CORBA Component Model

Alma Mater Studiorum - Università di Bologna
CdS Laurea Magistrale in Ingegneria Informatica
I Ciclo - A.A. 2017/2018

Corso di Sistemi Distribuiti M (8 cfu)

08 – Altro Modello Container Pesante: Cenni di CORBA Component Model (CCM)

Docente: Paolo Bellavista
paolo.bellavista@unibo.it

http://lia.disi.unibo.it/Courses/sd1718-info/
http://lia.disi.unibo.it/Staff/PaoloBellavista/

CORBA Component Model (CCM)

Giusto per rammentare che EJB3.x non è ***unico modello a componenti distribuiti*** basato sull'approccio a ***container pesante…***

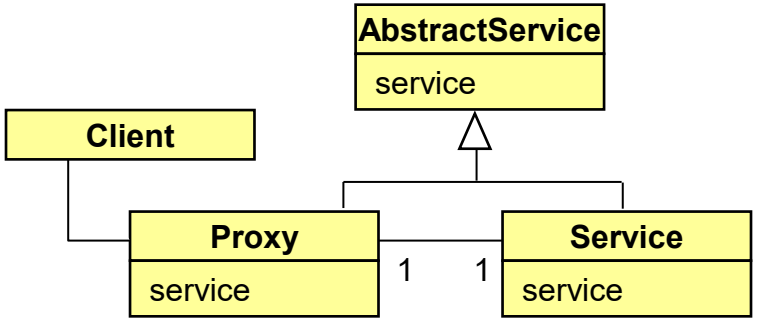
- CORBA come ***potente container per oggetti*** distribuiti
  - Necessità di ***funzionalità ulteriori*** (evoluzione ed estensione degli oggetti CORBA)
  - Verso un modello a ***componenti distribuiti per CORBA***

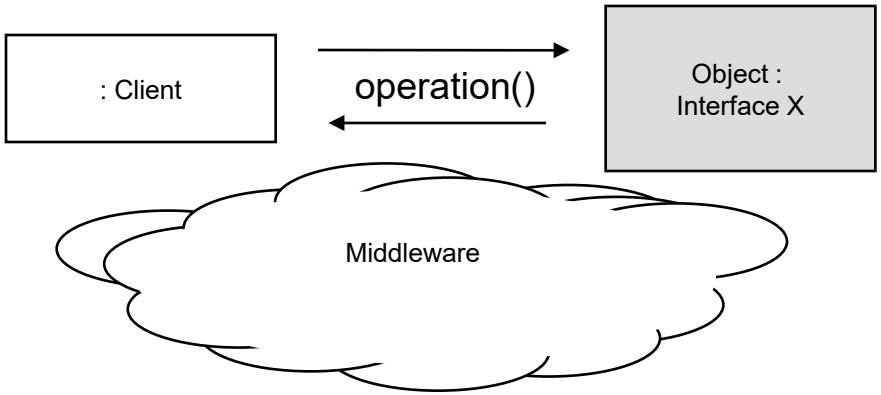
Verso CORBA Component Model (CCM)

Distributed Object Computing (DOC) middleware è già un buon passo:

- Applica ***pattern broker*** per astrarre da dettagli OS&protocol-specific
- Crea sistemi distribuiti che sono più semplici da modellare e costruire grazie all'uso di ***tecniche OO***

Ad es. CORBA, Java RMI, etc.

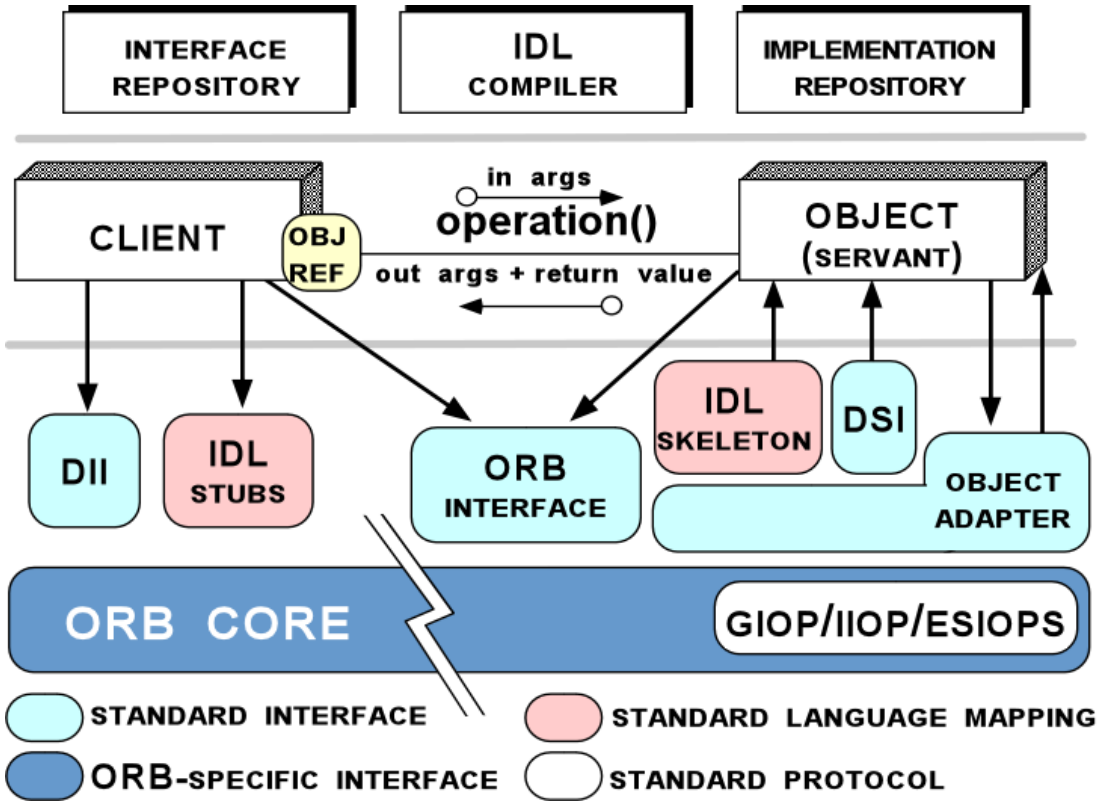




Vi ricordate CORBA 2.x Standard, vero?

CORBA 2.x è un DOC middleware che permette alle applicazioni di essere trasparenti ai dettagli e alle dipendenze specifiche delle piattaforme eterogenee sottostanti

ad es. linguaggi programmazione, OS, protocolli di rete



CORBA 2.x automatizza

- ***Object location***
- ***Connection&memory mgmt***
- ***Parameter (de)marshaling***
- ***Request demultiplexing***
- ***Error handling&fault tolerance***
- ***Object/server activation***
- ***Concurrency&synch-ronization***
- ***…***

Esempio di IDL CORBA 2.x

CORBA 2.x IDL permette di specificare ***interfacce*** che contengono ***operazioni***

Le interfacce vengono mappate su oggetti in linguaggi OO

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

IDL

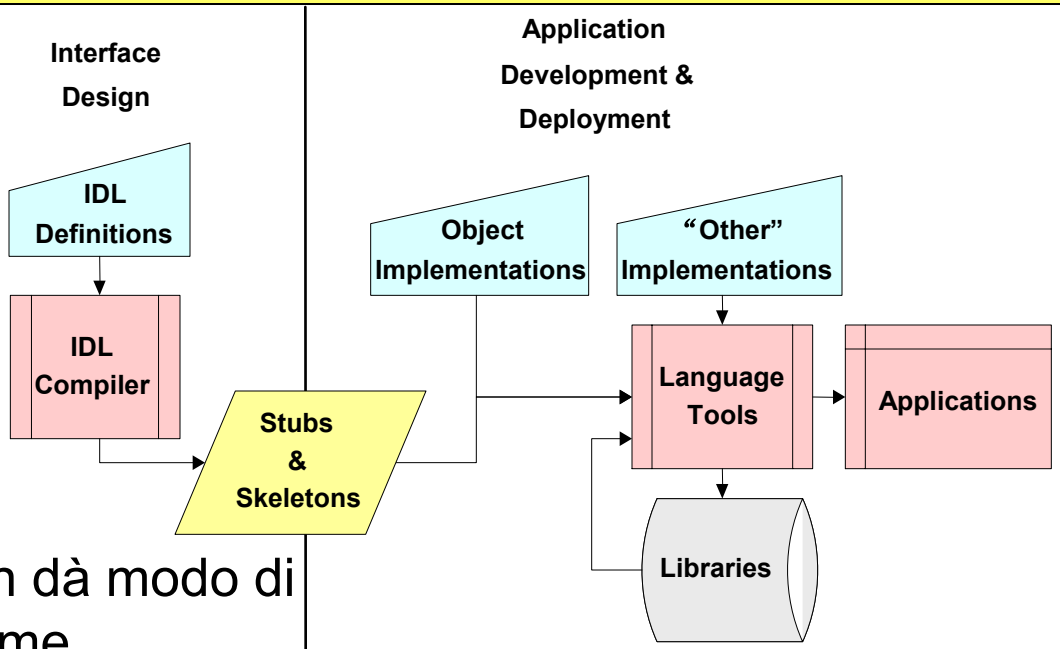
C++

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

Le operazioni definite nelle interfacce possono essere invocate ***sia su oggetti locali che remoti***

Punti Deboli di DOC-based CORBA 2.x Middleware

Sviluppo di applicazioni in CORBA 2.x è fortemente error-prone



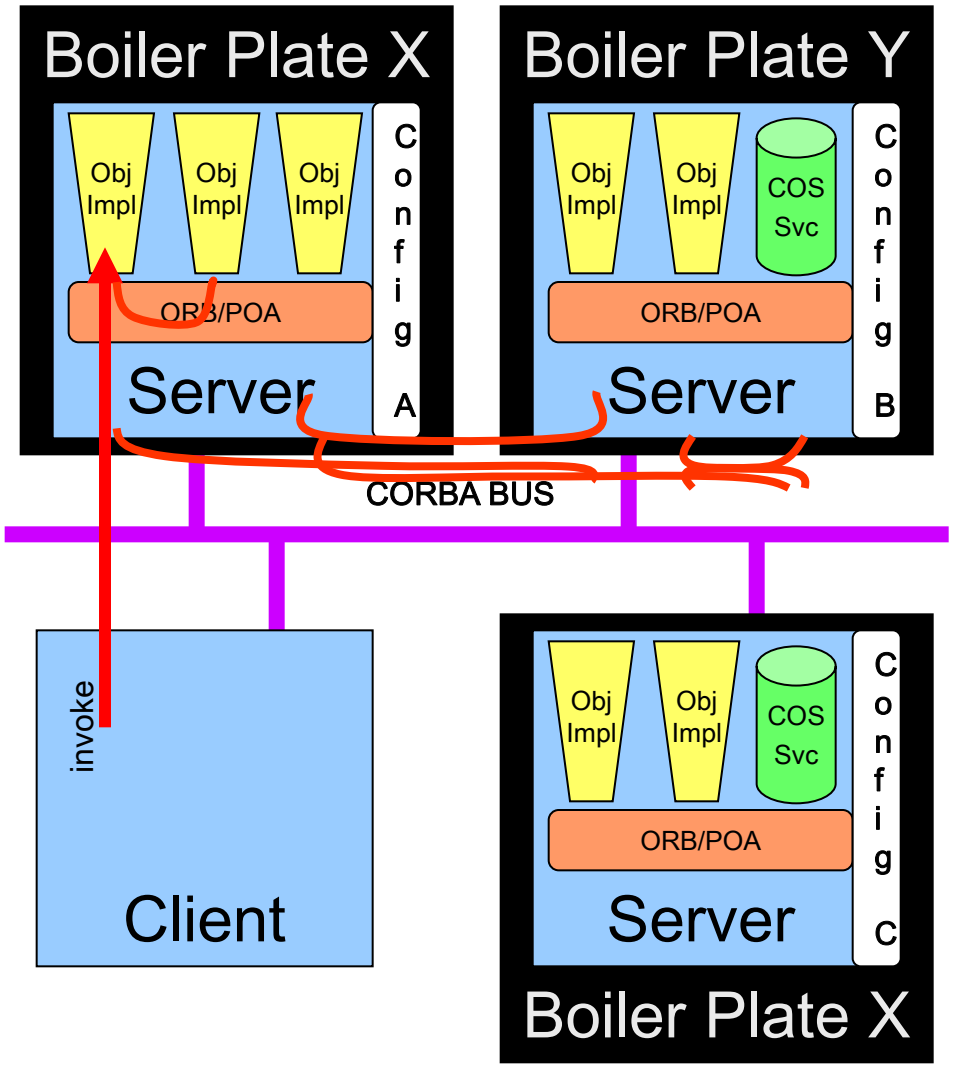
CORBA 2.x IDL non dà modo di raggruppare insieme ***interfacce correlate per offrire famiglie di servizi***

- Tale "bundling" deve essere fatto manualm. dagli sviluppatori via CORBA idiom&pattern

CORBA 2.x non specifica come fare ***configuration&deployment di oggetti*** per creare applicazioni complete

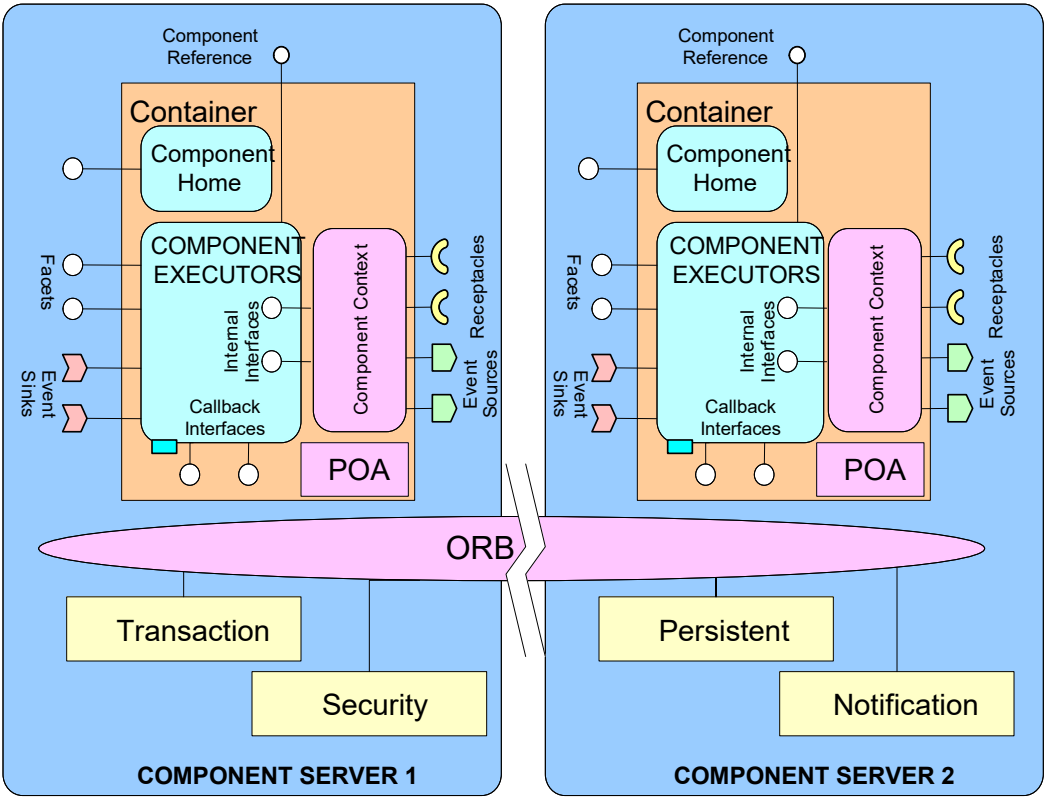
- Gli sviluppatori sono obbligati a preparare infrastrutture proprietarie e script ad hoc

Esempi di Limitazioni della Specifica CORBA 2.x



- Requisiti non-trivial di Distributed Runtime Environment (DRE):
  - Collaborazione e coordinamento di oggetti&servizi multipli su diverse piattaforme
- CORBA 2.x ha limiti come ***mancanza di standard*** per:
  - ***Server/node configuration***
  - ***Object/service config***
  - ***Application assembly***
  - ***Object/service deployment***
- Conseguenze:
  - Scarsa adattabilità e manutenibilità
  - Crescita time-to-market e costi integrazione

Funzionalità di CCM (1)



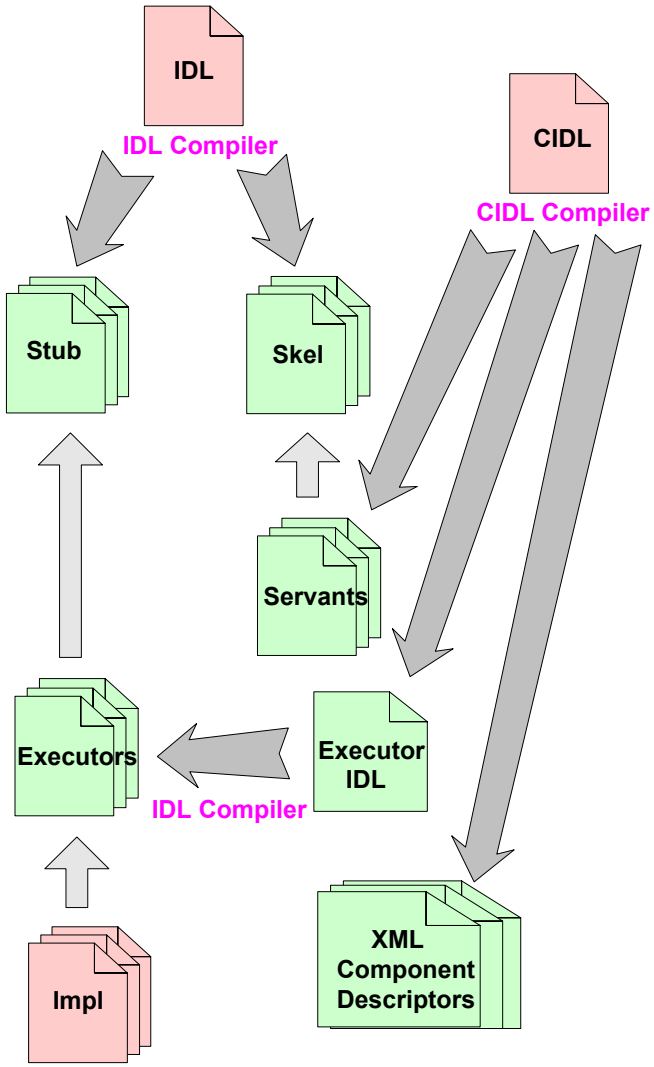
Container definiscono operazioni che abilitano i component executor ad accedere a ***servizi comuni di middleware & politiche runtime*** associate

Component Impl. Framework (CIF)

- Automatizza l'implementazione di molte funzionalità a livello componente

…

Funzionalità di CCM (2)



...

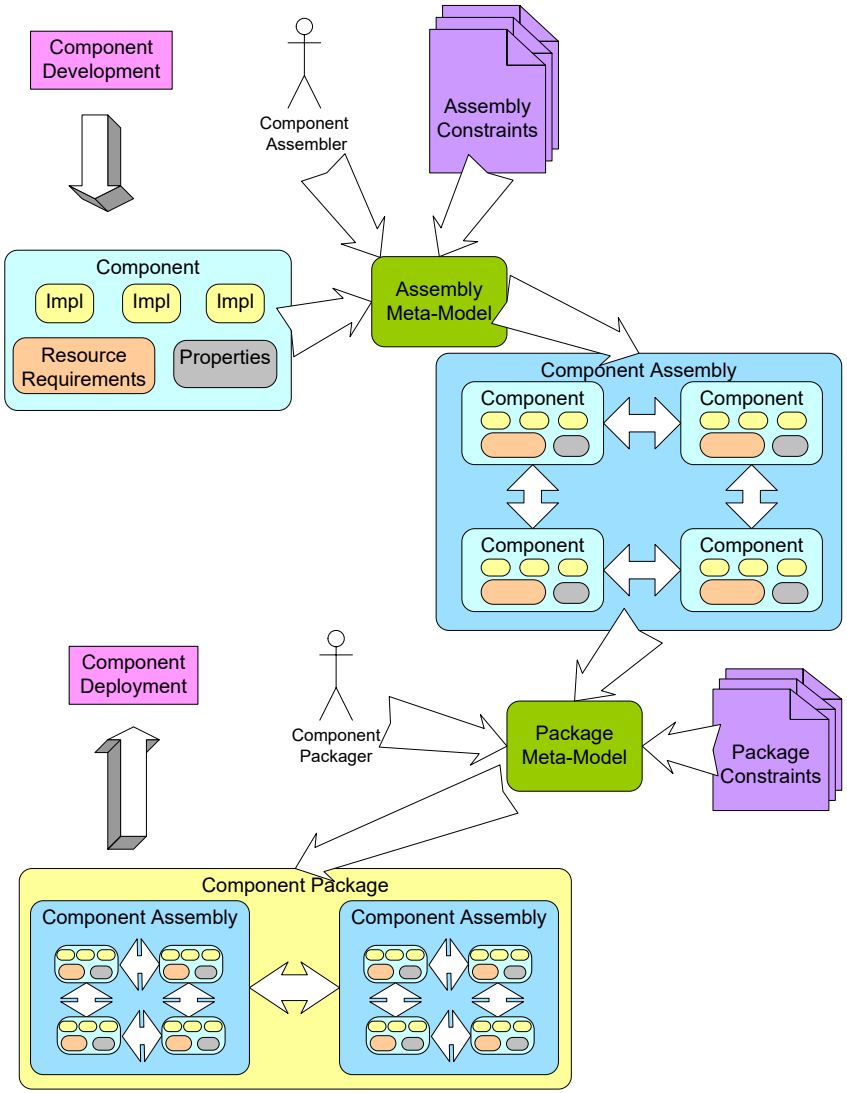
Component packaging

- Mette insieme metadati di implementation&configuration in assembly pronti per deployment

Strumenti per Component deployment

- Automatizzano il deployment di component assembly verso component server

Funzionalità di CCM (3)

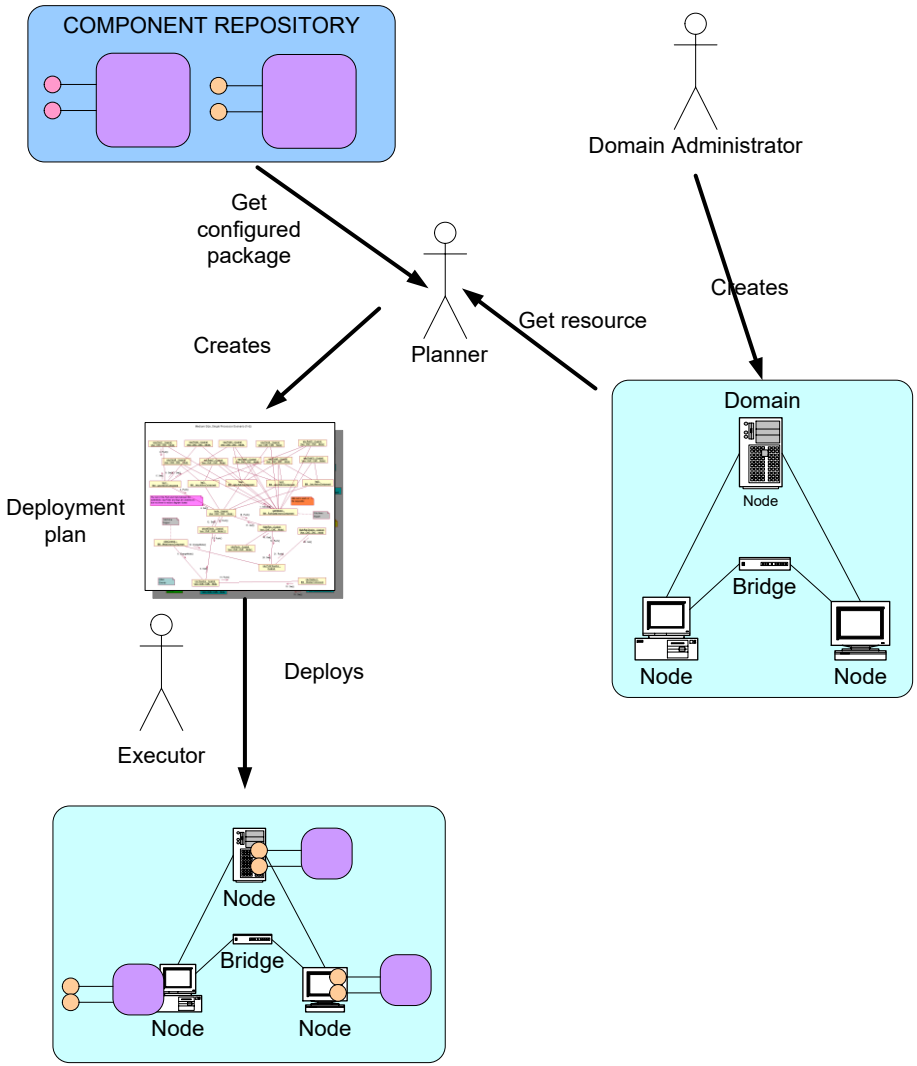


...

Strumenti per component deployment

- Automatizzano il deployment di component assembly verso component server

Funzionalità di CCM (4)



...

Component packaging

- Mette insieme metadati di implementation&configuration in assembly pronti per deployment

Implementazioni CCM Disponibili (un po' datate…)

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/

Comparazione CCM vs. EJB, COM, .NET

Come Enterprise Java Beans (EJB):

- Componenti CORBA creati&gestiti da home
- Eseguono in container che gestiscono trasparentem. servizi di sistema
- Ospitati da application component server generici
- MA possono essere realizzati in diversi linguaggi di implem.

Come Microsoft Component Object Model (COM):

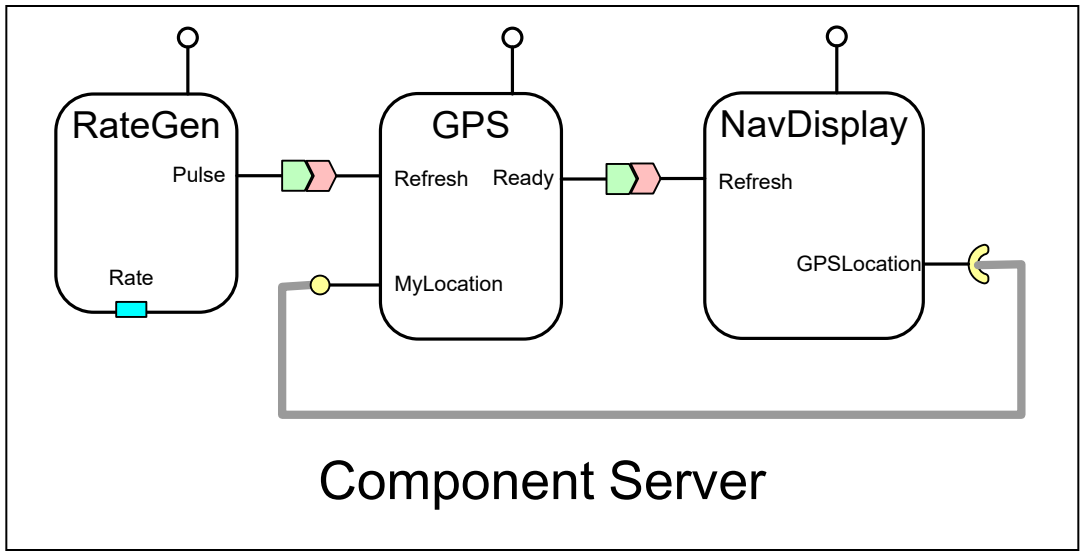
- Possono avere diverse interfacce input&output per ogni componente
- Sia operazioni point-to-point sync/async che eventi publish/subscribe
- Capacità di component navigation & introspection
- MA supporto più efficace e flessibile a proprietà di distribuzione e QoS

Come Microsoft .NET Framework:

- Possono essere realizzati in diversi linguaggi programmazione
- Possono essere packaged per facilitare distribuzione
- MA possono eseguire su piattaforme multiple (non solo Microsoft Windows)

Running Example

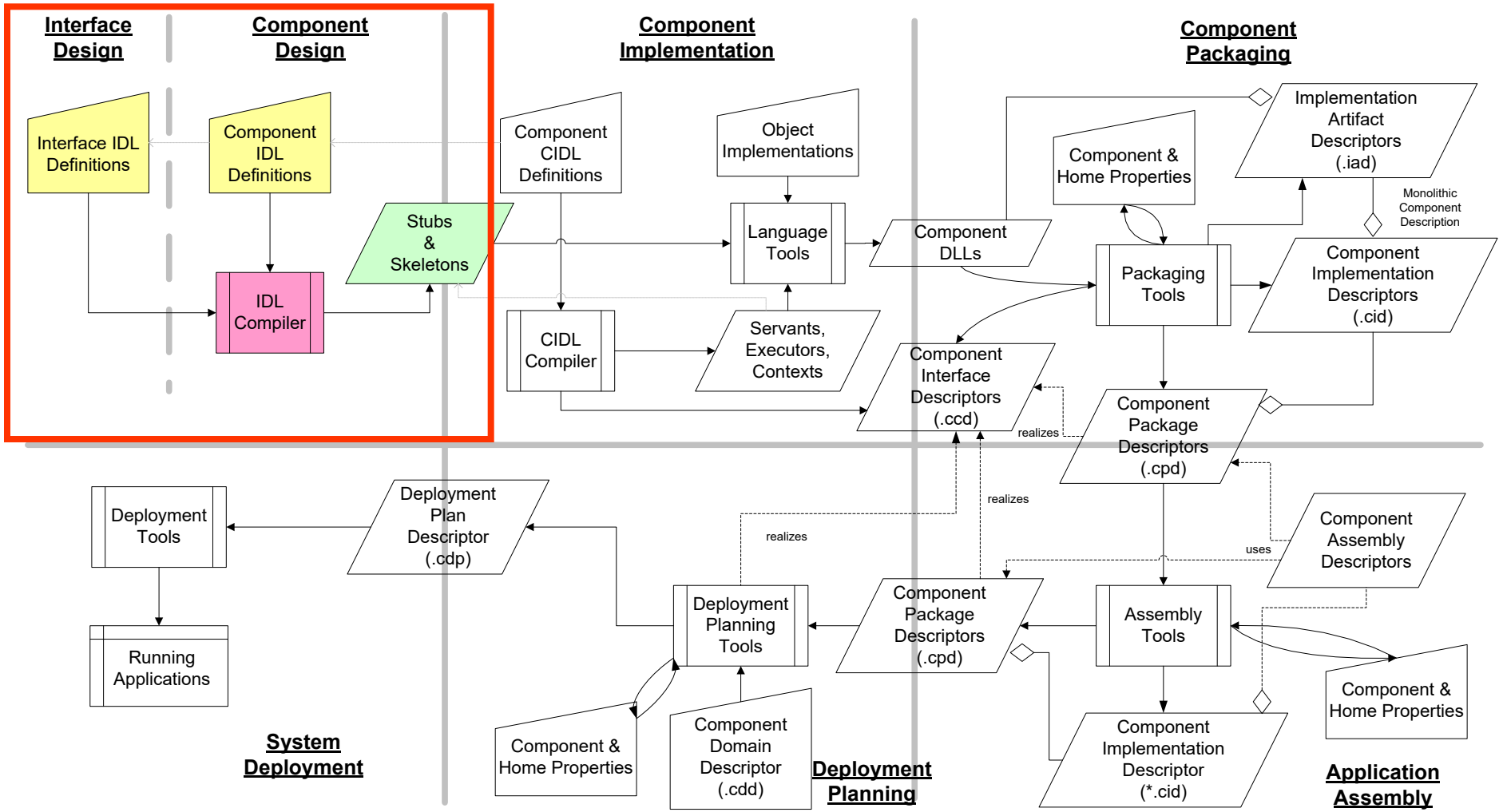
Rate Generator | Positioning Sensor | Display Device



`$CIAO_ROOT/examples/OEP/Display/`

- ***Rate Generator***
  - Invia eventi **Pulse** periodici ai consumatori
- ***Positioning Sensor (GPS)***
  - Riceve eventi di **Refresh** dai pubs
  - Rinfresca le coordinate cached disponibili via **MyLocation** facet
  - Notifica subs via eventi **Ready** events
- ***Display Device (NavDisplay)***
  - Riceve eventi **Refresh** da pubs
  - Legge coordinate correnti via **GPSLocation** receptacle
  - Aggiorna display

Primo Passo: Interface&Component Design



Semplice Esempio di Componente CCM

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


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

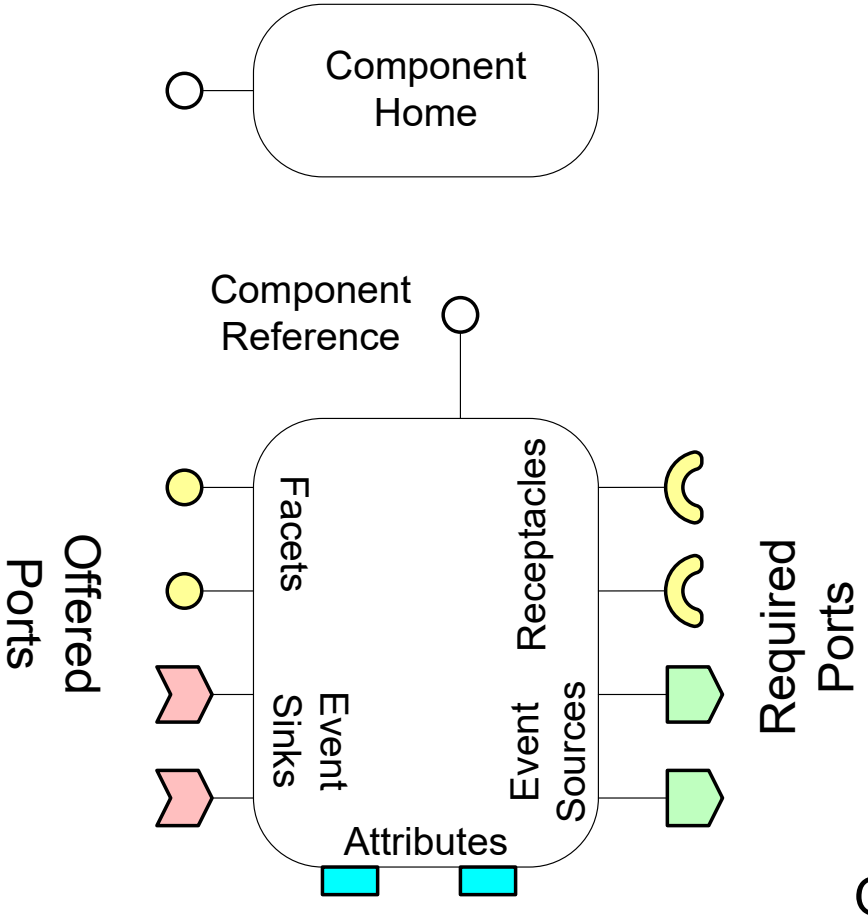
- Ruoli di un componente CCM
  - ***Definire una unità di composizione, riutilizzo e implementazione***
  - ***Incapsulare un modello di interazione e configurazione***
- Un componente CCM ha diverse opzioni possibili
  - Può ***ereditare*** da un ***singolo component type***

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

  - Può ***supportare interfacce IDL multiple***

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

Porte CCM



Un componente CCM può specificare ***porte*** (ovviamente diverse dalle porte di OSI layer4…):

- ***Facet*** (relazione **provides**)
  - Offrono interfacce per operazioni
- ***Receptacle*** (**uses**)
  - Interfacce richieste, da supportare
- ***Sorgenti per eventi*** (**publishes** & **emits**)
  - Eventi prodotti
- ***Sink per eventi*** (**consumes**)
  - Eventi consumati
- ***Attributi*** (**attribute**)
  - Proprietà configurabili

Ogni istanza di componente è creata e gestita da un singolo ***"componente di infrastruttura" home***

Gestione Lifecycle CCM

Componenti CCM devono essere creati dal runtime CCM

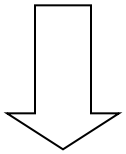
- Problemi con CORBA 2.x
  - ***No modo standard per gestire lifecycle*** degli oggetti
  - Bisogno di meccanismi standard e ***strategie gestione high-level***

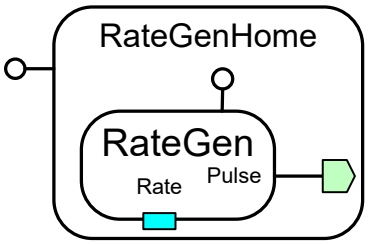
- Soluzione CCM
  - ***Integrare servizio lifecycle*** nella ***definizione (e configurazione)*** dei componenti CCM
  - Usare ***differenti home*** per componenti al fine di fornire diverse strategie di lifecycle management

    Soluzione basata su pattern Factory&Finder

CCM 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** come ***nuovo meta-type*** CORBA
  - **home** ha ***riferimenti a interfacce e oggetti***
- ***Gestisce una "famiglia" di componenti***
  - Diverse tipologie di home possono gestire la stessa famiglia di componenti
  - Una istanza di componente è gestita SOLO da una istanza di home
- Operazioni standard *factory* & *finder*
  - ad es. **create()**
- **home** può ospitare operazioni addizionali user-defined

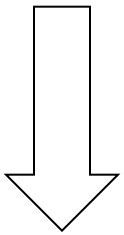
Componenti CCM Possono Mostrare "Viste" Differenti

Componenti possono avere bisogno di collaborare con altri tipi di componenti, che possono essere interessati a considerare ***differenti interfacce***

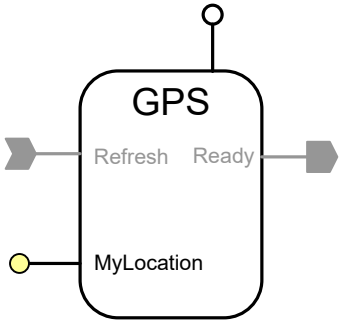
- Problemi con CORBA 2.x
  - Difficile estendere interfacce
  - ***No standard per acquisire nuove interfacce dinamicamente***
- Soluzione CCM
  - ***Definizione di facet***, ovvero interfacce esposte, che realizzano la ***vista di un componente*** e corrispondono al ***ruolo con cui un cliente può interagire con un componente***
    - "Top of the Lego"

CCM Facet

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



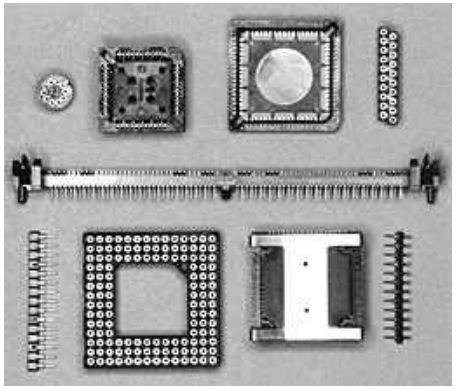
```
// Equivalent IDL 2
interface GPS
  : Components::CCMObject
{
  position
    provide_MyLocation ();
  …
};
```



Caratteristiche Facet:

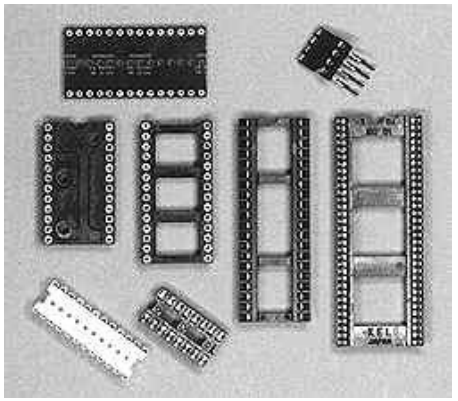
- Definiscono interfacce con operazioni offerte
- Specificate tramite keyword **provides**
- ***Rappresentano logicamente il componente*** stesso, non un'entità separata contenuta nel componente
- Facet hanno riferimenti a oggetti indipendenti ottenuti tramite operazione **provide_*()** da una factory
- Possono essere usati per implementare *Extension Interface* pattern

Utilizzo di Altri Componenti



Componenti hanno bisogno di collaborare con diversi tipi di componenti e applicazioni, caratterizzati da diversi tipi di interfaccia

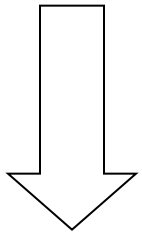
- Problemi con CORBA 2.x
  - No standard per specificare ***dipendenze fra interfacce***
  - No standard per ***connettere dinamicamente una interfaccia a un componente***
- Soluzione CCM
  - Definisce ***receptacle***, ovvero ***interfacce richieste***, che rappresentano punti diversi di named connection

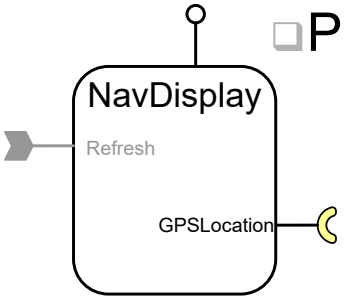


Rappresenta “bottom of the Lego”

CCM Receptacle

```
// 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 ();
  …
};
```

Caratteristiche Receptacle:

- Definiscono un modo per ***connettere una o più interfacce required*** a un componente
- Specificati tramite keyword **uses**
- Possono essere ***semplici o multipli***
  - Connessioni sono ***configurate staticamente*** durante fase di deployment
  - Connessioni sono ***gestite dinamicamente dai container*** che supportano interazione con clienti o altri componenti via callback
  - Inoltre CCM supporta ***connection establishment a runtime***

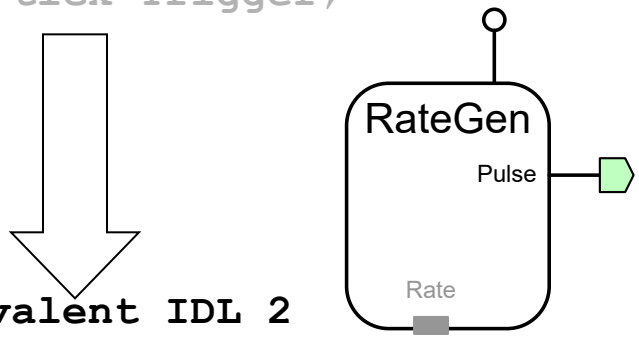
Distribuzione di Eventi

Componenti spesso necessitano di comunicare tramite ***meccanismi di message passing pub/sub***

- Problemi con CORBA 2.x
  - CORBA Event Service è dynamically typed, ovvero non c'è static type-checking fra entità pub/sub
  - Non banale estensione di interfacce request/response per supportare distribuzione di eventi
  - No standard per specificare la capacità di un oggetto di generare e processare eventi
- Soluzione CCM
  - Standardizzazione di **eventtype** & **eventtype *consumer interface*** (basata su **valuetypes**)
  - Standard per source/sink di eventi (push mode only)

Sorgenti di Eventi CCM

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



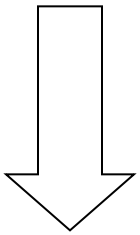

```
// Equivalent IDL 2
interface RateGen :
  Components::CCMObject {
  Components::Cookie
    subscribe_Pulse
    (in tickConsumer c);
  tickConsumer
    unsubscribe_Pulse
    (in Components::Cookie ck);
  …
};
```

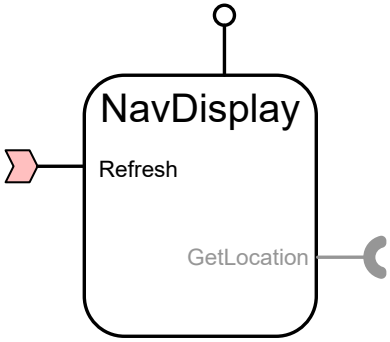
Caratteristiche sorgenti di eventi:

- Named connection per generazione eventi
- Due tipologie di sorgenti - ***publisher & emitter***
  - **publishes** = possono esserci ***consumatori multipli***
  - **emits** = ***un solo*** consumatore
- Due modi per connettersi a sink di eventi
  - Connessione diretta consumer
  - CCM container fa da intermediario verso canali CosNotification/ CosEvent o altro (ad es., OMG DDS)

Sink di Eventi CCM

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






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

Caratteristiche di sink di eventi:

- Named connection verso cui poter inviare eventi di tipologia specifica
- ***Event sink multipli dello stesso tipo possono essere subscriber della stessa sorgente***
- ***No distinzione fra emitter & publisher***
- Connessi a sorgenti via object reference ottenuta tramite operazione **get_consumer_*()** su factory

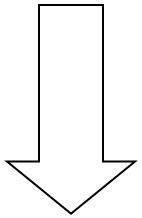
Sull'Utilità della Configurazione Dinamica di Componenti

Per rendere le implementazioni dei componenti più adattabili e flessibili, ***proprietà dei componenti dovrebbero essere (ri)configurabili***

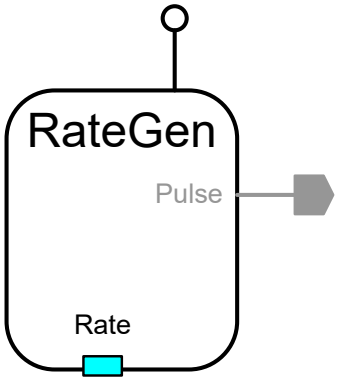
- Problemi
  - Applicazioni non dovrebbero legarsi a una specifica configurazione ***troppo presto***
  - No standard per specificare parametri configurabili per componenti in CORBA 2.x
  - Bisogno di meccanismi standard per configurazione
- Soluzione CCM
  - Configurare componenti ***via attributi di assembly/deployment, via home o nella fase di inizializzazione***

Attributi di Componenti CCM

```
// 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;
};
```



Caratteristiche degli attributi

- Proprietà configurabili con nome, mirate alla configurazione componenti
  - ad es. comportamenti opzionali, modalità, …
- Possono lanciare eccezioni user-defined (originale per CCM)
- Determinati tramite vari meccanismi di configurazione possibile
  - ad es. file XML descriptor generati da strumenti di modeling

Connettere Componenti CCM

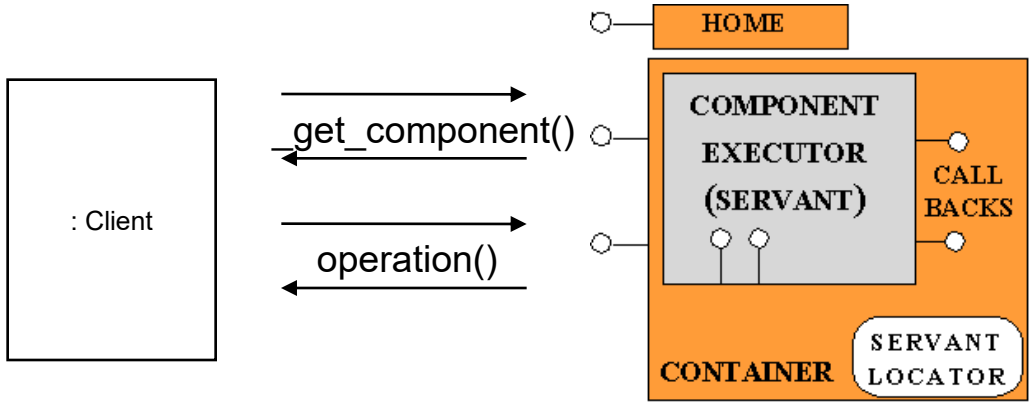
Componenti hanno bisogno di ***essere "aggregati"*** per formare applicazioni complete

- Problemi
  - Componenti possono avere porte multiple con diversi nomi e tipi
  - Dispendioso ***scrivere manualmente*** il codice necessario per ***collegare un insieme di componenti per una applicazione specifica***
- Soluzione CCM
  - ***Interfacce per introspezione*** per permettere scoperta dinamica di capacità componenti
  - ***Generiche operazioni sulle porte*** per connettere componenti tramite strumenti esterni di deployment&configuration

Rappresenta la capacità di mettere insieme i lego

Navigation&Introspection in CCM

- Capacità di navigation&introspection realizzate da **CCMObject**
  - Tramite interfaccia **Navigation** per facet, interfaccia **Receptacles** per receptacle e interfaccia **Events** per porte per eventi
- ***Navigation dal riferimento base di un componente a ogni sua facet via operazioni facet-specific automaticamente generate e supportate***
  - ad es. **Components::CCMObject::get_all_facets()** & **Components::CCMObject::provide()**
- Navigation da ogni facet al riferimento base di un componente tramite **CORBA::Object::_get_component()**

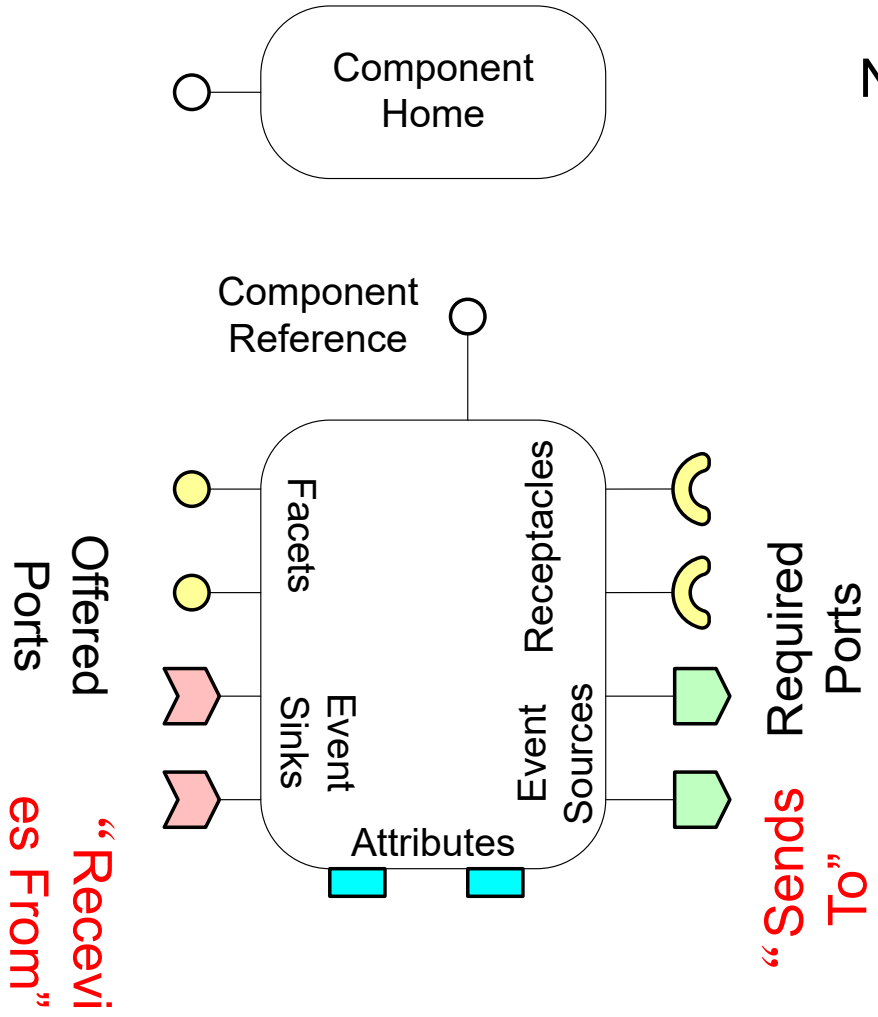


Tutto il codice per navigation&introspection è ***generato automaticamente dal compilatore CIDL*** come component servant
(lo vedremo brevem…)

Uso delle Interfacce di Navigazione di un Componente CCM

```
 1 int
 2 main (int argc, char *argv[])
 3 {
 4   CORBA::ORB_var orb =
 5     CORBA::ORB_init (argc, argv);
 6-10  // Get the NameService reference…
  CORBA::Object_var o = ns->resolve_str ("myHelloHome");
  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.
  ...
24   return 0;
25 }
```

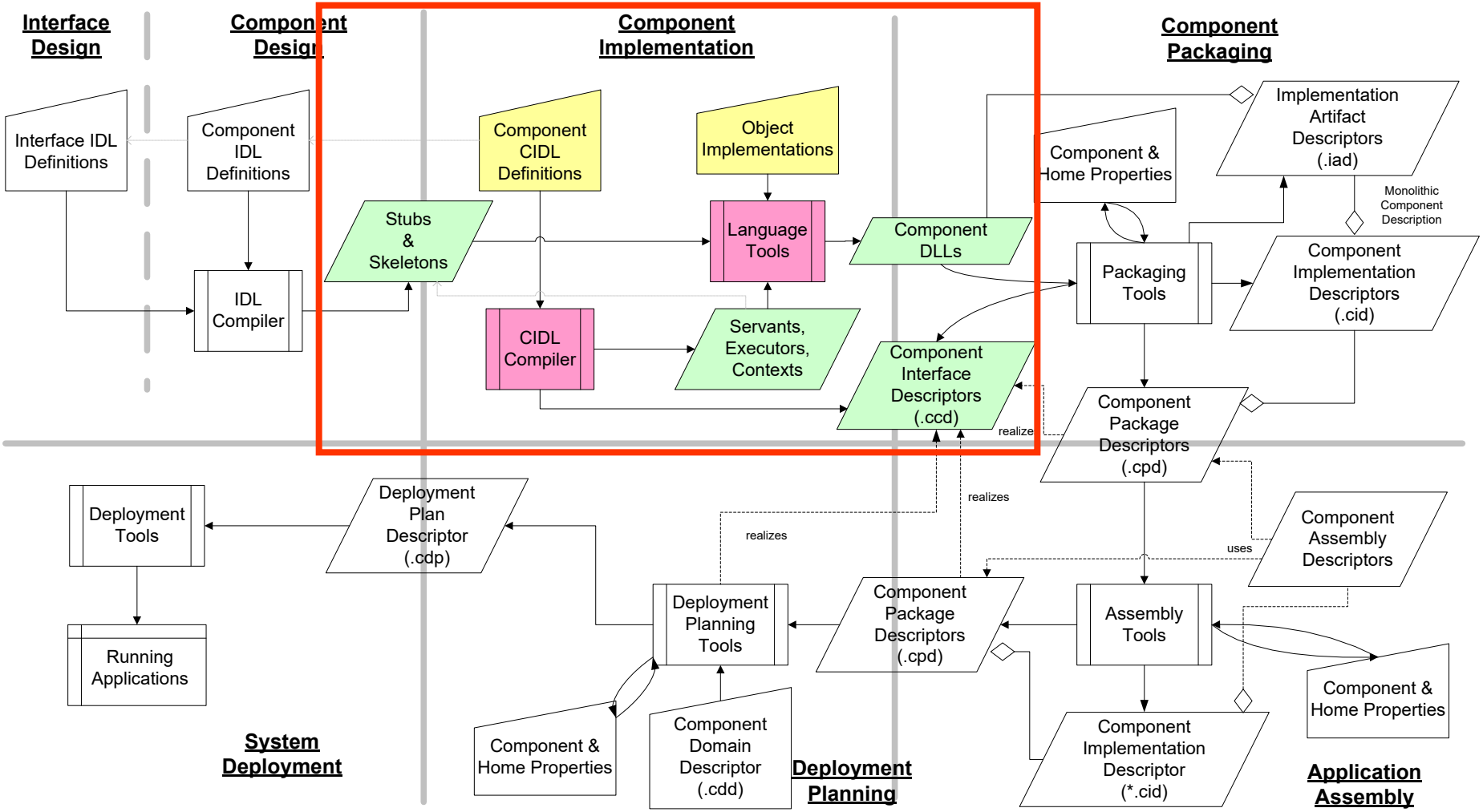
Recap: Funzionalità Componenti CCM



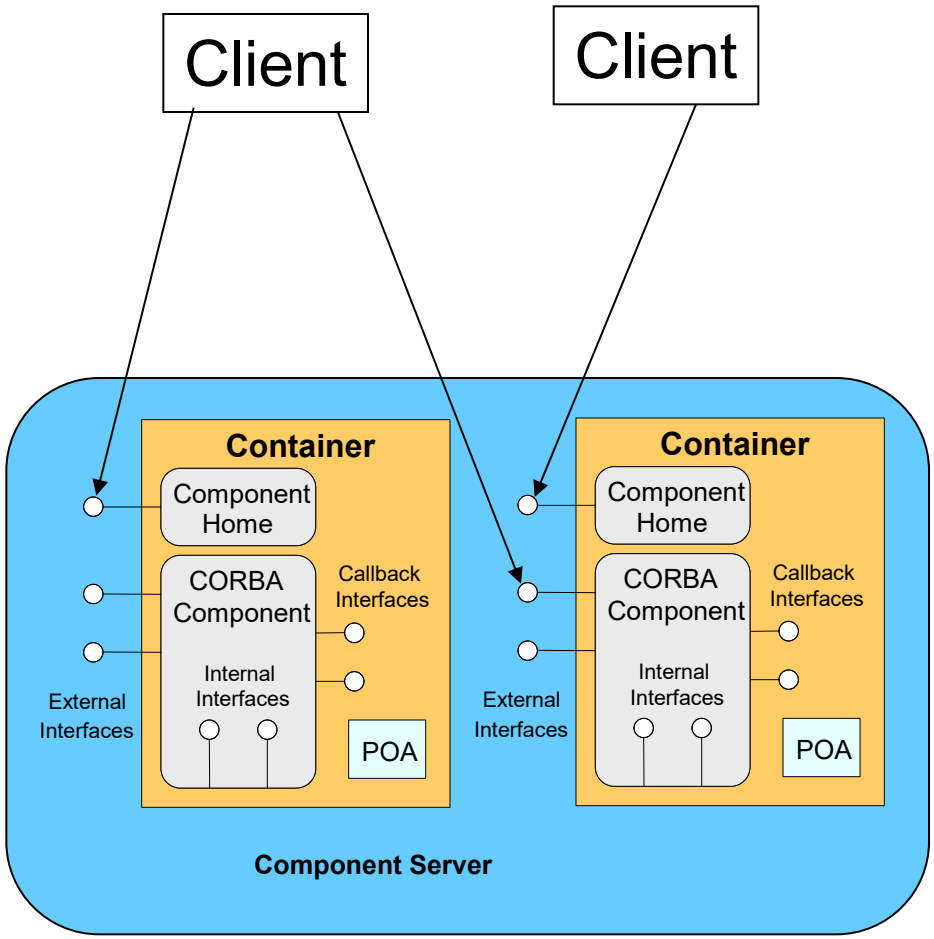
Nella prospettiva cliente il supporto CCM:

- Definisce operazioni per life cycle management (**home**)
- Definisce che cosa un componente **offre** agli altri componenti
- Definisce che cosa un componente **richiede** da altri componenti
- Definisce quali modalità di collaborazione sono usate fra componenti
  - ***Point-to-point via operation invocation***
  - ***Publish/subscribe via event notification***
- Definisce quali **attributi** dei componenti sono configurabili

Passo di Implementazione Componenti CCM



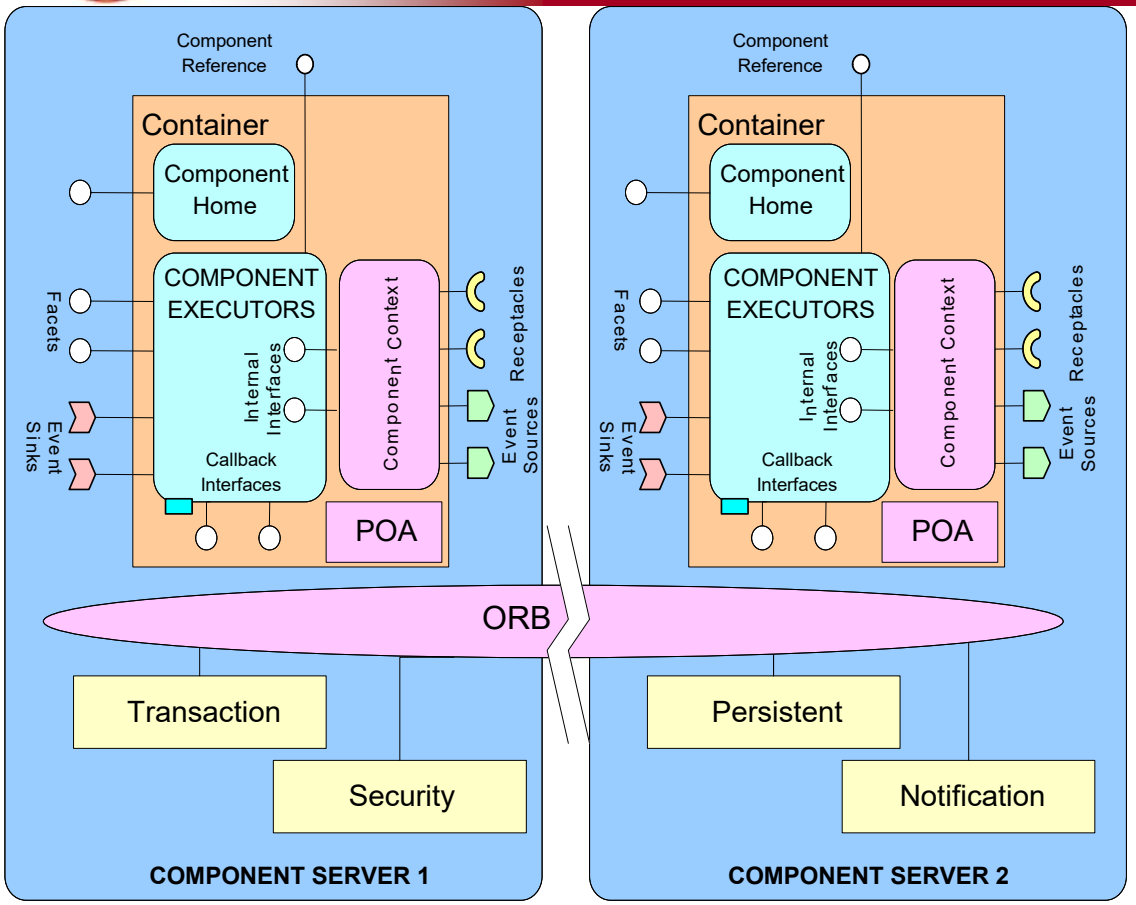
Supporto Runtime: Funzionalità Component Server



Principale step evolutivo di CCM rispetto a CORBA 2.x è il focus su component server e deployment/configurazione di applicazioni

- CCM estende CORBA 2.x tramite
  - Astrazioni high-level di ***modelli servant di frequente utilizzo***
  - Tool-based ***configuration & tecniche di meta-programming***
- Il ***container framework CCM*** gioca ruolo centrale in questo supporto

CCM Container Framework



Usa callback per gestire istanze di componenti

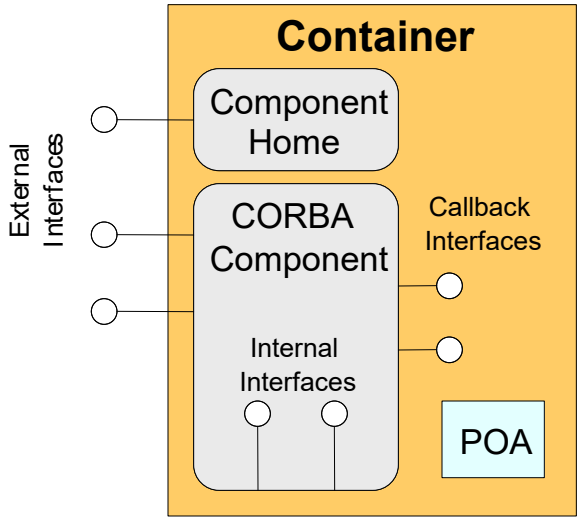
ad es. loro stato di sessione, activation/deactivation, etc.

Categorie di Component/Container in CCM

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

In CCM queste categorie possono essere specificate ***dichiarativamente tramite CIDL file*** piuttosto che programmate imperativamente

Container e Interfacce Esterne/Interne



- **External API** come interfacce verso clienti
- **Container API** come interfacce interne e interfacce di callback usate dagli sviluppatori di componenti nella logica applicativa

Interfacce interne usate dai componenti per accedere container facility

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

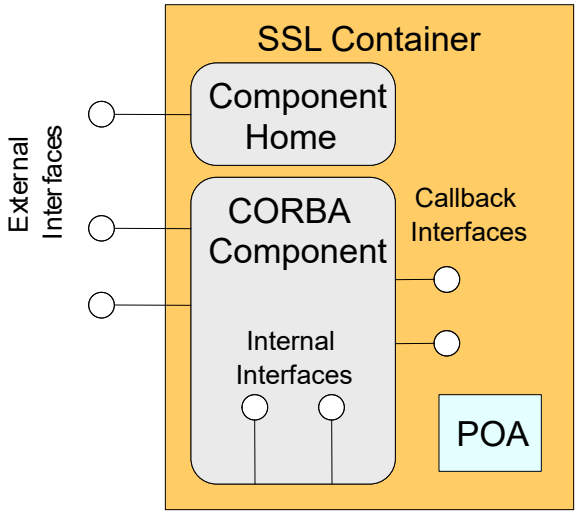
Interfacce di callback usate dai container per invocare il component executor

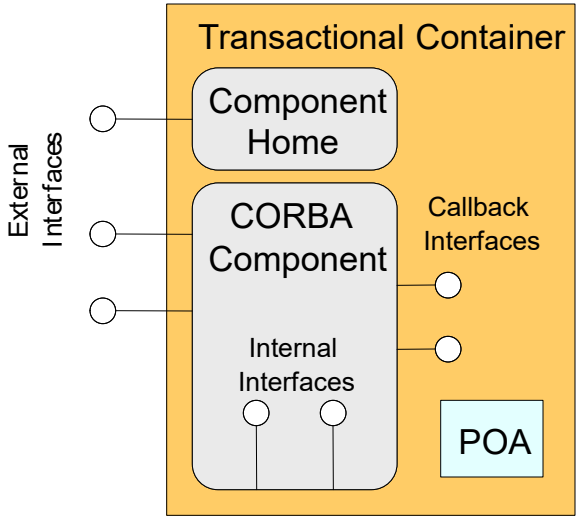
```
local interface EnterpriseComponent{};
local interface SessionComponent :
   EnterpriseComponent {
     void set_session_context
       (in SessionContext ctx)
     void ccm_activate ();
     void ccm_passivate ();
     void ccm_remove ();
 };
```

Strategie Container-managed

Obiettivo: ***disaccoppiare politiche di configurazione (installation/deployment time) dalla implementazione*** dei componenti

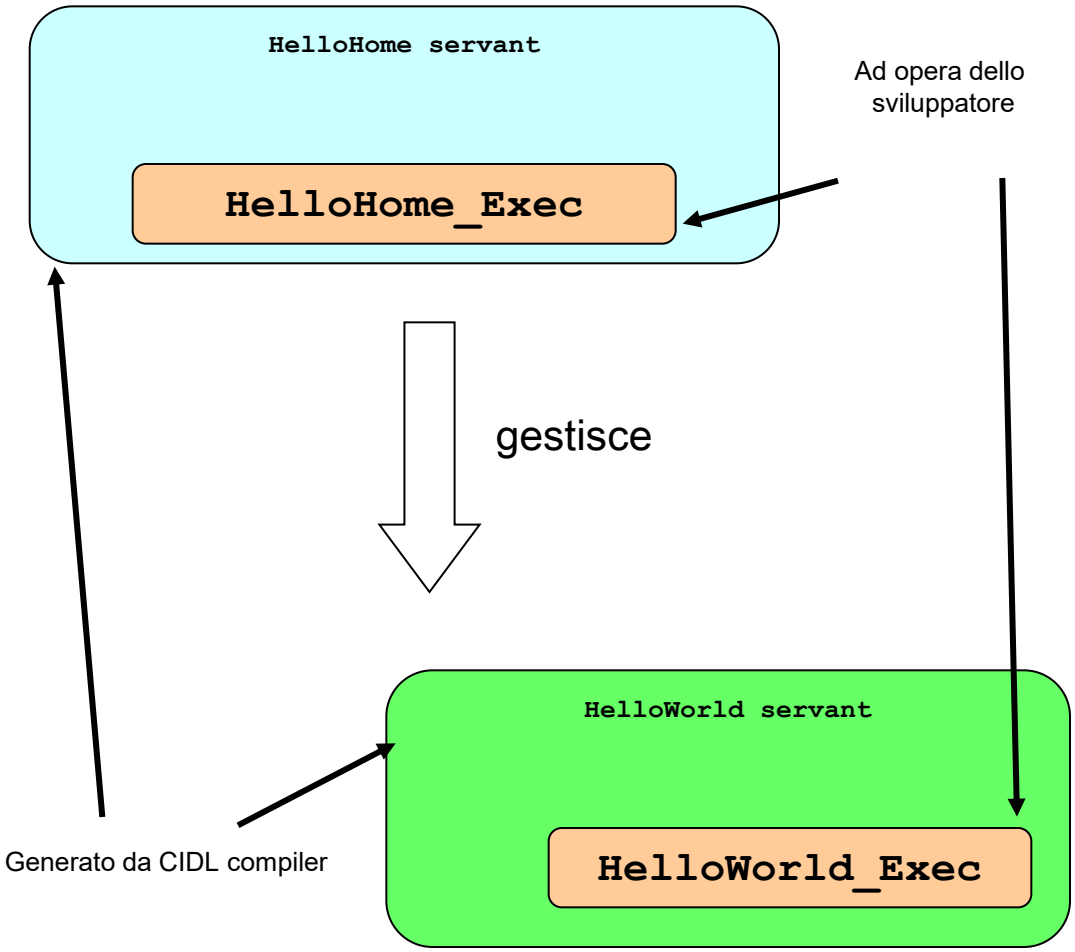
- Dichiarazioni definite per:
  - Servant lifetime
  - Transazioni
  - Eventi
  - Persistenza
  - Sicurezza
- Specificate da component/composition developer attraverso ***XML metadata e/o direttive CIDL***
- ***Implementate dal container, NON dai componenti***





Executor per Componenti e per Home

- Moduli server-side che implementano concetti di ***component (business logic) e home***
  - Oggetti CORBA locali con interfacce rispondenti a OMG IDL mapping
- ***Component executor*** possono essere
  - ***monolitici***, se tutte le porte sono implementate da singola classe
  - ***segmentati***, se porte suddivise su classi multiple
- ***Home executor sempre monolitici***



Executor Eseguono dentro il Container

- Container intercettano invocazioni su executor e gestiscono activation, sicurezza, transazioni, persistenza, … ***CONTAINER PESANTE***
- Executor dei componenti devono implementare ***interfaccia locale per callback lifecycle ad uso del container***
  - ***SessionComponent*** per componenti transienti
  - ***EntityComponent*** per componenti persistenti
- Executor dei componenti possono interagire con container e componenti connessi via ***context interface***