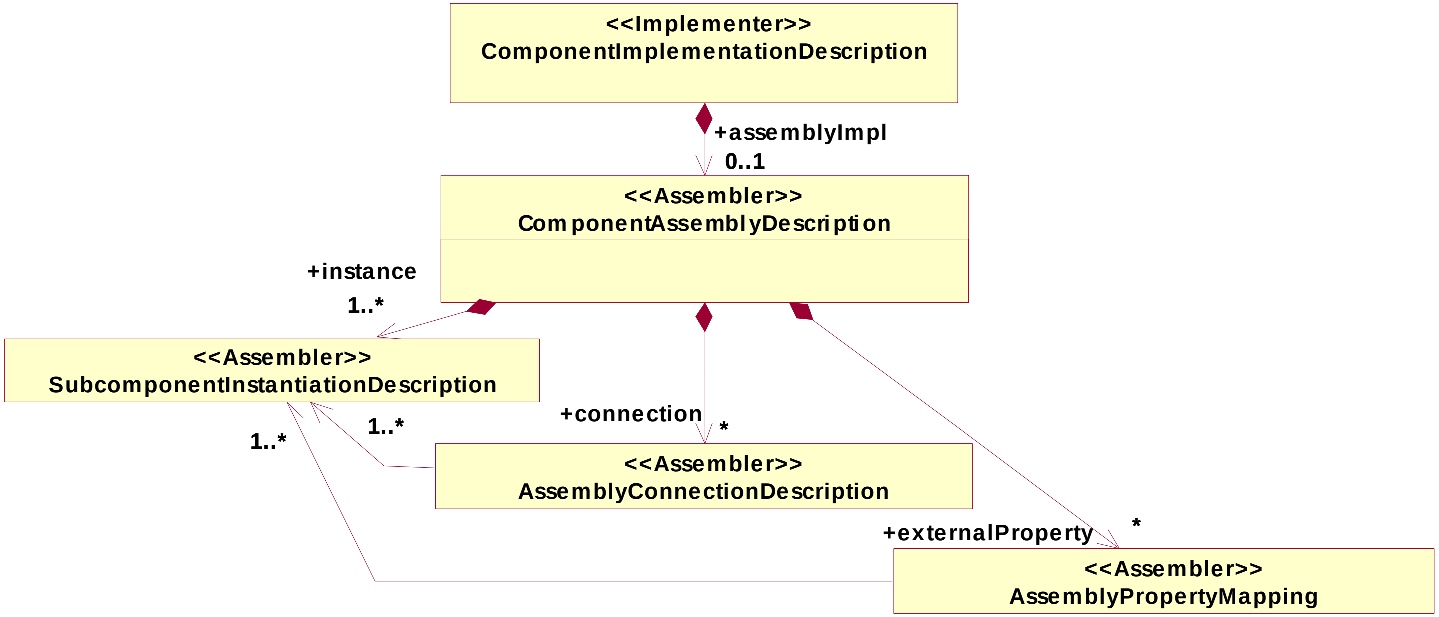
```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Deployment:ComponentInterfaceDescription
    xmlns:Deployment='http://www.omg.org/Deployment'>
  <label>Navigation System</label>
  <specificType>IDL:HUDisplay/Display:1.0</specificType>
  <idlFile>Display.idl</idlFile>
  <port>
    <name>control</name>
    <specificType>IDL:HUDisplay/rate_control:1.0</specificType>
    <supportedType>IDL:HUDisplay/rate_control: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

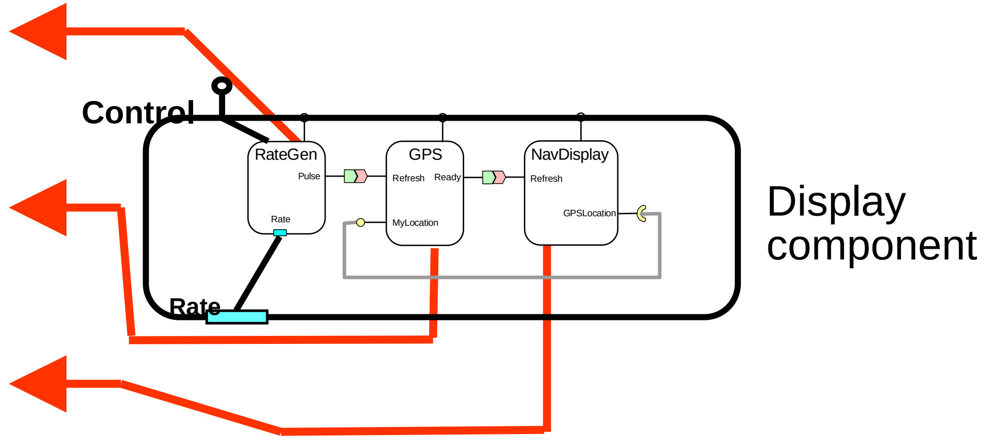


- Metadata used by *Assemblers* to describe an assembly-based implementation (*.cid files)
  - Define subcomponent instances
  - Connections between subcomponent ports
  - Connecting assembly (external) ports to subcomponent (internal) ports
  - Mapping assembly properties to subcomponent properties

# Component Implementation Descriptor for the 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>
```

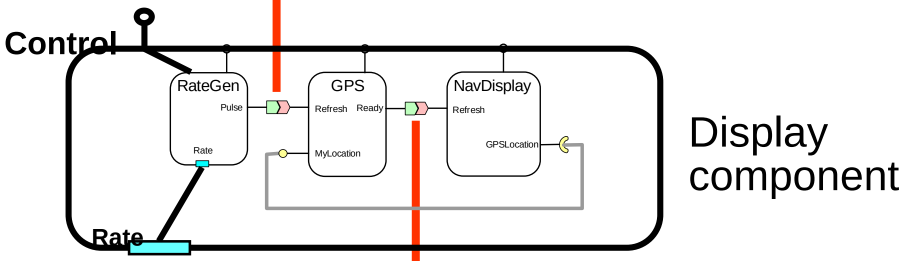
Define subcomponent instances



# Component Implementation Descriptor for the 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>
```

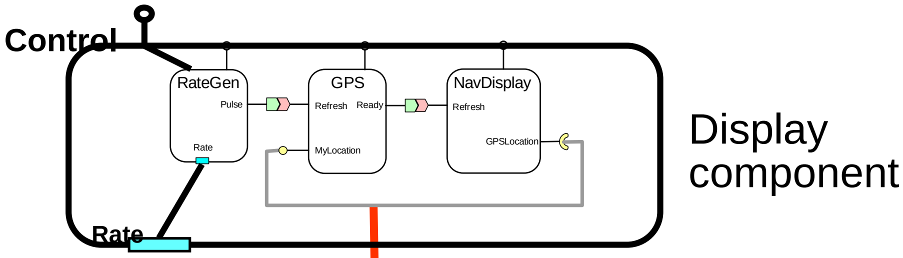
Connections between subcomponent ports



# Component Implementation Descriptor for the 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>
```

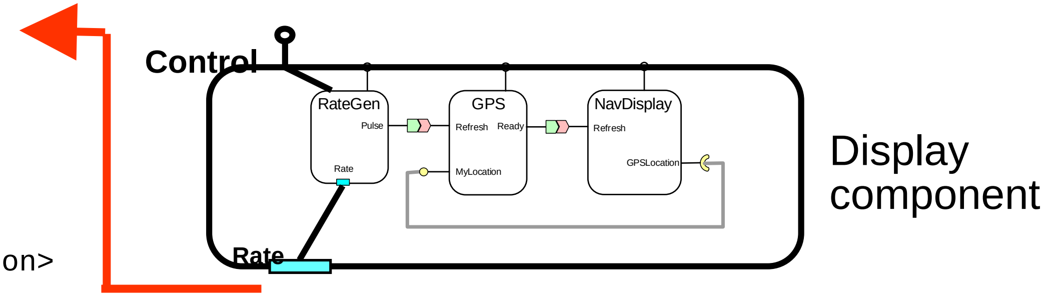
Connecting assembly (external) ports to subcomponent (internal) ports



The external/internal mappings are virtual, i.e., there’s no extra indirection overhead

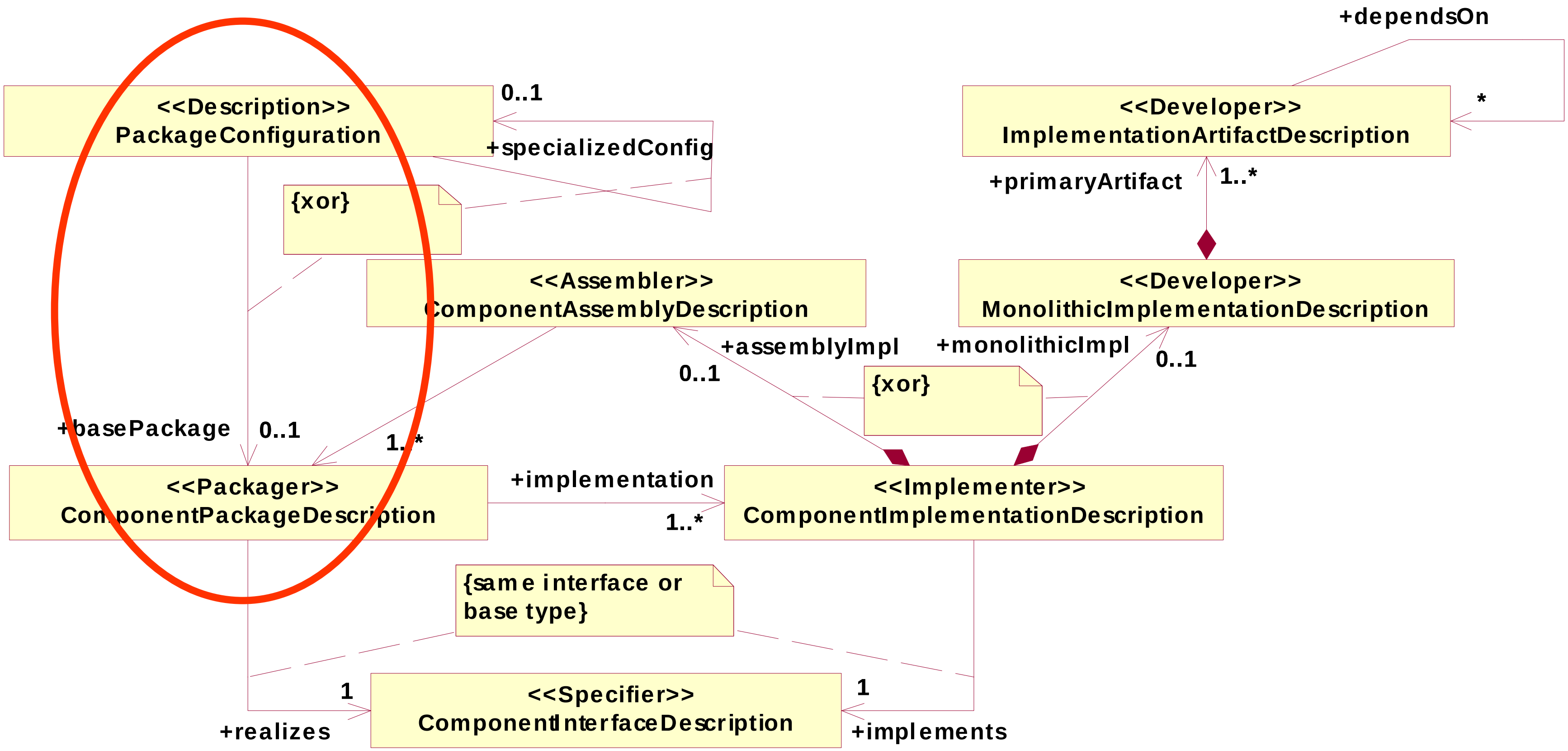
# Component Implementation Descriptor for the Display Component: Display.cid (4/4)

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

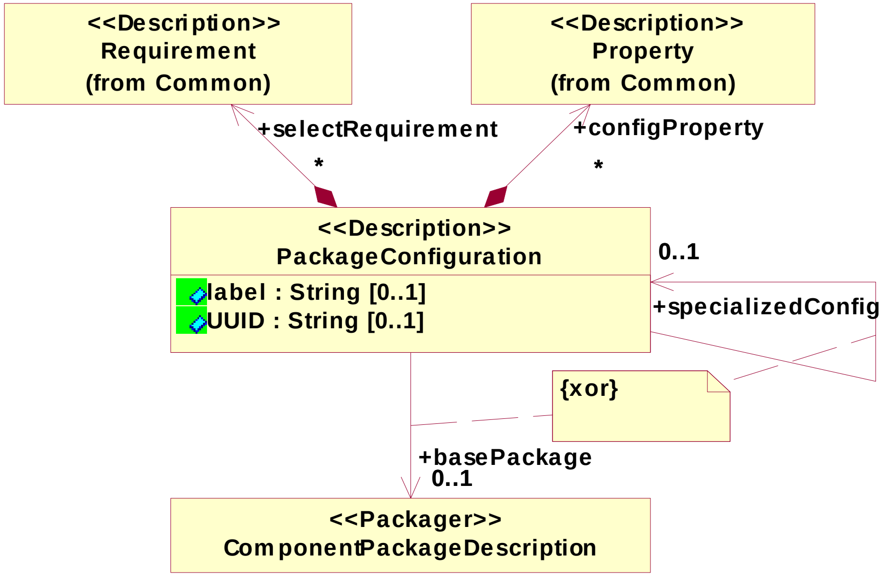


Mapping an assembly’s (external) properties to subcomponent (internal) properties

# Package Configuration



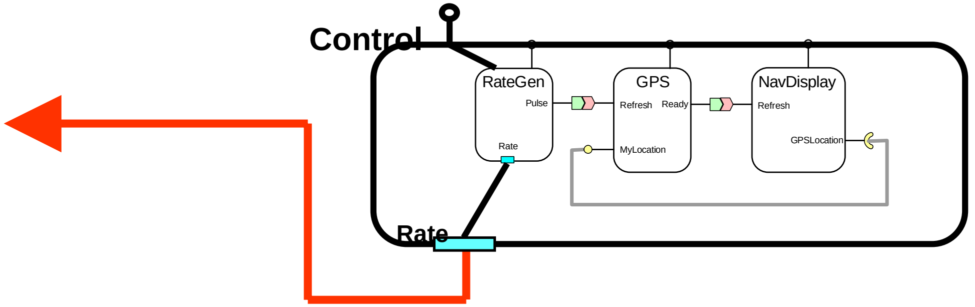
# Package Configuration



- Metadata used by *Packagers* to describe a reusable component package (*.pcd files)
  - Sets initial configuration
  - Sets QoS requirements
    - to be matched against implementation capabilities
  - May refine (specialize) existing package

# Package Configuration for the Display Application: Display.pcd (1/1)

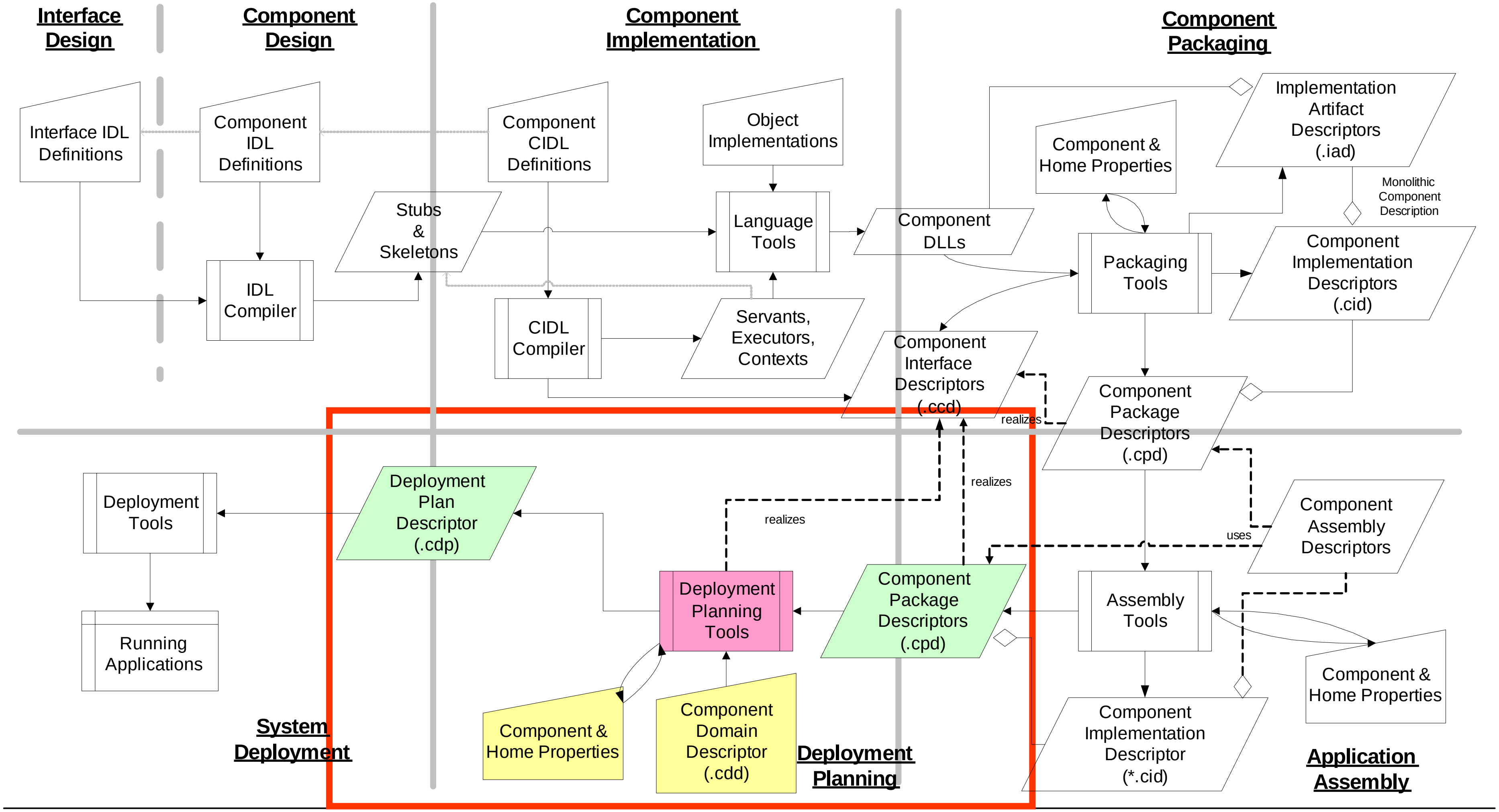
```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Deployment:PackageConfiguration
    xmlns:Deployment='http://www.omg.org/Deployment'
    xmlns:xmi='http://www.omg.org/XMI'
    >
  <label>Display Application</label>
  <configProperty>
    <name>Rate</name>
    <value>
      <type>
        <kind>tk_long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



Display component

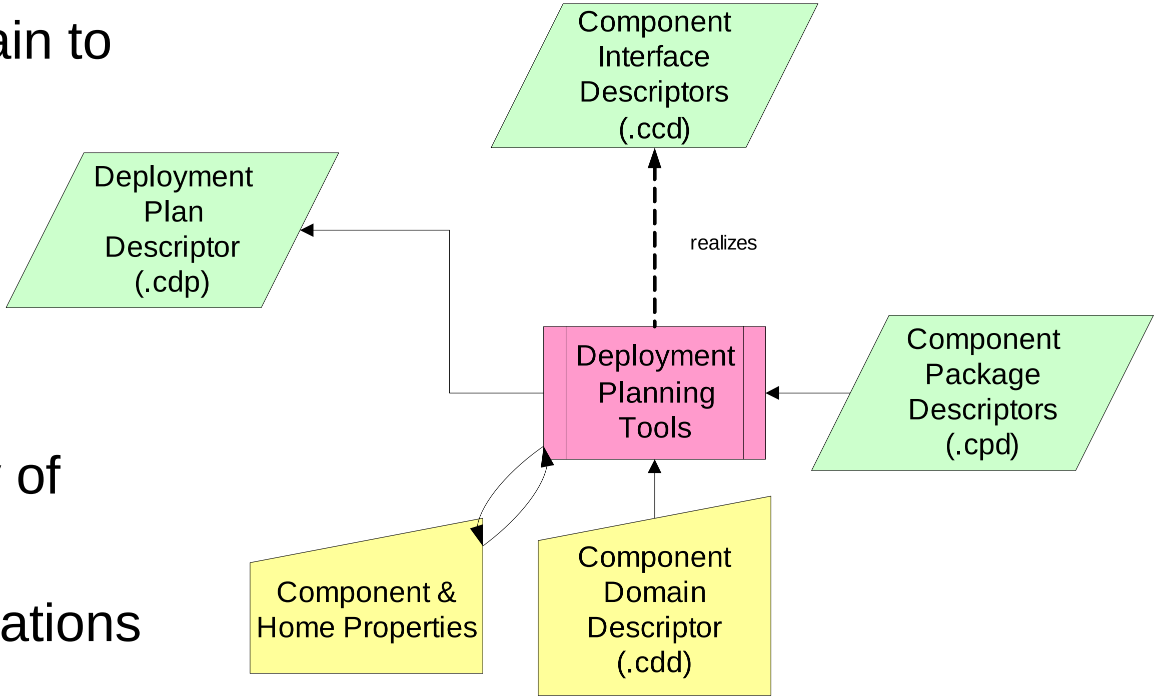
# Deployment Planning

Goal: Extract application and match with deployment target into *deployment plan*

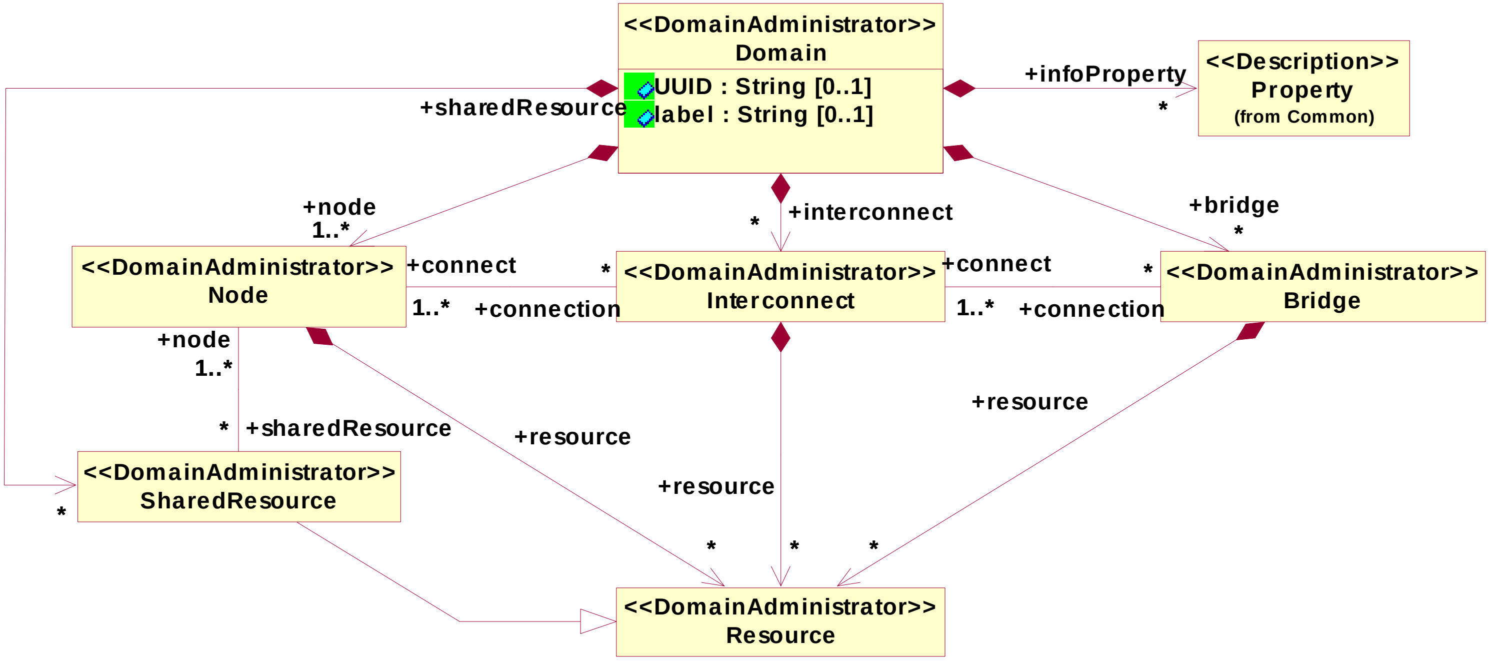

Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Implementation Artifact Descriptors (.iad)
Component & Home Properties
Stubs & Skeletons
Language Tools
Component DLLs
Monolithic Component Description
IDL Compiler
CIDL Compiler
Servants, Executors, Contexts
Packaging Tools
Component Implementation Descriptors (.cid)
Component Interface Descriptors (.ccd)
Component Package Descriptors (.cpd)
realizes
realizes
realizes
uses
Deployment Tools
Deployment Plan Descriptor (.cdp)
Component Assembly Descriptors
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Running Applications
Component & Home Properties
Component & Home Properties
Component Domain Descriptor (.cdd)
Component Implementation Descriptor (*.cid)
System Deployment
Deployment Planning
Application Assembly

# Deployment Planning Tools

- Goals
  - Concretize deployment meta-data
  - Using Deployment Domain to describe deployment environment
- Component Deployment Plan description:
  - Flatten the assembly hierarchy -- an assembly of monolithic components
  - Deployment details – locations to deploy components
  - Interconnections
  - Mapping of ports and properties to subcomponents

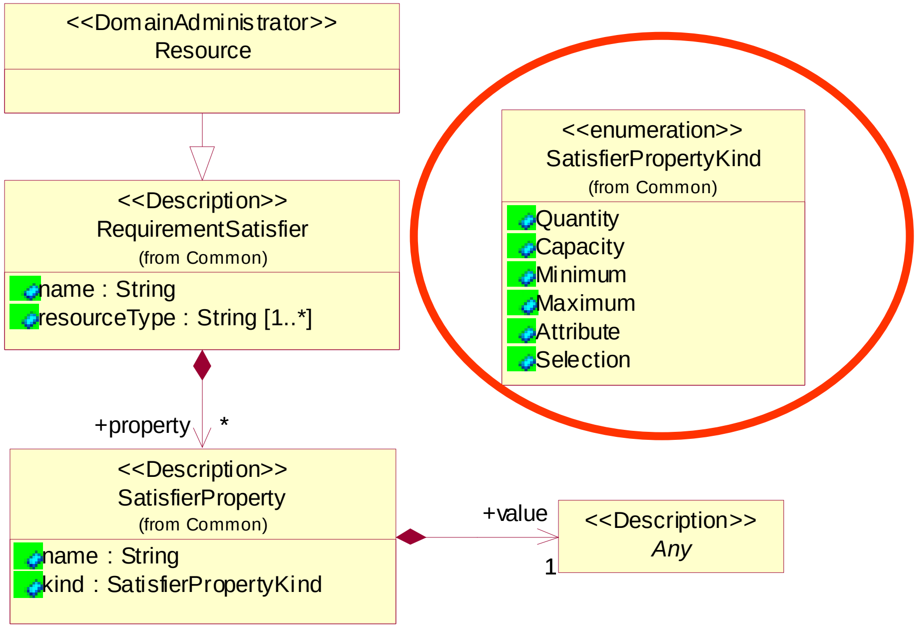

Component Interface Descriptors (.ccd)
realizes
Deployment Plan Descriptor (.cdp)
Deployment Planning Tools
Component Package Descriptors (.cpd)
Component & Home Properties
Component Domain Descriptor (.cdd)

# Target Data Model



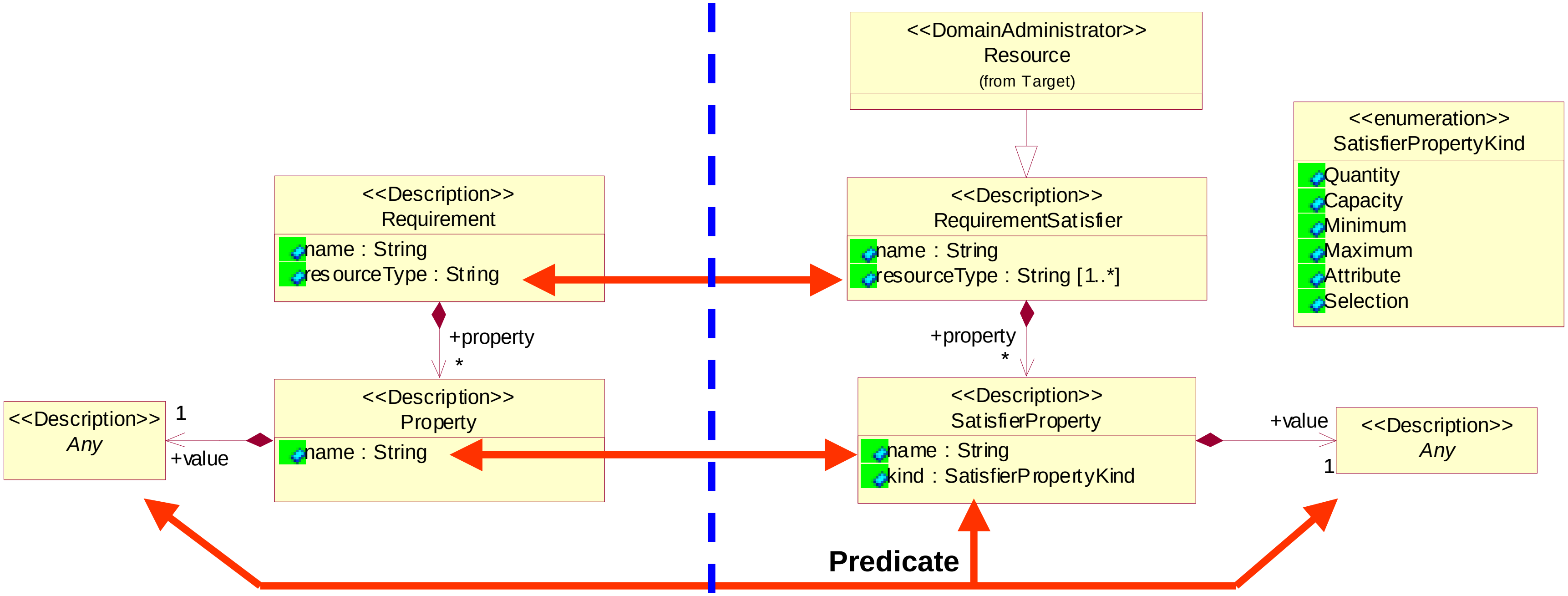
- Metadata used by *Domain Administrators* to describe a “target domain” (*.cdd files)
  - –***Nodes***: targets for executing monolithic component implementations
  - –***Interconnect***: direct connections (e.g., Ethernet cable, Myrinet)
  - –***Bridge***: indirect connections (e.g., routers, switches)

# Target Data Model: Resources



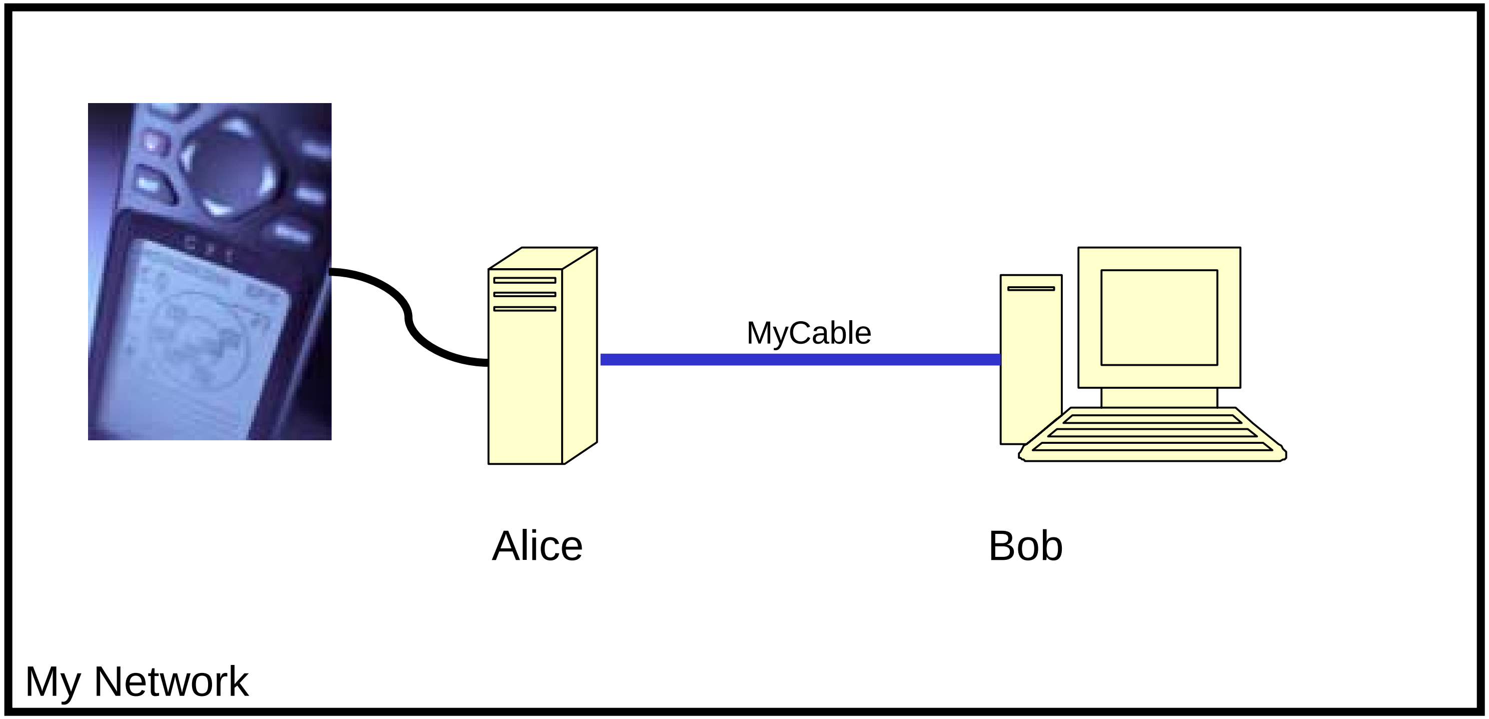
- Metadata used by *Domain Administrators* to describe a consumable resource (*.cdd files)
  - Satisfies a requirement (from monolithic implementation)
  - **`SatisfierPropertyKind`**: Operators & predicates to indicate if/how a resource property is "used up"

# Matching Requirements against Resources



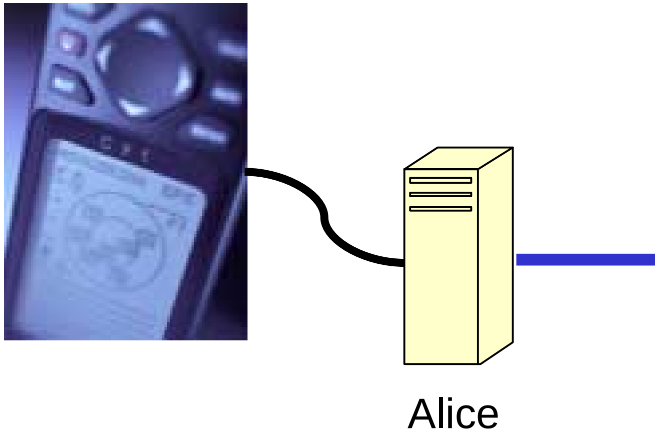
- Generic grammar for defining resources & requirements
- Well-defined, generic matching & accounting algorithm
  - Depending on predicate, resource capacity can be "used up"

# Example Domain



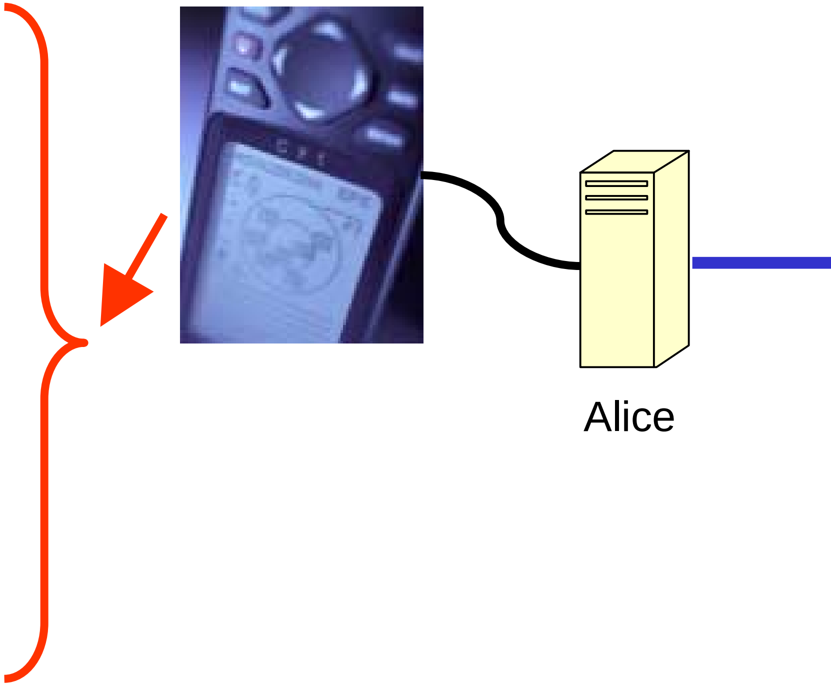
# Domain Descriptor: MyNetwork.cdd (1/3)

```
<?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>
```



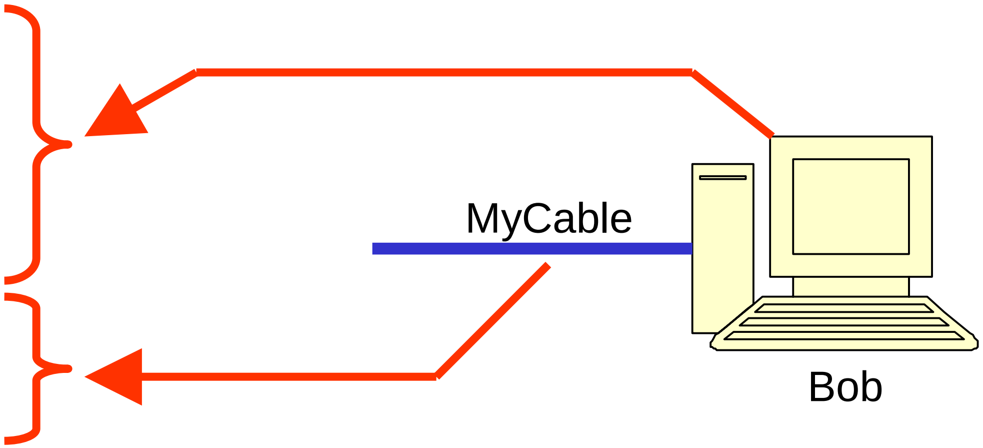
# Domain Descriptor: MyNetwork.cdd (2/3)

```
<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>
```



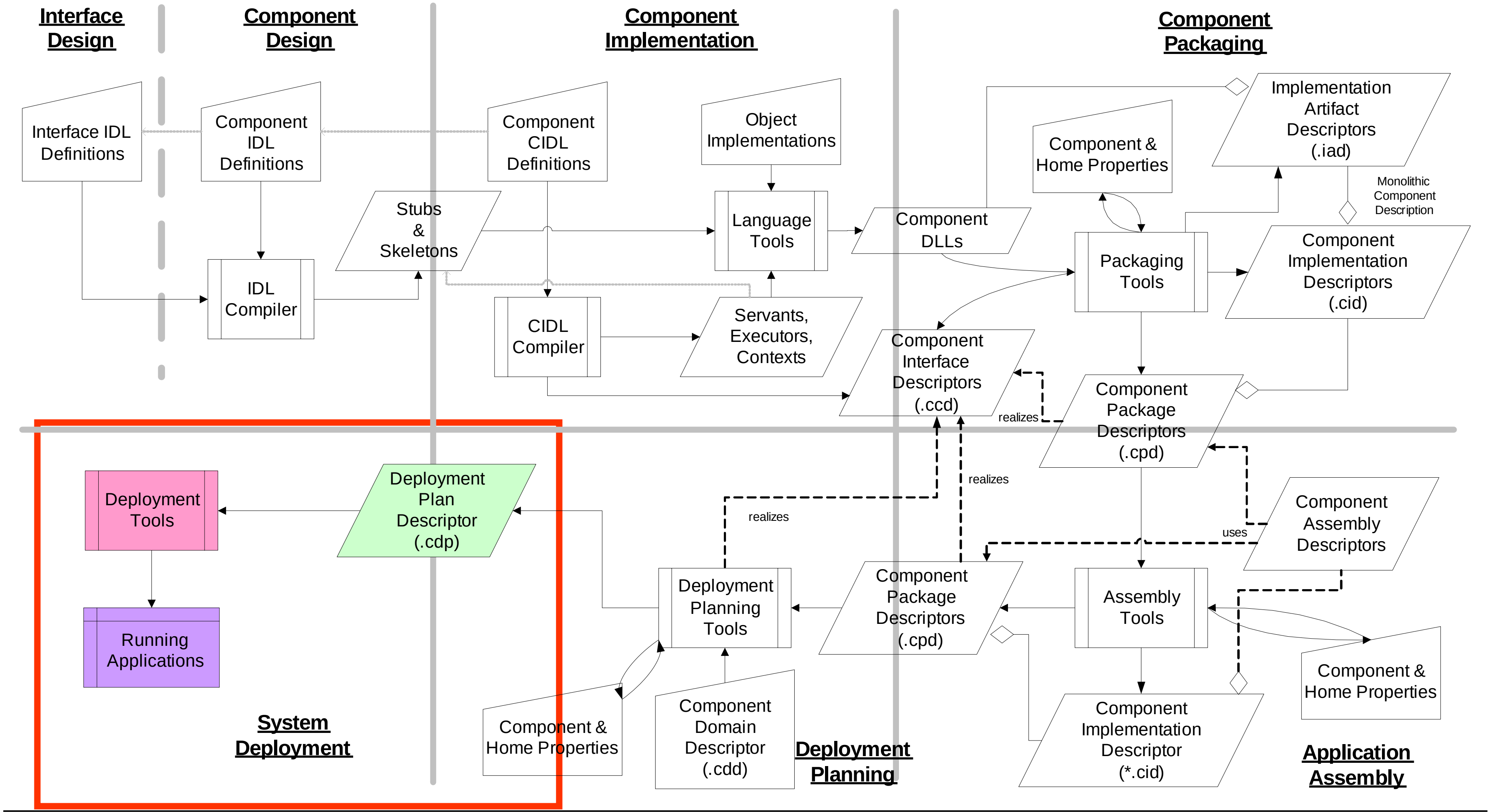
# Domain Descriptor: MyNetwork.cdd (3/3)

```
<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>
```

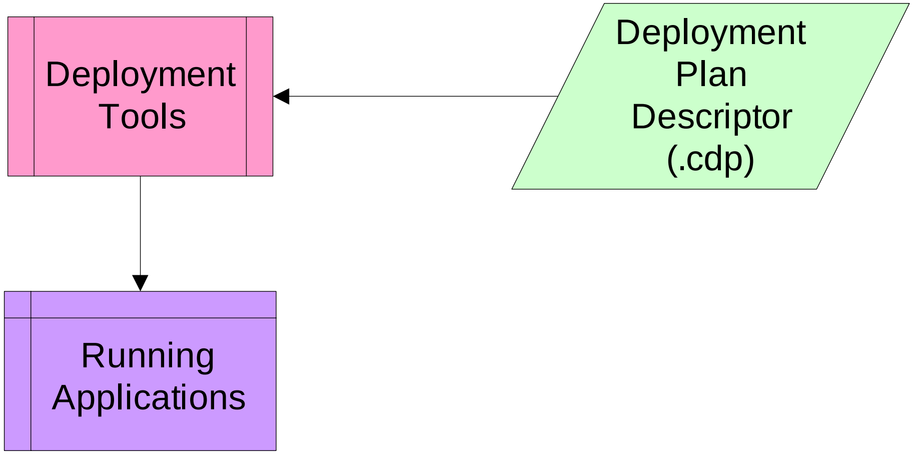


# Deployment

Goal: Deploy/execute application/components according to *deployment plan*


Interface Design
Component Design
Component Implementation
Component Packaging
Interface IDL Definitions
Component IDL Definitions
Component CIDL Definitions
Object Implementations
Implementation Artifact Descriptors (.iad)
Component & Home Properties
Monolithic Component Description
Stubs & Skeletons
Language Tools
Component DLLs
Packaging Tools
Component Implementation Descriptors (.cid)
IDL Compiler
CIDL Compiler
Servants, Executors, Contexts
Component Interface Descriptors (.ccd)
realizes
Component Package Descriptors (.cpd)
Deployment Tools
Deployment Plan Descriptor (.cdp)
realizes
realizes
uses
Component Assembly Descriptors
Running Applications
Deployment Planning Tools
Component Package Descriptors (.cpd)
Assembly Tools
Component & Home Properties
System Deployment
Component & Home Properties
Component Domain Descriptor (.cdd)
Deployment Planning
Component Implementation Descriptor (*.cid)
Application Assembly

# Deployment Infrastructure Overview



- Goals
  - Realize a deployment plan on its target deployment platform
- Deployment phase includes:
  - Performing work in the target environment to be ready to execute the software (such as downloading software binaries)
  - Install component instances into the target environment
  - Interconnecting and configuring component instances

# Deployment Infrastructure Overview (1/2)

- **Repository Manager**
  - Database of components that are available for deployment ("staging area")
- **Target Manager**
  - Re
    available nodes & res s)

“Execution” Runtime Model

- **Execution Manager**
  - Execution of an application according to a “Deployment Plan”
- **Domain Application Manager**
  - Responsible for deploying an application on the domain level

- **Domain Application**
  - Represents a “global” application that was deployed across nodes

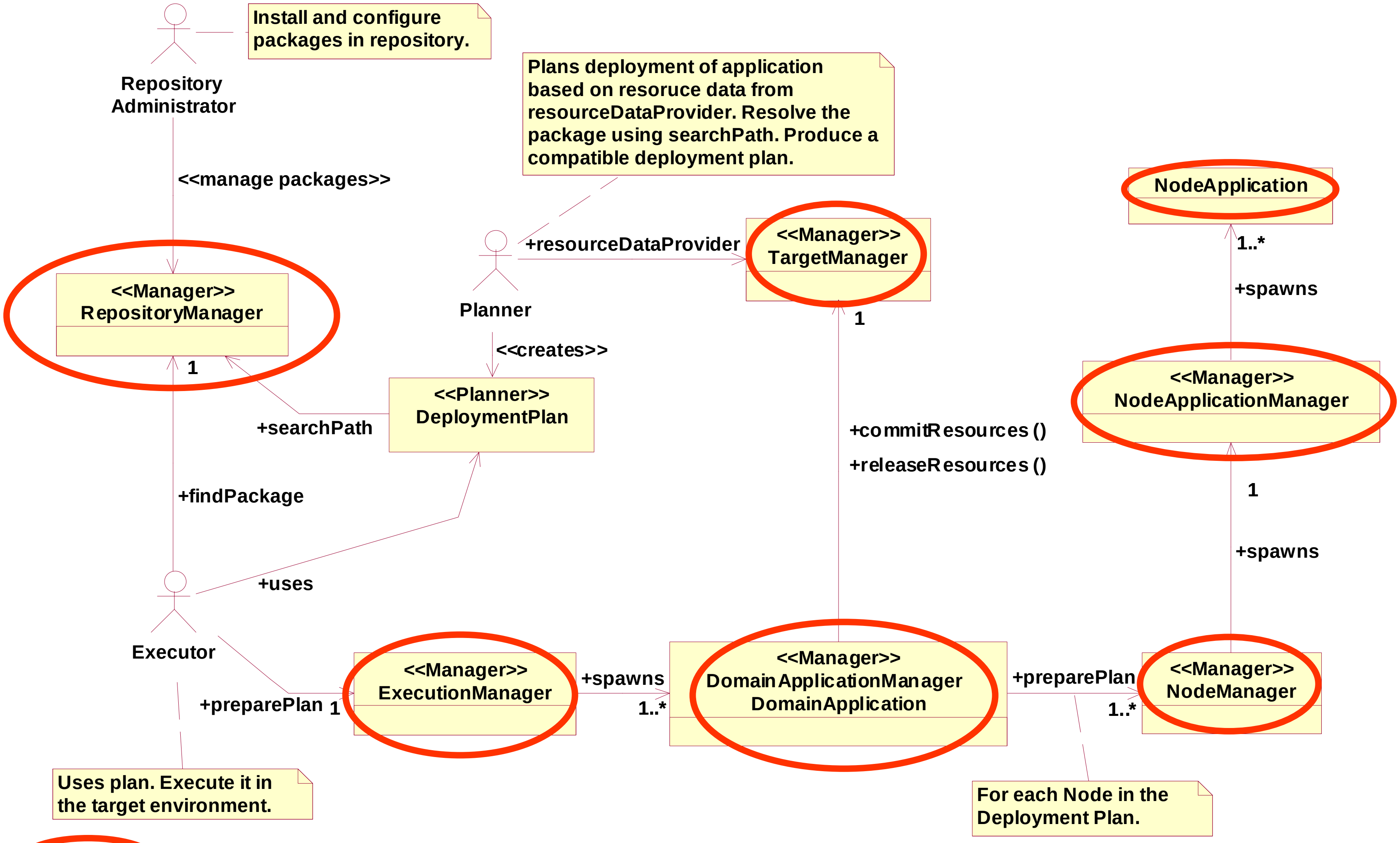
“Component Software” Runtime Model

  - Responsible for managing a partial applications that is limited

“Target” Runtime Model

- **Node Application Manager**
  - Responsible for deploying an locality constrained application onto a node
- **Node Application**
  - Represents a piece of an application that is executing within a single node

# Deployment Infrastructure Overview (2/2)



Repository Administrator

Install and configure packages in repository.

<<manage packages>>

<<Manager>> RepositoryManager

1

+searchPath

+findPackage

Plans deployment of application based on resoruce data from resourceDataProvider. Resolve the package using searchPath. Produce a compatible deployment plan.

Planner

+resourceDataProvider

<<Manager>> TargetManager

1

<<creates>>

<<Planner>> DeploymentPlan

+commitResources ()

+releaseResources ()

NodeApplication

1..*

+spawns

<<Manager>> NodeApplicationManager

1

+spawns

Executor

+uses

+preparePlan 1

<<Manager>> ExecutionManager

+spawns

1..*

<<Manager>> DomainApplicationManager DomainApplication

+preparePlan

1..*

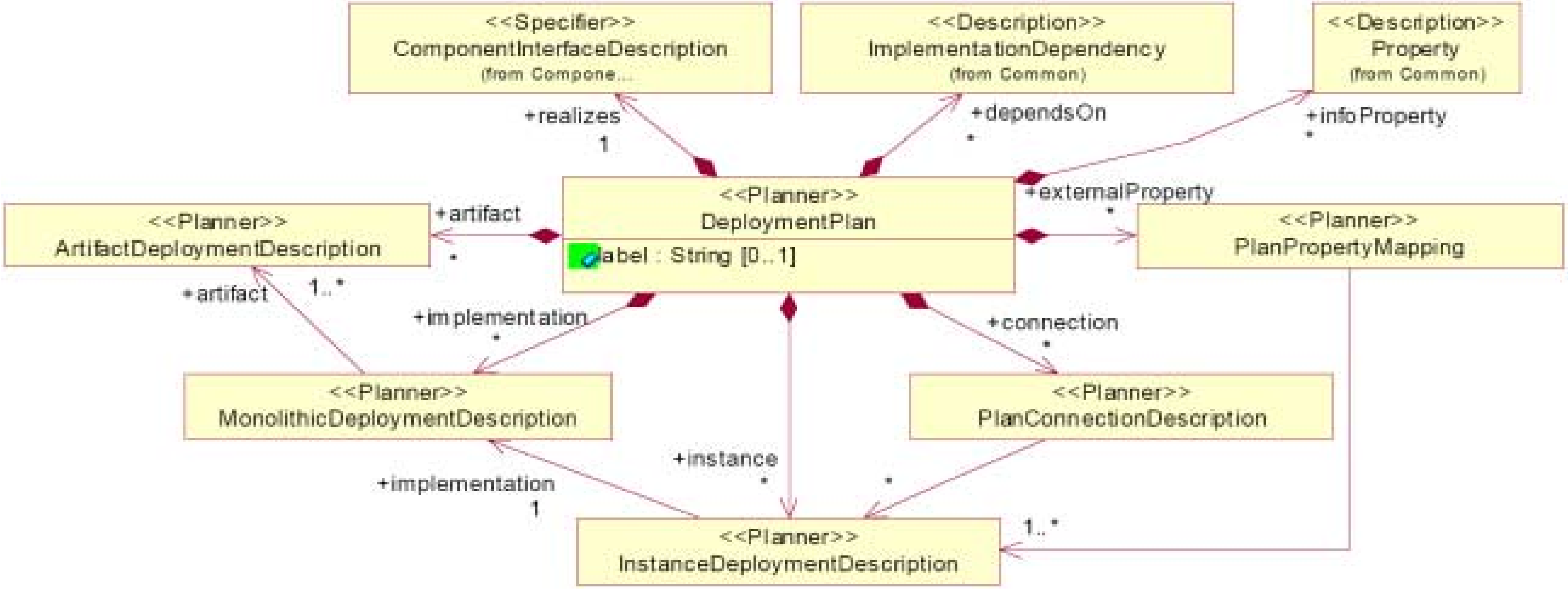
<<Manager>> NodeManager

Uses plan. Execute it in the target environment.

For each Node in the Deployment Plan.

Infrastructure (Services)

# Deployment Plan (1/2)



- Self-contained IDL data structure (struct type) for executing an application within a specific domain, based on a specific set of resources
  - Records all decisions made during planning, e.g., implementation selection, instance to node assignment, resource allocation, etc.
  - “Flat” assembly of instances of components with monolithic implementations (all assemblies are resolved)

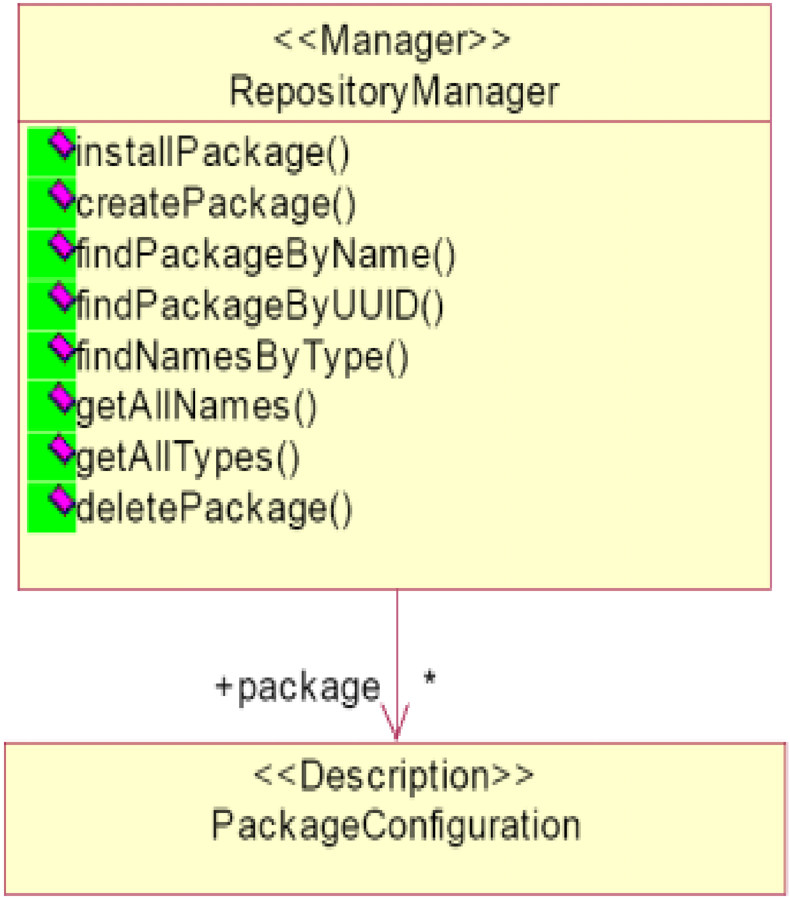
# Deployment Plan (2/2)

```
struct DeploymentPlan {
    string label;
    string UUID;
    ComponentInterfaceDescription realizes;
    MonolithicDeploymentDescriptions implementation;
    InstanceDeploymentDescriptions instance;
    PlanConnectionDescriptions connection;
    PlanPropertyMappings externalProperty;

    ImplementationDependencies dependsOn;
    ArtifactDeploymentDescriptions artifact;
    Properties infoProperty;
};
```

- A deployment plan 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

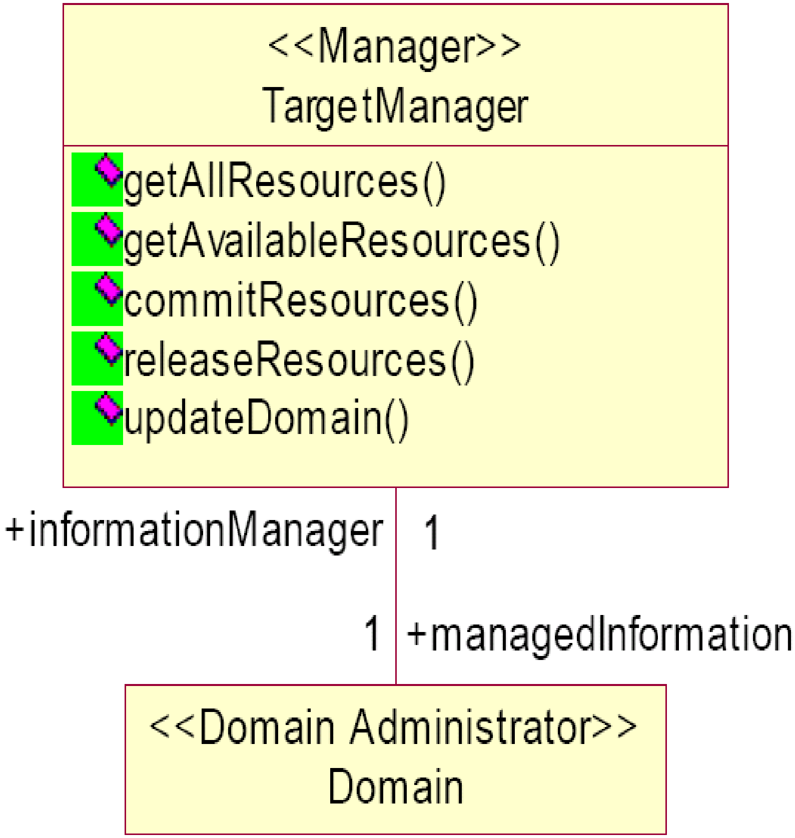
# Deployment Infrastructure: Repository Manager

- Database of applications
  - Metadata (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 metadata in XML format & artifacts
- CORBA interface for installation of packages retrieval, & introspection of metadata
- HTTP interface for downloading artifacts
  - Used by Node Managers during execution



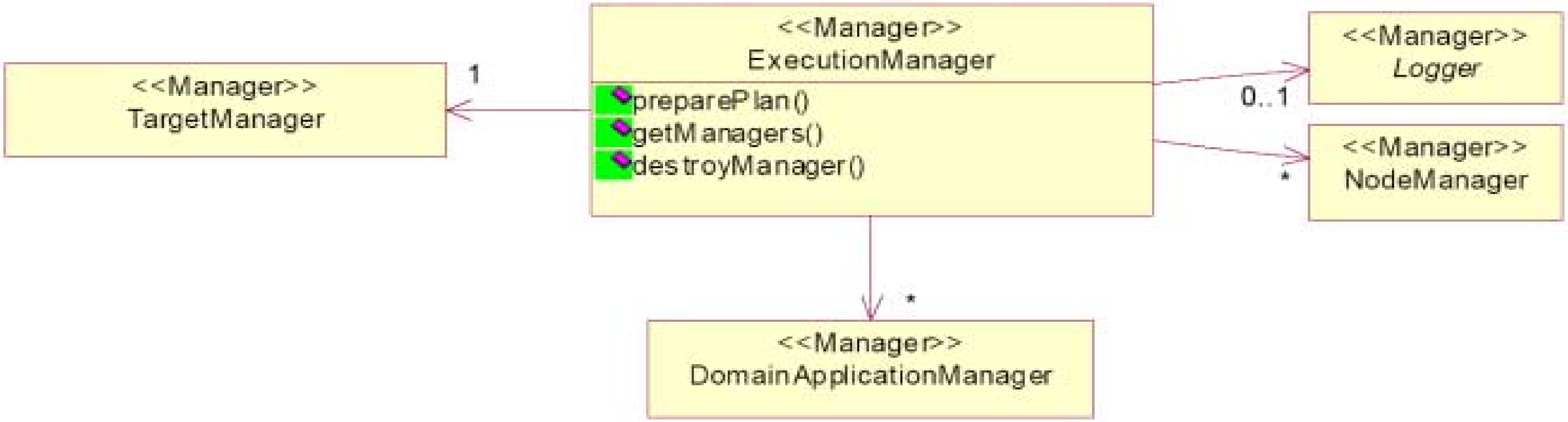
# Deployment Infrastructure: Target Manager

- Singleton service, i.e., one *TargetManager* per domain
- Retrieval of available or total resource capacities
- Allocation & release of resources (during application deployment)
- No “live” monitoring of resources implied (optional)
  - Assumption: all resources are properly allocated & released through this interface



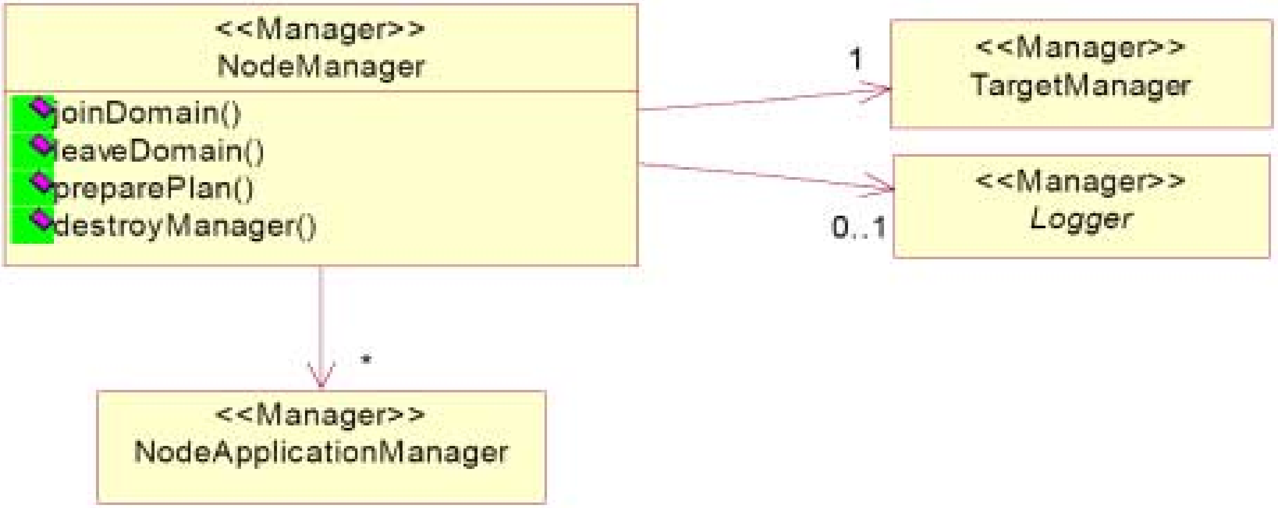
- Allows “off-line” scenarios where the possibility & the effect of deploying applications is analyzed
  - e.g., “Given this configuration, is it possible to run this set of applications concurrently? How?”

# Deployment Infrastructure: Execution Manager



- Singleton service, i.e., one *ExecutionManager* per domain
- A “daemon-like” process always running on each domain
- User-visible front-end for executing a global (domain-level) deployment plan
  - Deployment plan results from planning for the deployment of an application, based on a specific set of nodes & resources
- Has information on all NodeManagers in the domain.
- Instructs *NodeManagers* to execute respective per-node pieces of an application

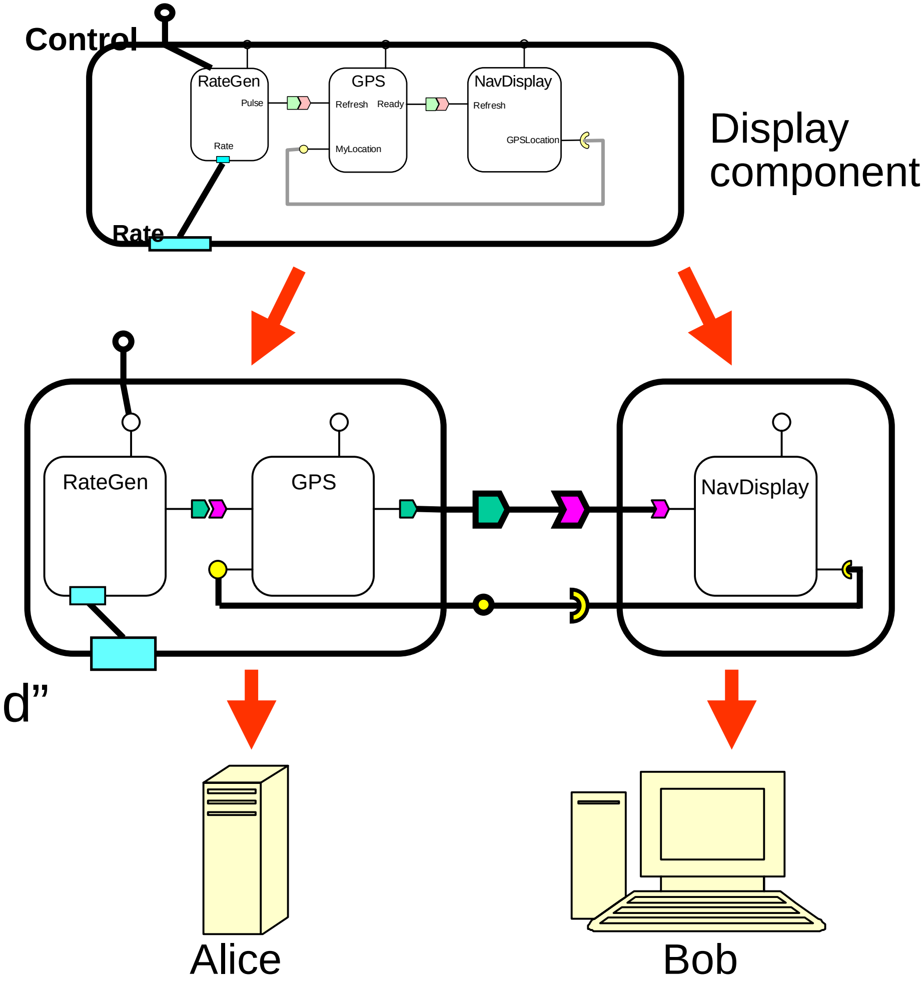
# Deployment Infrastructure: Node Manager



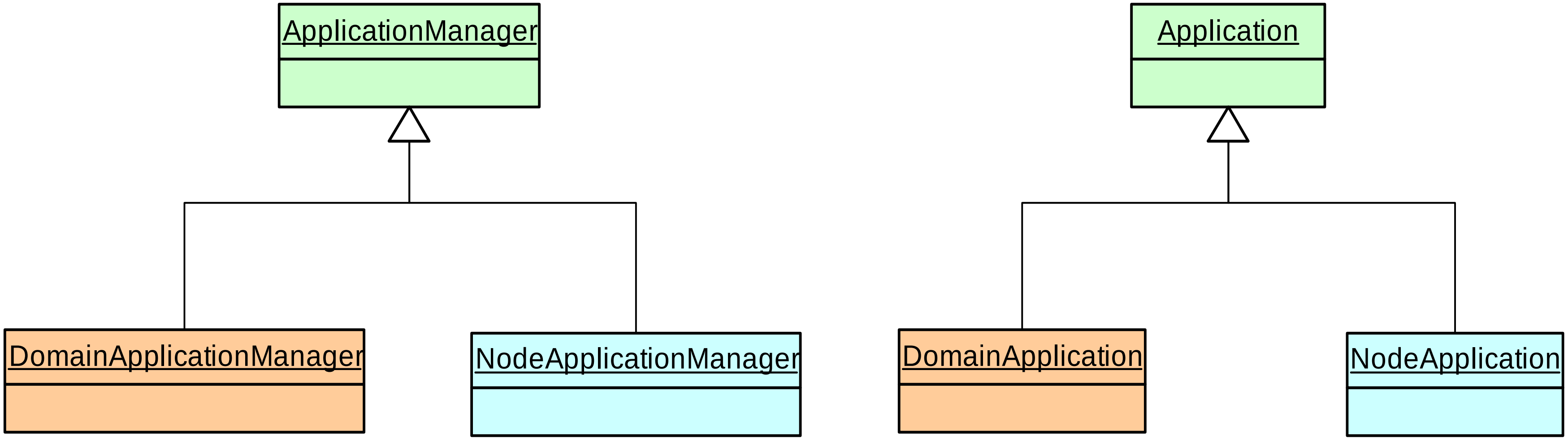
- Mirrors the *ExecutionManager*, but is limited to one node only
- A “daemon-like” process that is always running on each individual node
- Responsible for deploying local (node-level) deployment plan

# Execution/Node Managers Interaction

- *ExecutionManager* 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 *NodeManager*
  - Can be done concurrently
- *ExecutionManager* then sends “provided” references to their users
- Transparent to “Executor” user

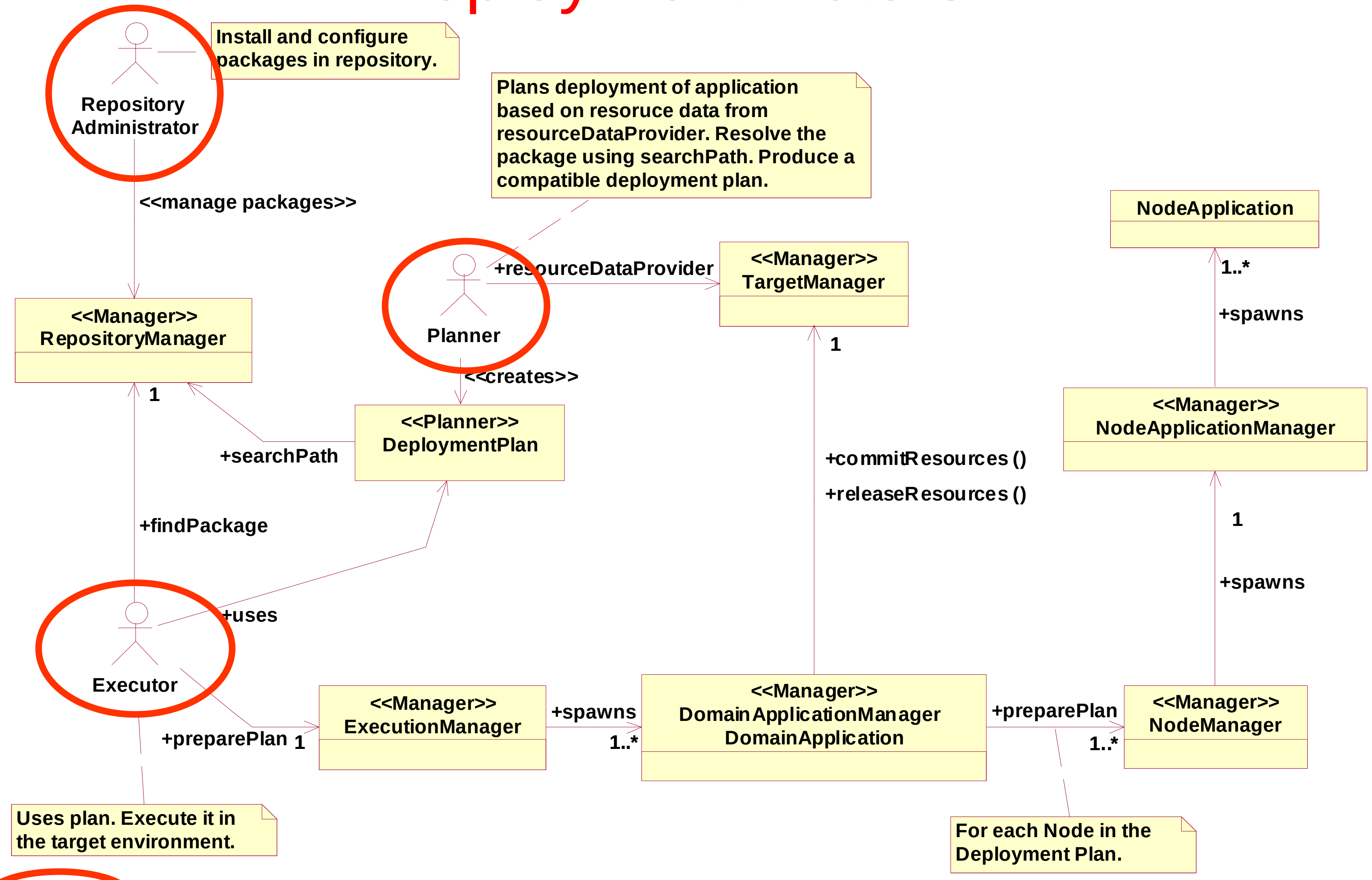


# Launch Application: Domain vs. Node



- ***Domain**** provides functionality at the domain level
- ***Node**** provides similar functionality, but restricted to a Node
- **ApplicationManager**
  - `startLaunch()` & `destroyApplication()` operations
- ***Application***
  - `finishLaunch()` & `start()` operations

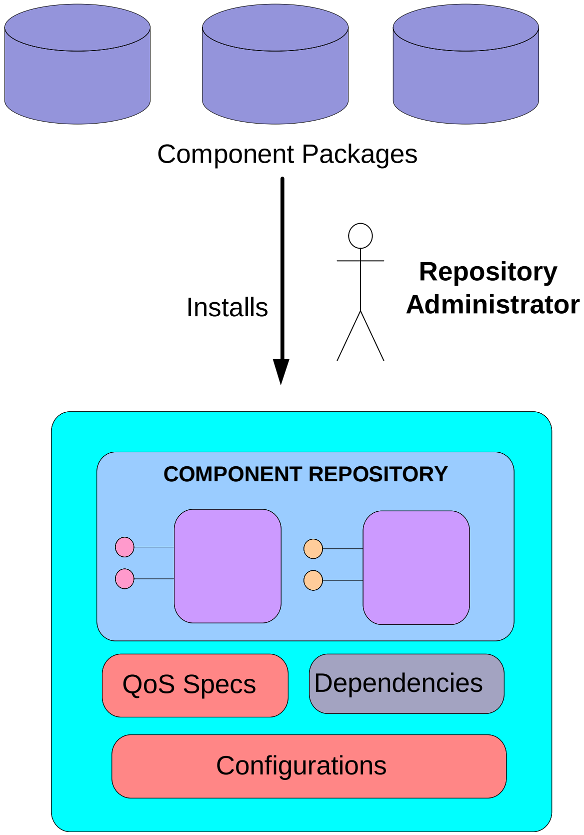
# Deployment Actors


Repository Administrator
Install and configure packages in repository.
Plans deployment of application based on resoruce data from resourceDataProvider. Resolve the package using searchPath. Produce a compatible deployment plan.
<<manage packages>>
Planner
+resourceDataProvider
<<Manager>> TargetManager
NodeApplication
1..*
+spawns
<<Manager>> RepositoryManager
1
<<creates>>
<<Planner>> DeploymentPlan
+searchPath
1
+commitResources ()
+releaseResources ()
<<Manager>> NodeApplicationManager
1
+findPackage
+spawns
+uses
Executor
<<Manager>> ExecutionManager
+spawns
<<Manager>> DomainApplicationManager DomainApplication
+preparePlan
<<Manager>> NodeManager
+preparePlan 1
1..*
1..*
Uses plan. Execute it in the target environment.
For each Node in the Deployment Plan.

Actors — usually, humans aided by software tools

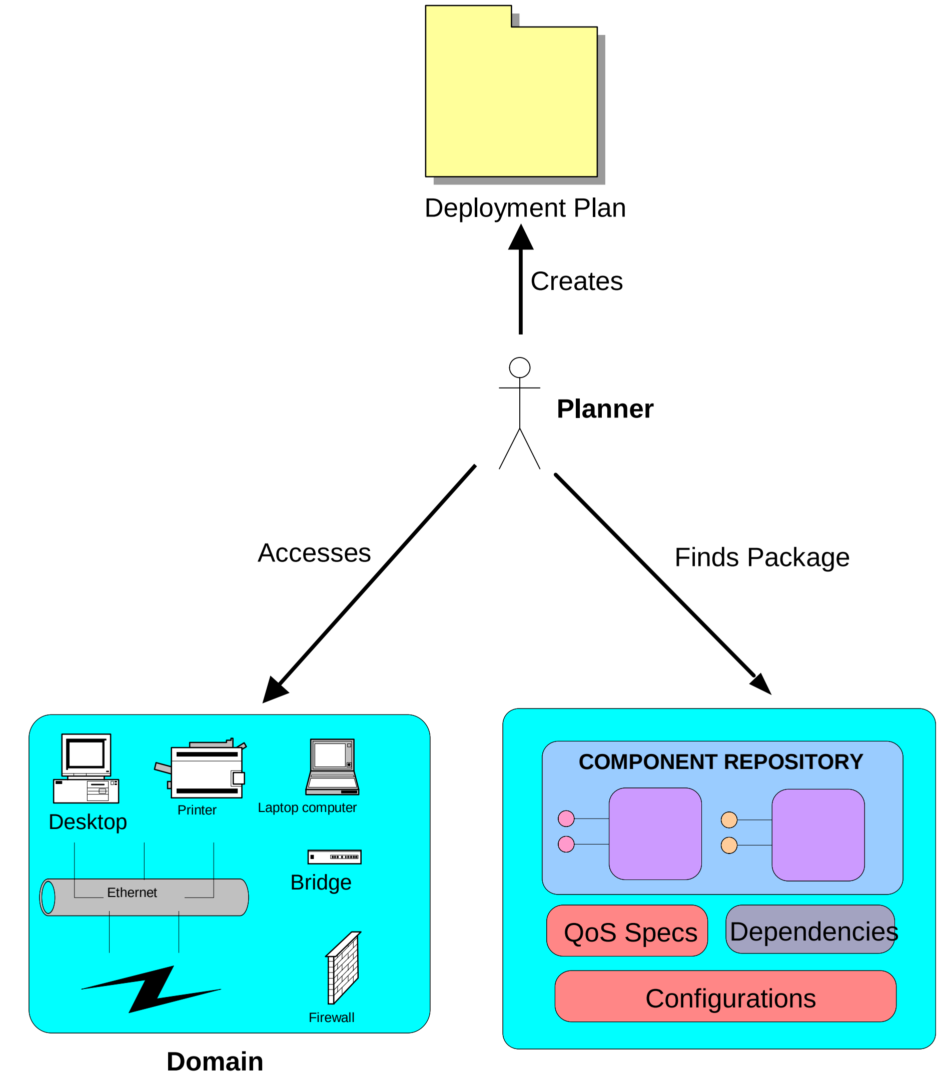
# Deployment Actors: Repository Administrator

- 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 …



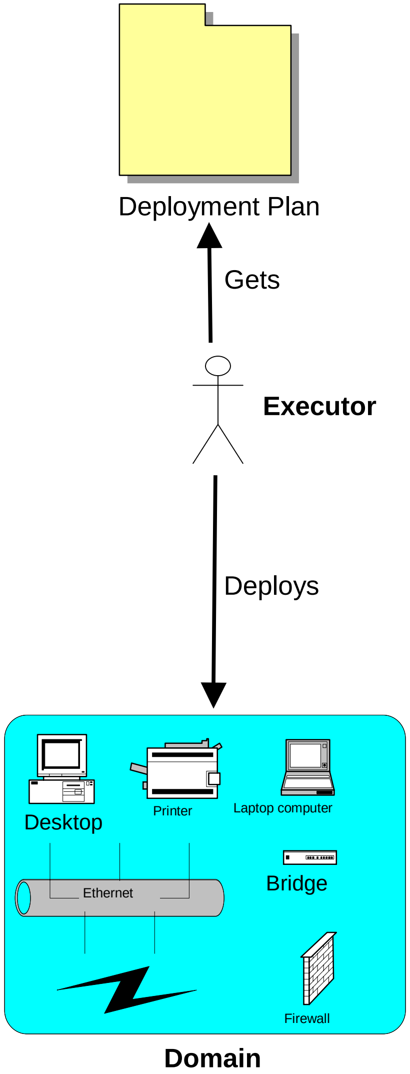
# Deployment Actors: Planner

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

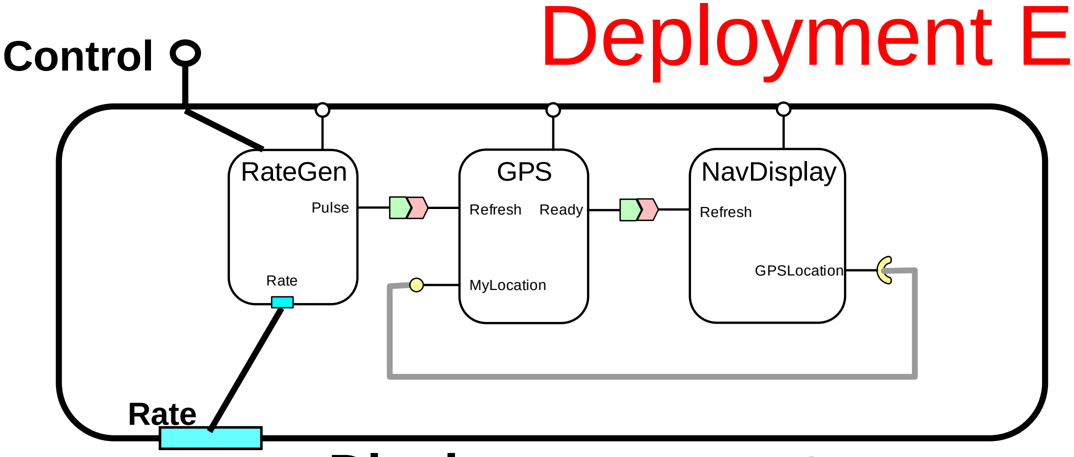


# Deployment Actors: Executor

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



# Deployment Example



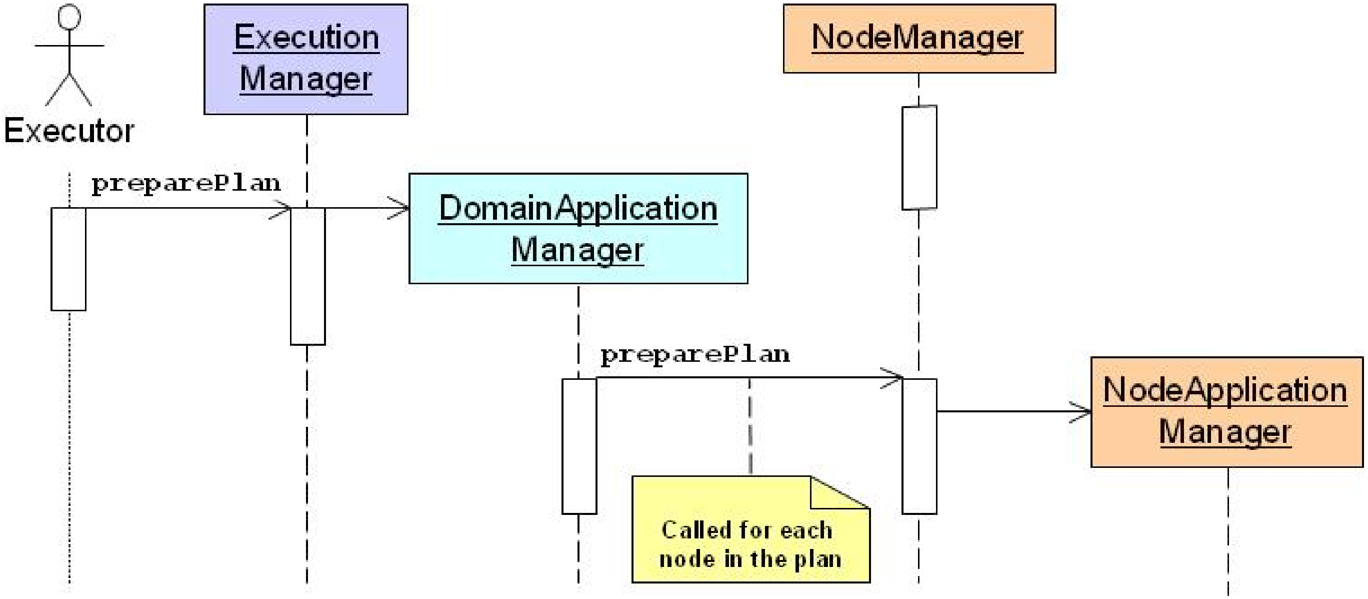
**Display component**

```
<Deployment:DeploymentPlan …
  <label>Display Deployment Plan</label>
  <instance xmi:id="RateGen_Instance">
    <name>RateGen_Instance</name>
    <node>Alice</node>
  </instance>
  <instance xmi:id="GPS_Instance">
    <name>GPS_Instance</name>
    <node>Alice</node>
  </instance>
  <instance xmi:id="NavDisplay_Instance">
    <name>NavDisplay_Instance</name>
    <node>Bob</node>
  </instance>
</Deployment:DeploymentPlan>
```

Mapping components to nodes

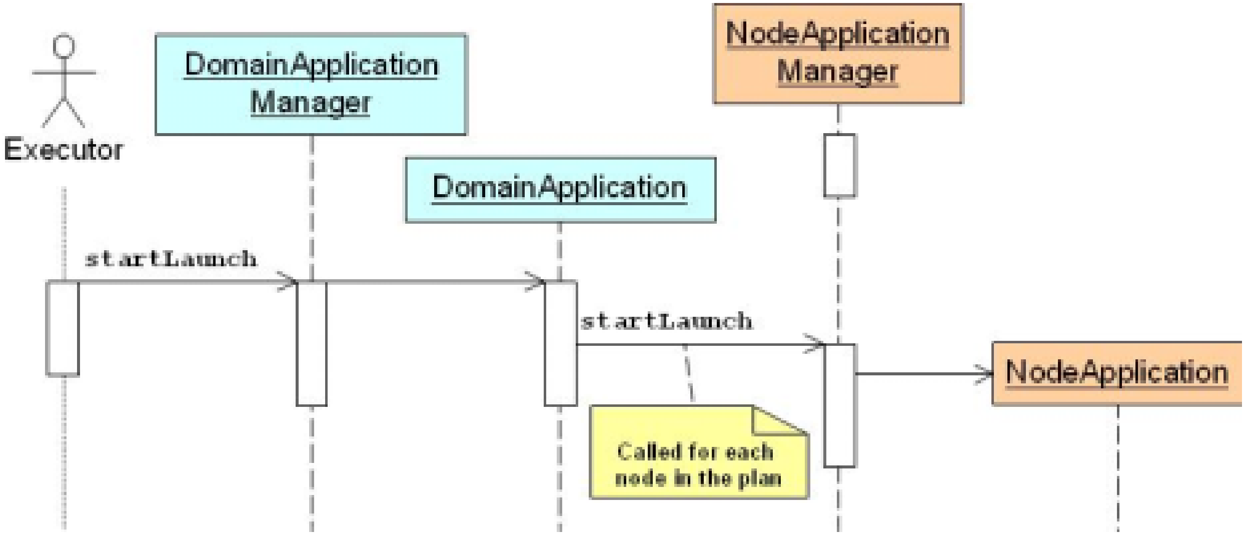
- Recall that the **`Display`** component is an *assembly component*
- When we deploy it, only the "monolithic" components will be actually deployed
- "*Deployer actor*" can specify which "monolithic" component(s) maps to which nodes, as specified by the *ComponentDeploymentPlan (.cdp)* descriptor
- We display 3 component to 2 nodes

# Deployment Example: Prepare Plan



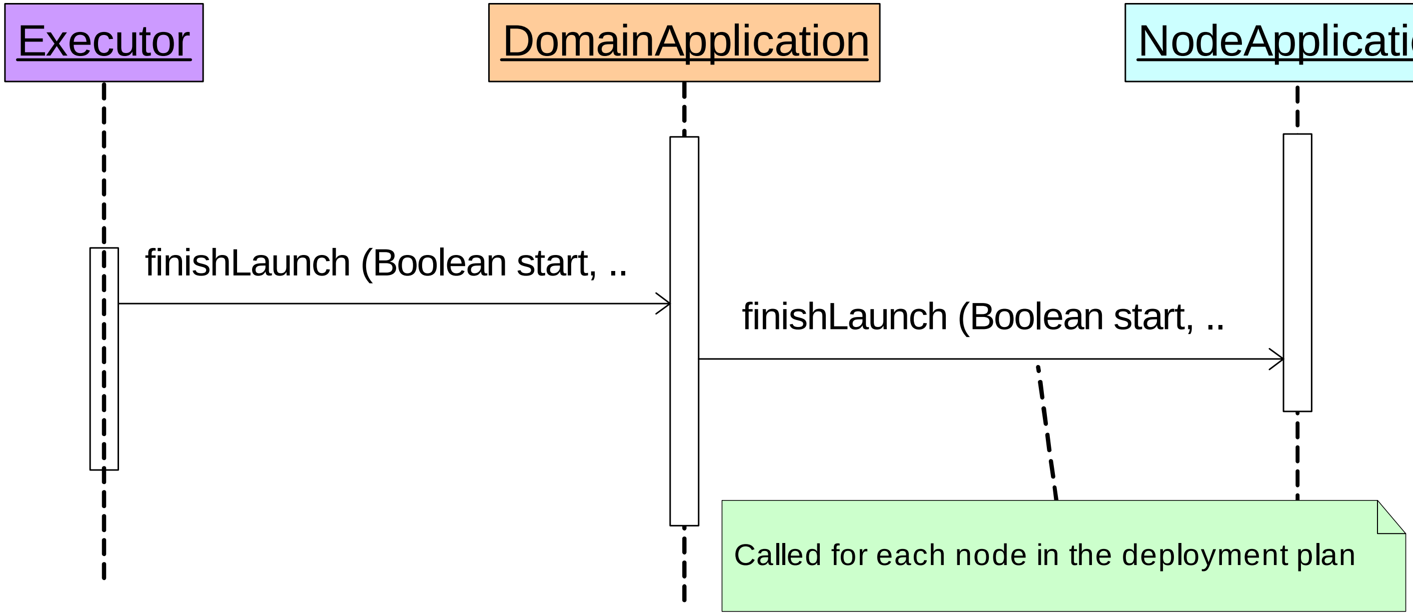
- Before calling **`preparePlan()`**, *ExecutionManager* should be running & 2 *NodeManagers* should be running on Alice and Bob nodes
- Component Packages are retrieved from the *Component Repository*
- *RepositoryManager* parses XML metadata into an in-memory representation
- *RepositoryManager* creates global deployment plan & passes it to *ExecutionManager* to **`preparePlan()`**, which delegates to *DomainApplicationManager*
- *DomainApplicationManager* splits it into multiple local plans
- Contacts the 2 *NodeManager* residing in Alice & Bob nodes to create appropriate *NodeApplicationManagers* & dispatch individual local plans

# Deployment Example: Start Launch



- *Executor* initiates launching of the application
- *DomainApplicationManager* creates a *DomainApplication* object
  - Facilitates application launch by contacting individual *NodeApplicationManagers*
- *NodeApplicationManagers* residing in Alice & Bob nodes will create a *NodeApplication* individually.

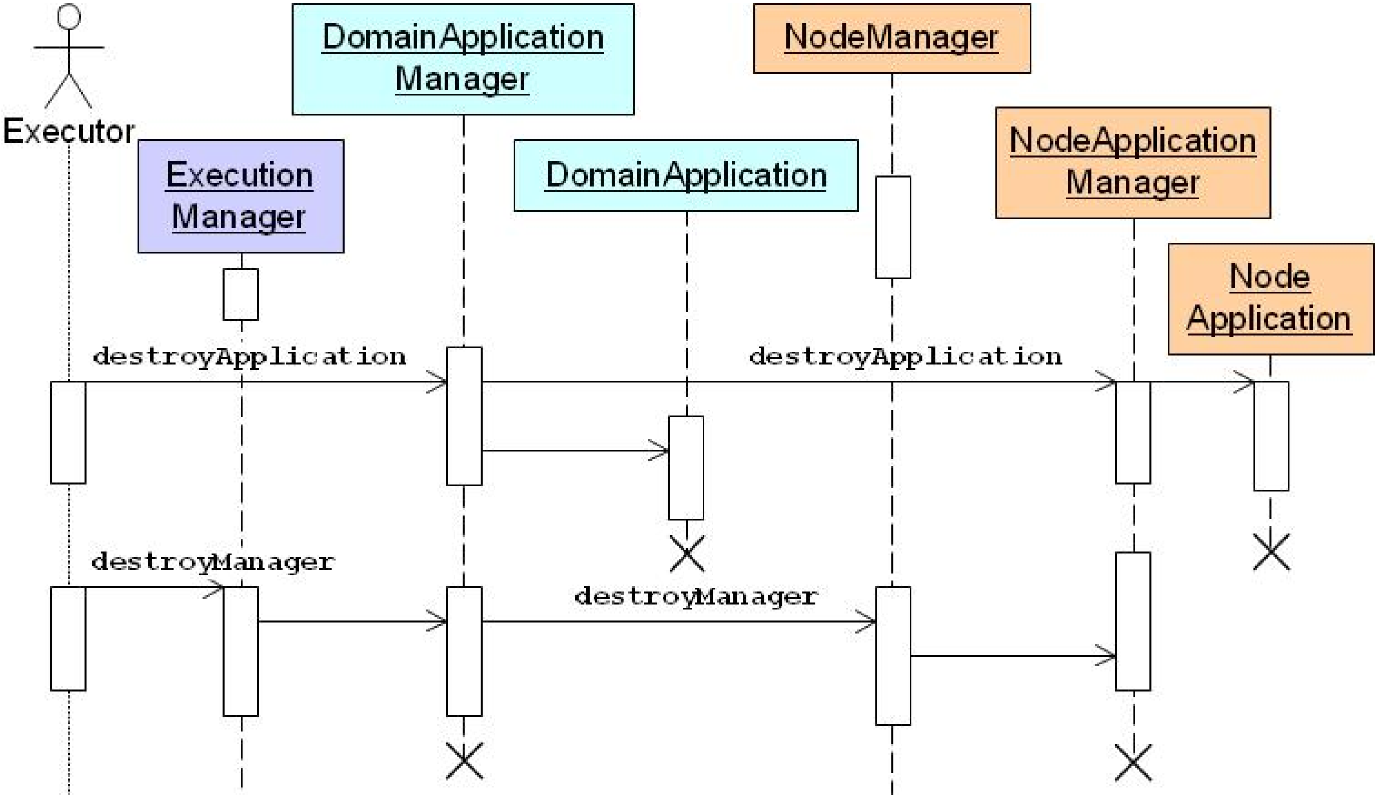
# Deployment Example: Finish Launch & Start


Executor
DomainApplication
NodeApplication
finishLaunch (Boolean start, ..
finishLaunch (Boolean start, ..

Called for each node in the deployment plan

- Executor notifies *DomainApplication* of completion of application launch
- *DomainApplication* notifies *NodeApplications* running on Alice & Bob nodes to complete application launch
- Connections between components are made at this stage
- Optional "*start*" parameter could be given to indicate whether actually "start" the application (i.e., **`SetSessionContext()`**, etc)

# Deployment Example: Application Teardown



- *Executor* initiates tear-down by first terminating running applications under its control
  - *DomainApplicationManager* ensures tear down of *NodeApplications* running on both Alice & Bob nodes
- It then tears down both managers in Alice & Bob nodes

# 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” metadata

# 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 & Deployment spec enhanced to support:
    - Hierarchical assemblies, allowing better component reuse
    - Resource management
    - Automated distribution & deployment
- Well-defined inter-vendor boundaries
  - Planner & 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

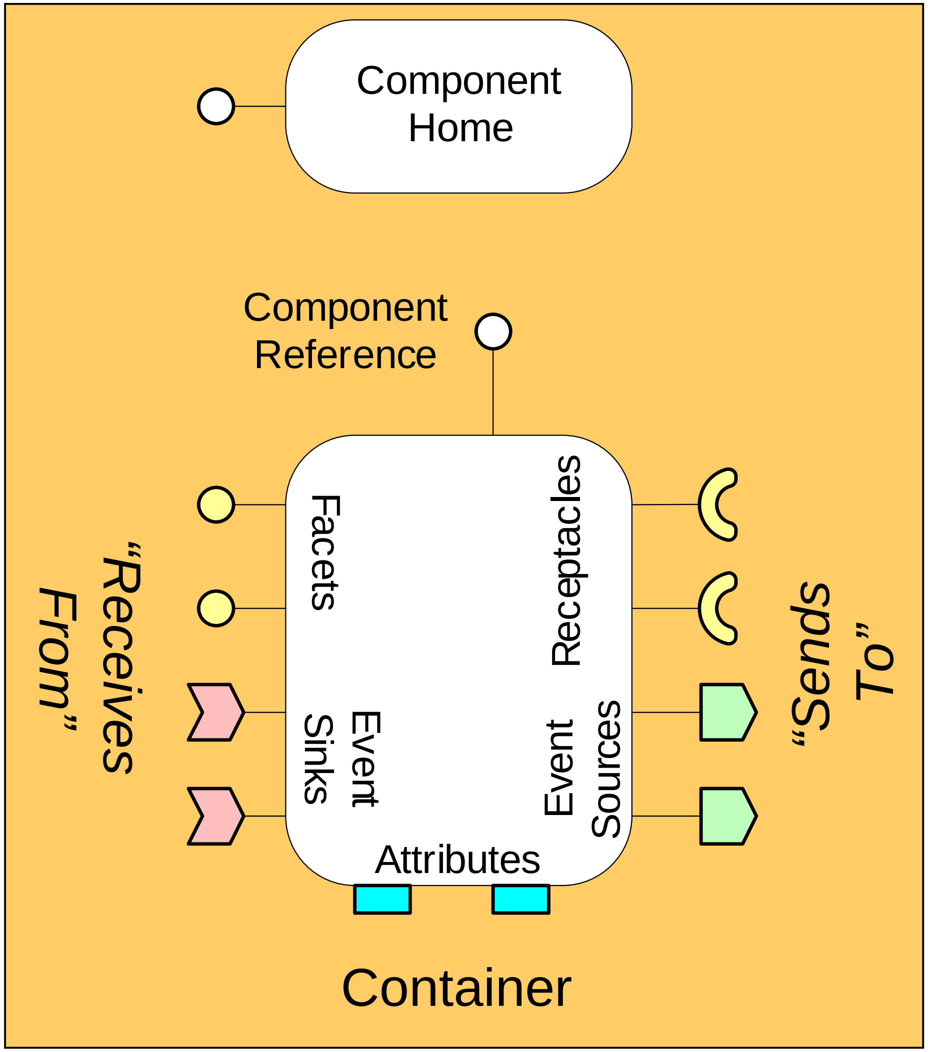
www.omg.org/cgi-bin/doc?realtime/2003-05-05

# Motivation for Lightweight CCM (LwCCM)

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

# CCM Features Retained in LwCCM Subset

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



# CCM Features Excluded from LwCCM Subset

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

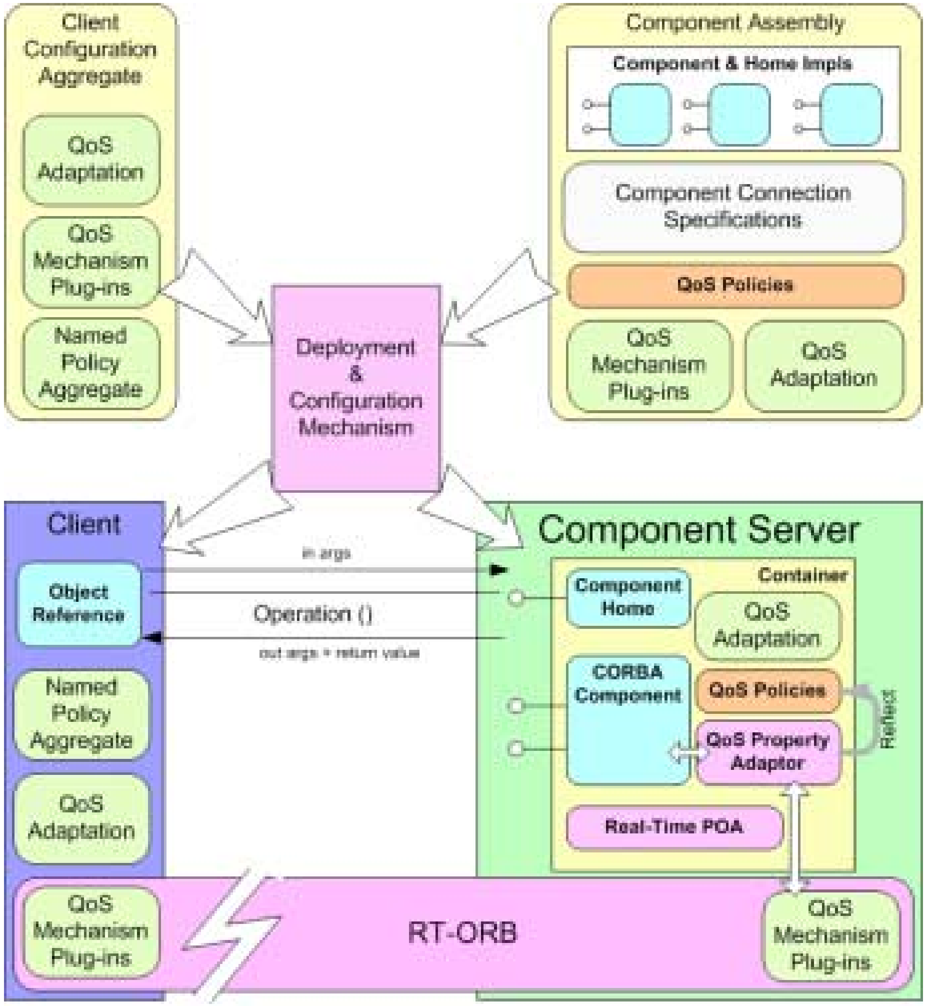
Lightweight CCM should be treated like Minimum CORBA, i.e., *advisory*

# Overview of CIAO & Future R&D Directions

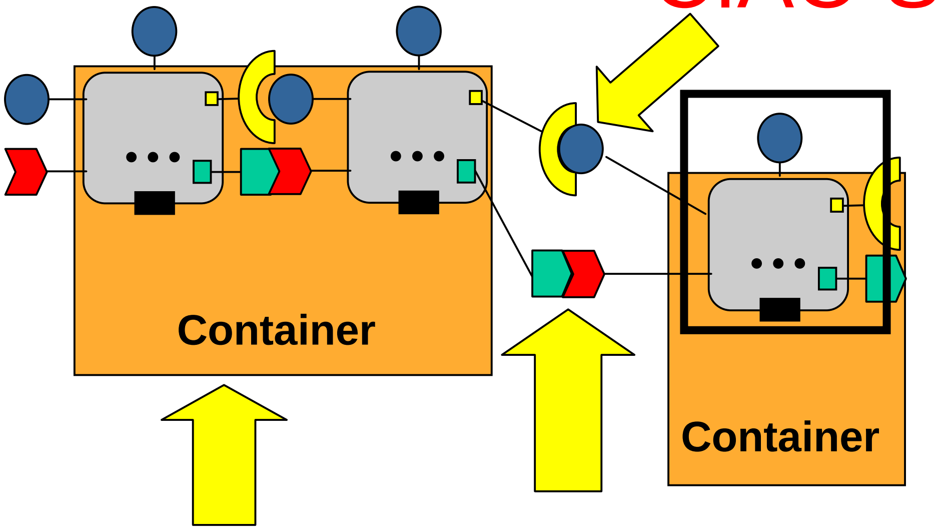
# Overview of CIAO

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

  deuce.doc.wustl.edu/Download.html



# CIAO Status



- Components can be built as shared libs or static libs
- Component server supported
- Tools to install, host, load, & manage component implementations are available
- The CIAO Deployment & Configuration (D&C) framework provides support for component assemblies in compliance with ptc/02-08-03
- CIAO also supports real-time extensions
  - www.cs.wustl.edu/~schmidt/CIAO.html

- Support for IDL 3 (**`component, home`** & related keywords) & many CIDL features have been added
- Support for all types of ports: facets (**`provides`**), receptacles (**`uses, uses multiple`**), event sources (**`emits, publishes`**) & event sinks (**`consumes`**)
- Support for the Session container via CIDL compiler

# CIAO Next Steps

- **Deployment & Configuration (Leads: Kitty Balasubramanian & Jai Balasubramanian)**
  - Implementing the new deployment & configuration specification, ptc/03-07-02, necessary for DARPA ARMS program
  - Changes to the deployment & assembly toolset to support lightweight components, as prescribed by ptc/04-02-03
- **Core CCM Infrastructure (Leads: Gan Deng & Will Otte)**
  - Additional support for Real-time CORBA Policies at the ORB level & object level
    - i.e., at the object reference level of a component receptacle
  - Integration of different event propagation mechanisms (such as Event Channel & Notification Services) within the container
  - Compliant with Lightweight CCM specification
- **Modeling tool support for CIAO (Leads: Andy Gokhale & Jeff Parsons)**
  - See www.dre.vanderbilt.edu/cosmic for details

# How to Learn about CCM & CIAO Programming

- Examples available with the distribution
  - **`CIAO/docs/tutorial/Hello`**, a simple example that illustrates the use of some basic CCM concepts
  - **`CIAO/examples/OEP/BasicSP`**
    - A simple example that shows the interaction between 4 components
  - **`CIAO/examples/OEP/Display`**
    - Similar to the BasicSP, but has an additional feature showing integration with Qt toolkit
- Step-by-step to create and deploy components based on CIAO available at
  - **`CIAO/examples/Hello`**
- “Quick CORBA 3”, Jon Siegel, John Wiley and Sons  provides a quick start

# 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 run-time 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 on CORBA & CCM

OMG specifications pertaining to CCM

- CORBA Component Model (CCM)
  - ptc/02-08-03
- Lightweight CCM
  - ptc/04-02-03
- 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/05-01-07

Books pertaining to CCM

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

Web resources pertaining to CCM

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