



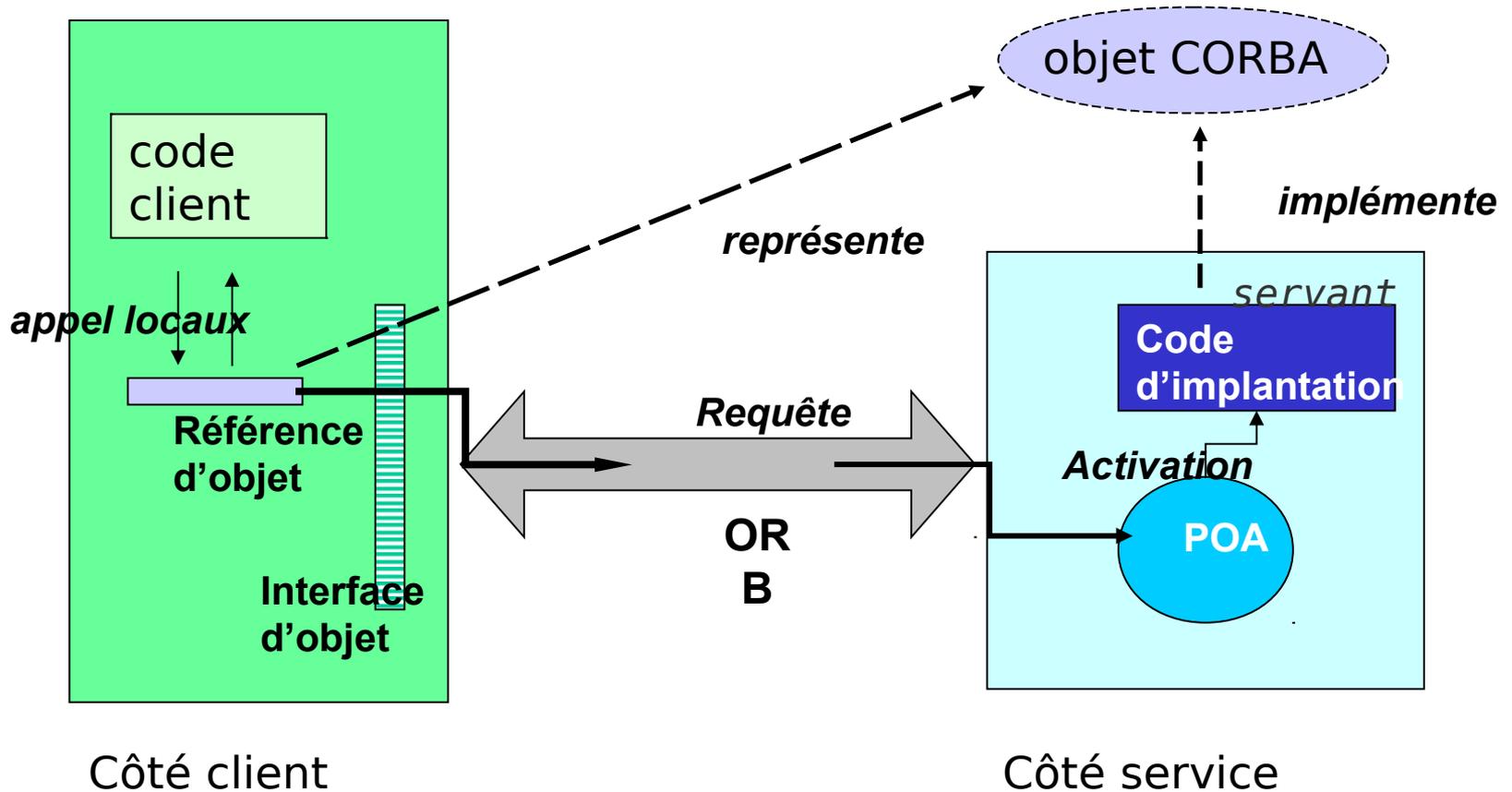
CORBA Avancé

Alexandre Denis – Alexandre.Denis@inria.fr

**Inria Bordeaux – Sud-Ouest
France**

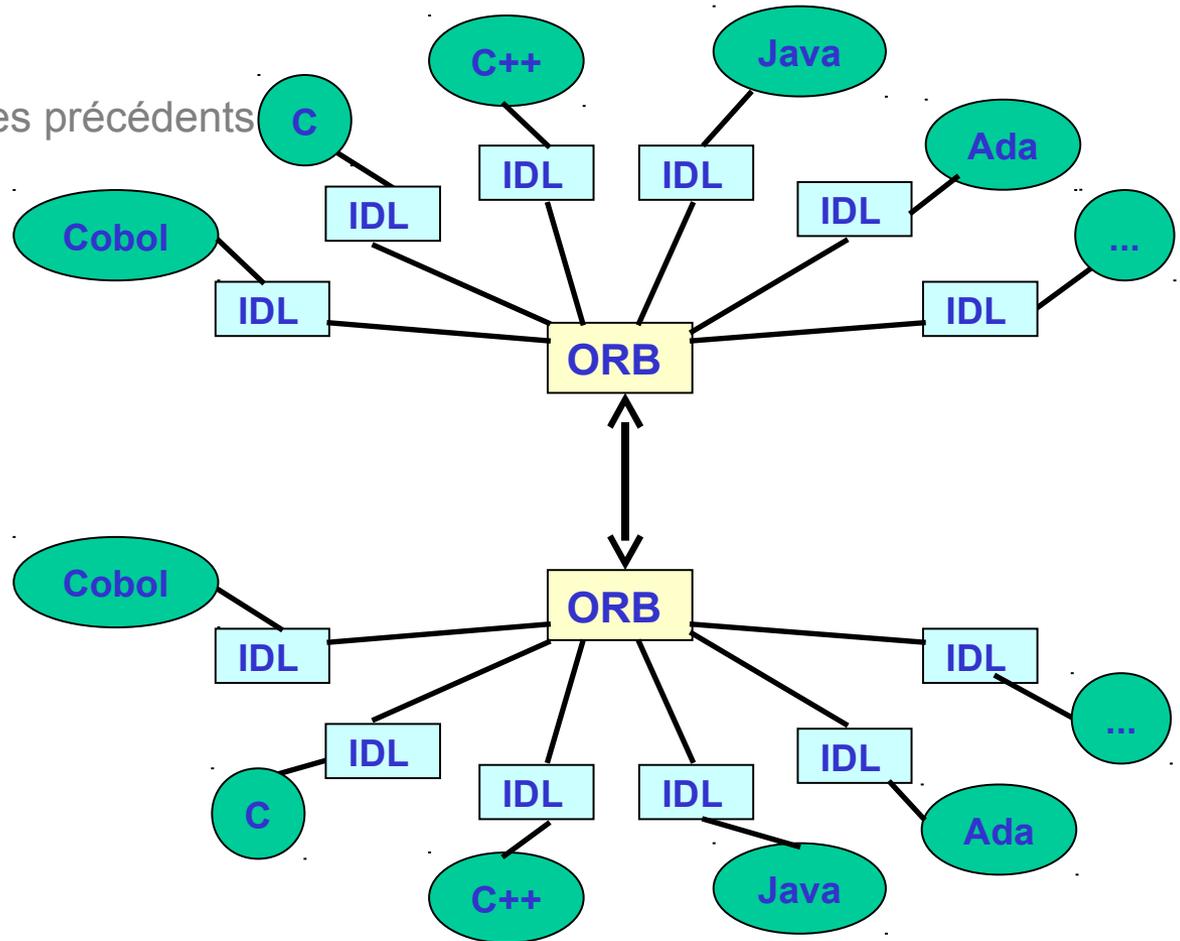
Résumé CORBA

- Dans les épisodes précédents...



Résumé CORBA

- Dans les épisodes précédents



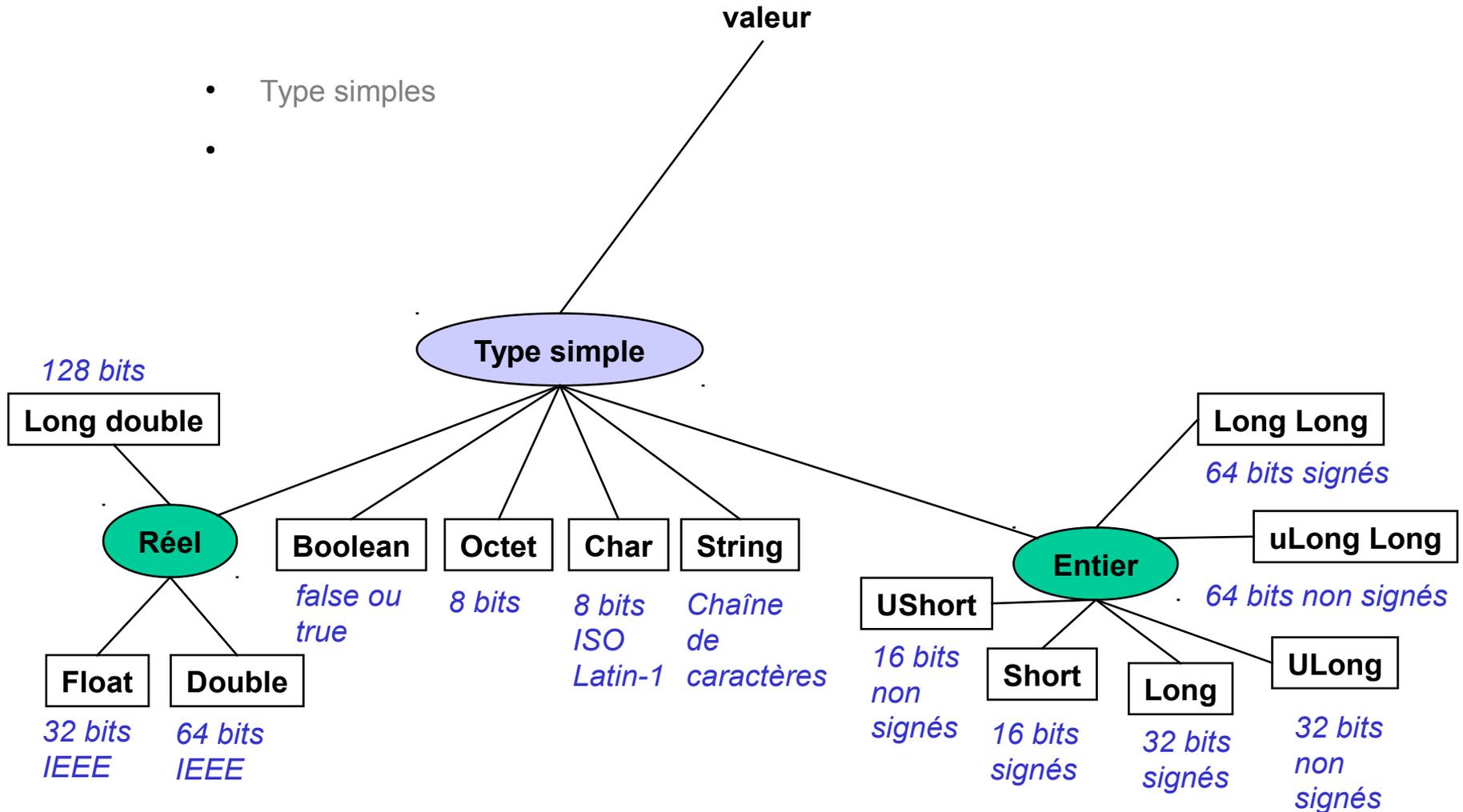
Au programme aujourd'hui

- Types IDL avancés
 - Sequence, Any, DynAny, TypeCode, ValueType
- Interface dynamique
- Pointeur intelligent *_var
- POA avancé
 - Politiques du POA, ServantLocator, ServantActivator
- Intercepteurs
- Composants CCM

Fonctions avancées CORBA 2.x

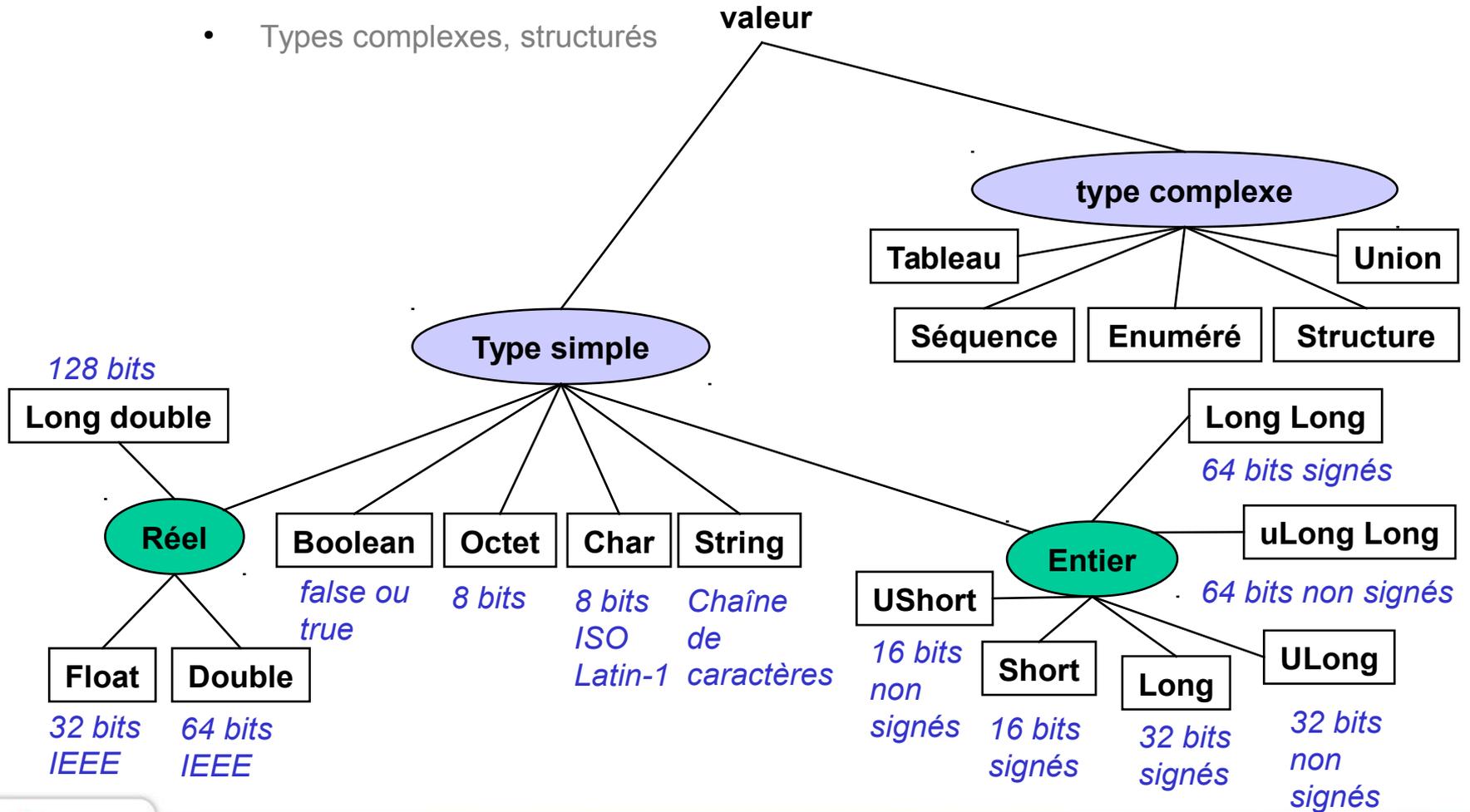
Types IDL

- Type simples
-



Types IDL

- Types complexes, structurés



Séquences

- Tableau
 - Taille fixe uniquement (donc très peu utilisé!)
 - Définition obligatoire d'un type !

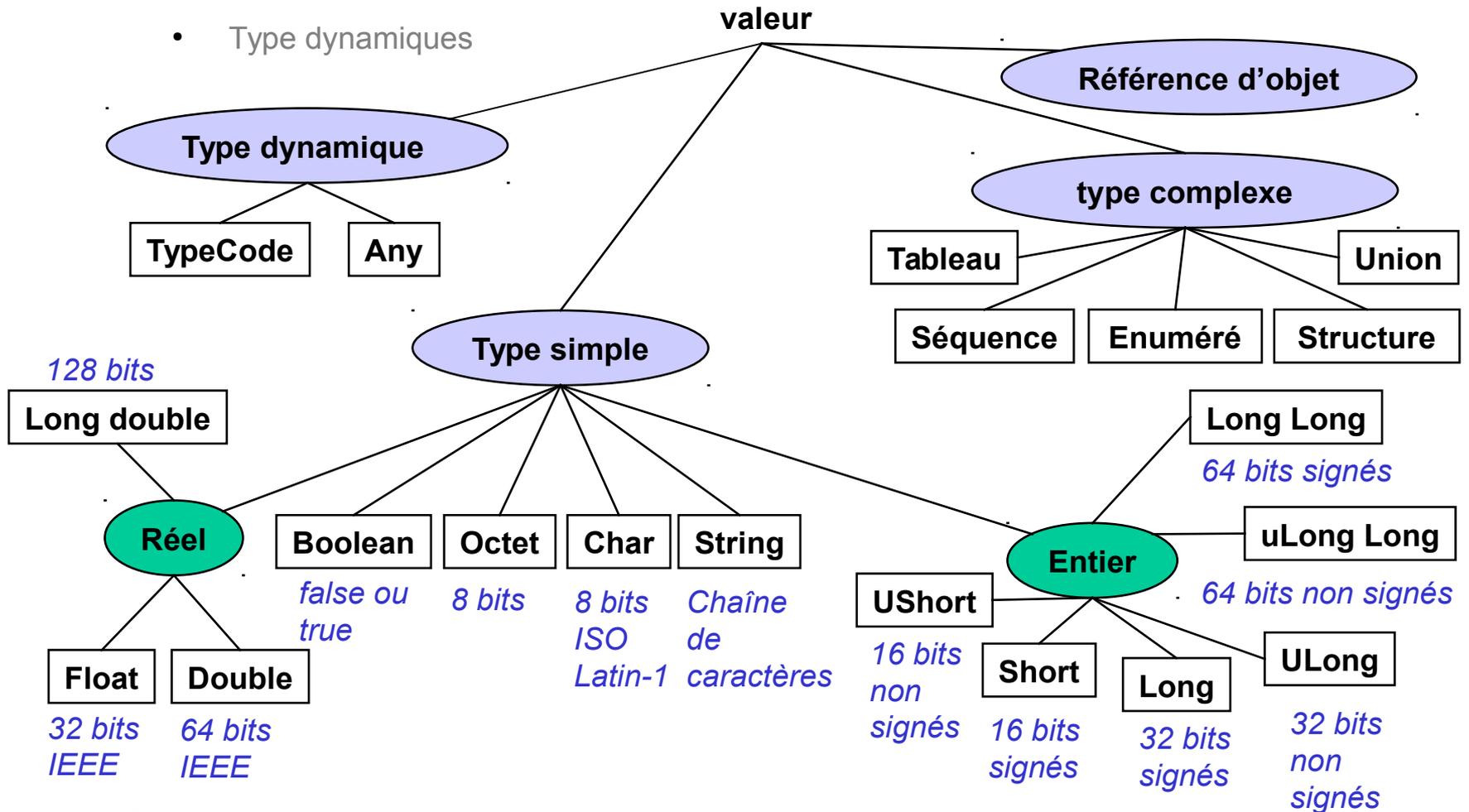
```
const short TAILLE = 10;  
typedef float[20][TAILLE] Matrice;
```

- Sequence
 - Vecteur de taille quelconque
 - non borné
 - borné

```
typedef sequence<string> StrSeq // séquence de chaîne  
typedef sequence<short> entiers; // séquence non bornée  
typedef sequence<long, 100> Nombres; // au plus 100 nombres  
typedef sequence<Nombres> ListDeNombres; // séquence de séquence
```

Types IDL

- Type dynamiques



Types dynamiques

- TypeCode
 - Structure décrivant un type de données (nom + champs)
- **any**
 - Contient
 - le type de la donnée (CORBA TypeCode)
 - la valeur de la donnée
 - Très utile pour les opérations polymorphes
 - Permet un nombre variable de paramètres
- dynany
 - Type *dynamique* pour manipuler localement un any avec itérateurs

```
interface ValueStore
{
    void put(in string name, in any value);
    any get(in string name);
};
```

Types valuetype

- Types qui sont des objets :
 - CORBA::Object – objet accessible à distance
 - champs privés, attachés à leur serveur (passés par référence)
 - IDL struct – données sérialisables
 - Passés par valeur, données uniquement (pas de méthodes)
 - Valuetype – objet **local sérialisable**
 - Passés par valeur
 - Méthodes avec invocation locale uniquement
 - Pas de garantie que tous les noeuds utilisent la même implémentation des méthodes...

```
valuetype Date {  
    short    year;  
    short    month;  
    short    date;  
    void     next_day();  
    void     previous_day();  
};
```

Types C++ *_var – Gestion de la mémoire

- Type T_var
 - Généré à partir de la description IDL
 - Gère l'allocation et la désallocation (~ ramasse miette)
- Type T_ptr
 - Type pointeur vers une donnée de type T
 - Généré à partir de la description IDL
-

```
IDL:    interface A { ...};
```

```
C++:    typedef ... A_var;  
        typedef ....A_ptr;
```

Interface de l'ORB

- L'ORB vu comme un objet CORBA
 - Uniquement local !

```
#pragma prefix "omg.org"

module CORBA {
  ...
  interface ORB {
    ...
    string object_to_string ( in Object obj );
    Object string_to_object ( in string str );

    typedef string ObjectId;
    Object resolve_initial_references ( in ObjectId id)
        raises (InvalidName);

    void run();
    boolean work_pending();
    void perform_work();
  };
};
```

Interface dynamique

- Utiliser CORBA sans stub/squelette
 - Sérialiser/desérialiser des requêtes explicitement
- Utile pour :
 - Langages avec typage dynamique, typage faible
 - ex.: python, perl, Tcl, LISP, Visual Basic, etc.
 - Debug
 - Prototyper rapidement un client « script » pour tests unitaires
 - Interfaces IDL évolutives
 - S'adapter à la volée au type
 - Passerelles, proxy
 - Contrôler manuellement l'envoi des requêtes
 - Asynchronisme, envoi groupé, etc.

DII - Interface client dynamique

- DII – Dynamic Invocation Interface
 - Usage assez courant
- Manipulation explicite de `CORBA::Request`
 - Création : `CORBA::Object::_create_request()`
 - Empaquetage explicite des arguments dans des `Any`
 - Contiennent des `NamedValue` – paire nom, valeur
 - Invocation : `CORBA::Request::invoke()`
 - Récupération explicite du résultat :
`CORBA::Request::get_response()`

DII - Exemple

- IDL de l'exemple : `short anOpn(in string a);`
- Exemple DII détaillé

```
CORBA::NVList_var args;  
orb->create_list(1, args);  
*(args->add(CORBA::ARG_IN)->value()) <<= (const char*) "Hello World!";  
  
CORBA::NamedValue_var result;  
orb->create_named_value(result);  
result->value()->replace(CORBA::_tc_short, 0);  
  
CORBA::Request_var req = obj->_create_request(CORBA::Context::_nil(),  
                                              "anOpn", args, result, 0);
```

- Même exemple, syntaxe idiomatique :

```
CORBA::Request_var req = obj->_request("anOpn");  
req->add_in_arg() <<= (const char*) "Hello World!";  
req->set_return_type(CORBA::_tc_short);
```

DSI - Interface serveur dynamique

- DSI – Dynamic Skeleton Interface
 - Usage plus anecdotique que DII
- Servant générique déclaré au POA : `PortableServer::DynamicImplementation`
 - Hériter de la classe générique
 - Surcharger la fonction `invoke(CORBA::ServerRequest)` qui reçoit tous les appels
 - Dépaqueter le contenu de `ServerRequest`, traiter la requête, empaqueter le résultat, de manière symétrique à la gestion des DII
 - Servant enregistré dans le POA comme n'importe quel servant
 - Le plus souvent via un `ServantManager` pour gérer plusieurs objets

DSI - Exemple

```
void MyDynImpl::invoke(CORBA::ServerRequest_ptr request)
{
    try {
        if( strcmp(request->operation(), "echoString") )
            throw CORBA::BAD_OPERATION(0, CORBA::COMPLETED_NO);
        CORBA::NVList_ptr args;
        orb->create_list(0, args);
        CORBA::Any a;
        a.replace(CORBA::_tc_string, 0);
        args->add_value("", a, CORBA::ARG_IN);
        request->arguments(args);
        const char* mesg;
        *(args->item(0)->value()) >= mesg;
        CORBA::Any* result = new CORBA::Any();
        *result <= CORBA::Any::from_string(mesg, 0);
        request->set_result(*result);
    }
    catch(CORBA::SystemException& ex){
        CORBA::Any a;
        a <= ex;
        request->set_exception(a);
    }
    catch(...){
        cout << "echo_dsiimpl: MyDynImpl::invoke - caught an unknown exception."
        << endl;
        CORBA::Any a;
        a <= CORBA::UNKNOWN(0, CORBA::COMPLETED_NO);
        request->set_exception(a);
    }
}
```

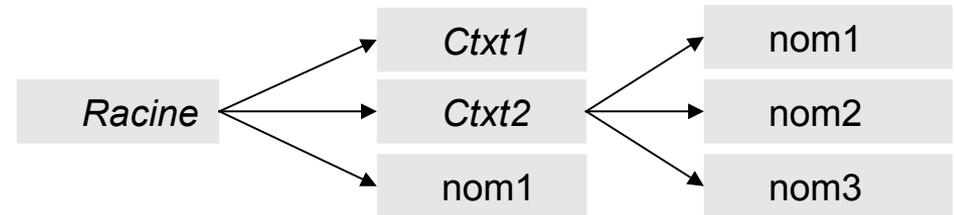
Asynchronisme (ou presque)

- Abus de langage – en fait, requête **non-bloquante**
- Construction du langage IDL : **oneway**
 - Invocation non-bloquante, pas de valeur de retour, pas d'exception
 - Aucune garantie de fiabilité !
- Utilisation de DII
- Construction selon un pattern *callback*
 - Le servant rend la main immédiatement, crée un thread pour le travail, appelle une fonction en retour sur le client pour donner le résultat
- AMI – Asynchronous Message Interface
 - Un pattern *callback* généré automatiquement par le compilateur IDL
 - Partie *optionnelle* de la norme CORBA, implémentation peu répandue

Asynchronisme, pour de vrai

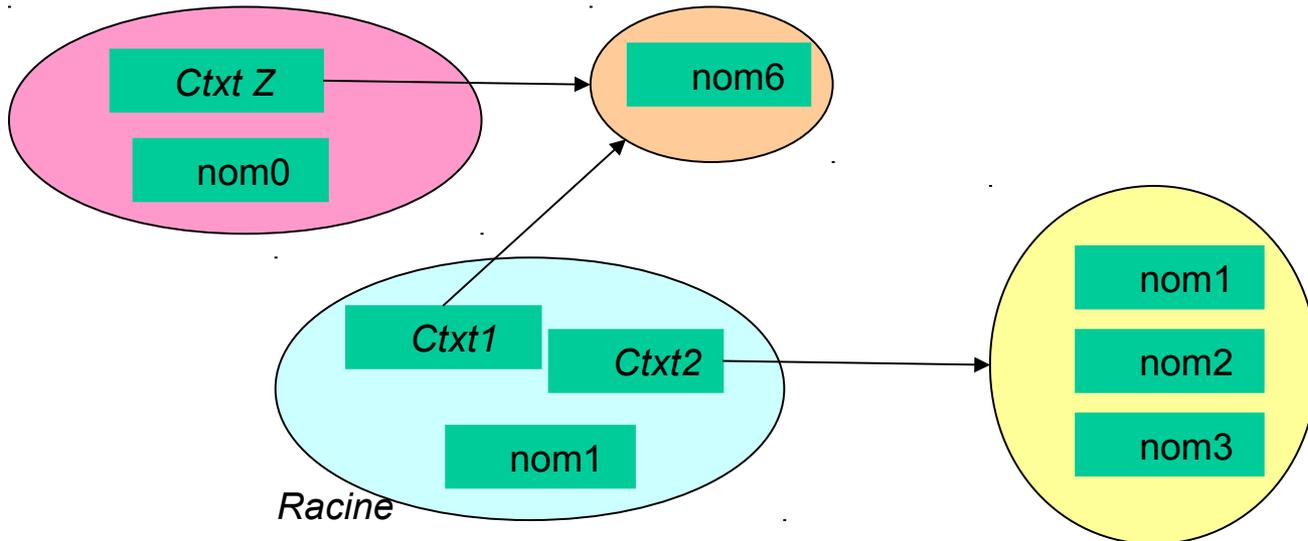
- Vrai asynchronisme : la requête n'est pas un point de synchronisation (**barrière**) entre client et serveur
 - Messages asynchrones, boîte aux lettres
- Avec une **tierce-partie** persistante
 - Pattern *Mediator*
 - Les messages transitent par la tierce-partie
 - Le récepteur lit explicitement sa boîte aux lettres (modèle *pull*) ou reçoit une notification quand il est connecté (modèle *push*)
- Avec des événements asynchrones
 - Pattern *Publish/subscribe*
 - Implémenté dans les services CosEvent et CosNotification

NameService avancé



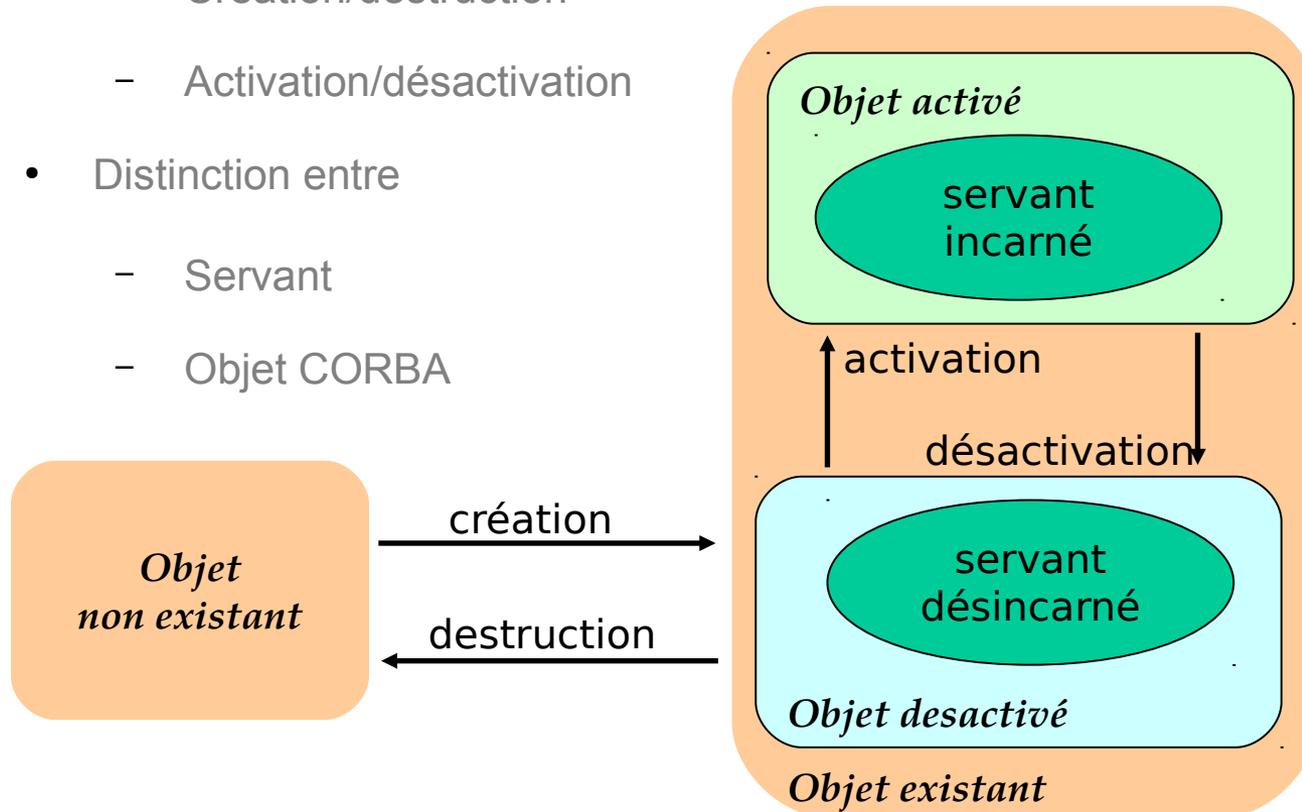
- Fédération de name service
 - Un NamingContext est un objet CORBA comme un autre
 - Les références peuvent être externes

Autre Racine



Cycle de vie d'un objet CORBA

- Notions orthogonales
 - Création/destruction
 - Activation/désactivation
- Distinction entre
 - Servant
 - Objet CORBA

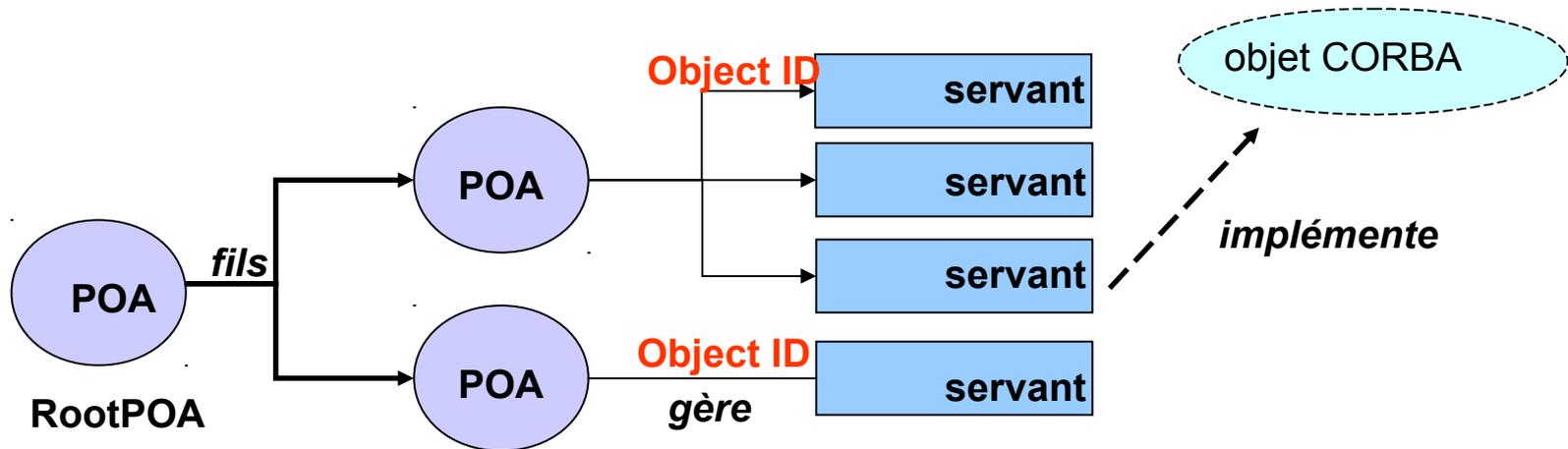


POA – Portable Object Adapter

- Intermédiaire entre l'ORB et l'implémentation d'un objet CORBA (le servant)
- Fournit des services pour
 - La création d'objet CORBA
 - La création de référence
 - L'aiguillage des requêtes aux servants appropriés
 - L'activation/déactivation de servants
 - Décrire des propriétés non fonctionnelles des servants
 - Politiques de : multithreading, persistance, ...

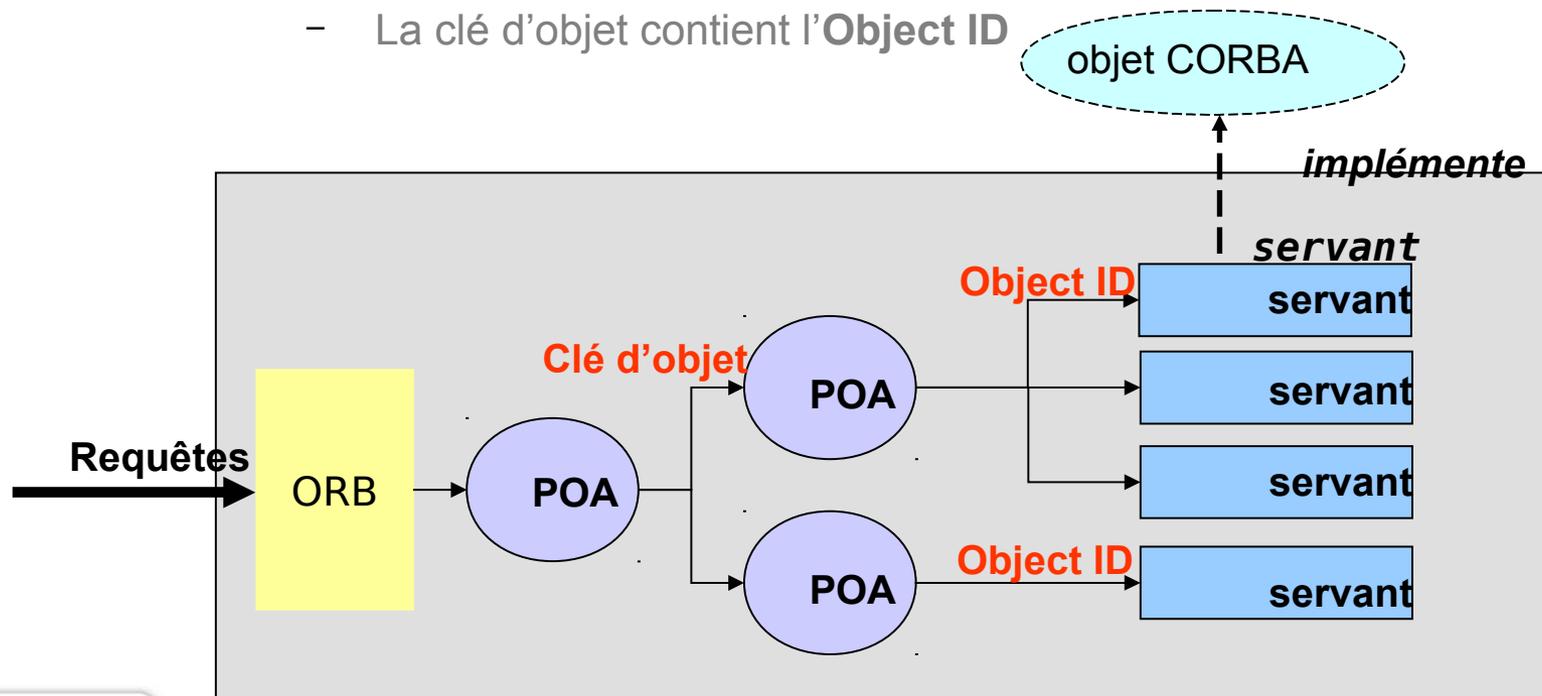
Hierarchie de POA

- Les POA sont organisés en arbre
 - La racine est le RootPOA
 - Chaque POA peut avoir des politiques différentes
- Un POA peut gérer plusieurs objets CORBA
- Un servant est identifié via son ObjectID dans son POA



Cheminement dans le POA

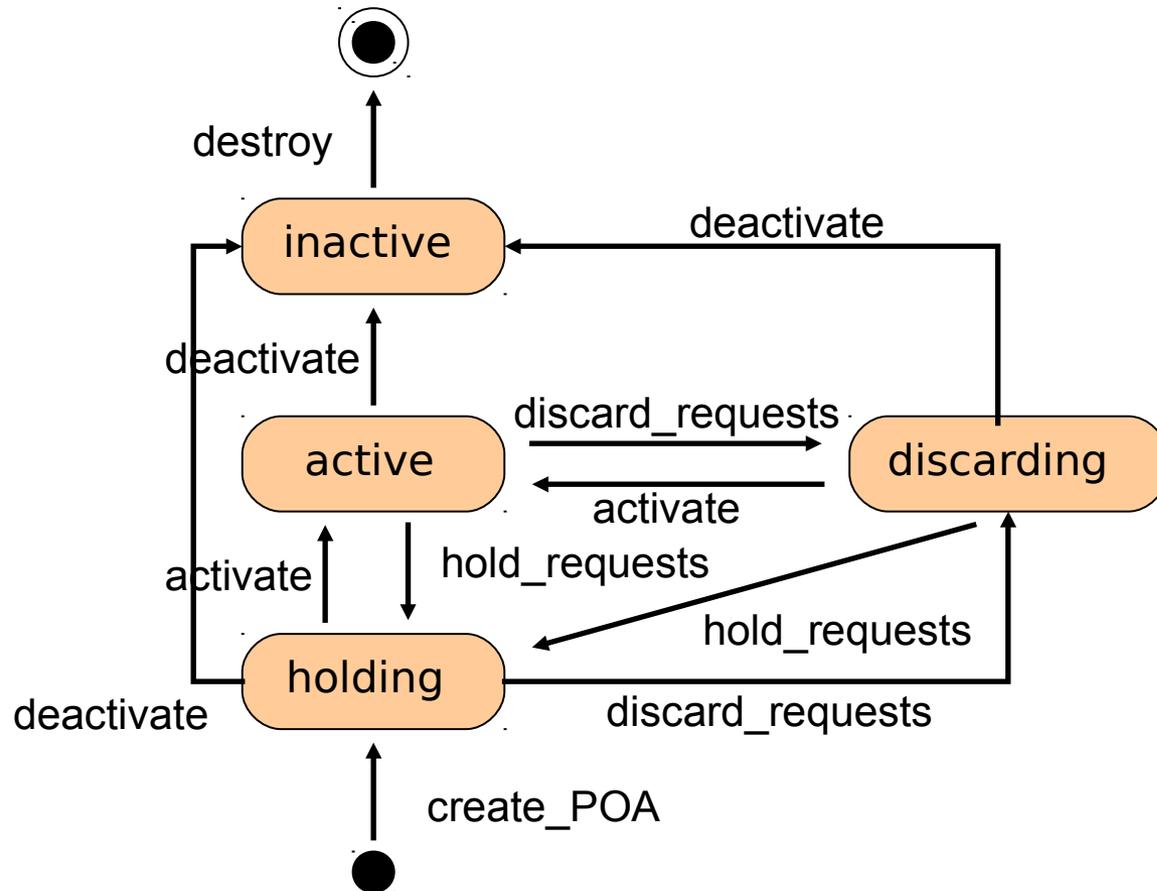
- Référence d'objet: type opaque
 - Doit permettre à l'ORB de trouver le POA associé via le champ "clé d'objet" (*object key*)
 - La clé d'objet contient l'**Object ID**



Gestion des POA

- Les POA sont gérés par des POAManager
 - Tout POA à son POAManager
 - Un POAManager peut avoir plusieurs POA
 - Permet d'avoir des opérations groupées atomiques
- Les gestionnaires de POA contrôlent l'état des POA
 - **Holding** : stocke les requêtes (en nombre limité puis discard)
 - **Active** : traite les requêtes (par ex. délivre aux servants)
 - **Discard** : détruit les requêtes (retourne TRANSIENT)
 - **Inactive** : détruit les requêtes (non-spécifié)
-

Diagramme d'état du POAManager



IDL du POAManager

```
#pragma prefix "omg.org"
module PortableServer
{
  local interface POAManager
  {
    exception AdapterInactive{};
    enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};

    void activate() raises(AdapterInactive);
    void hold_requests( in boolean wait_for_completion)
      raises(AdapterInactive);
    void discard_requests( in boolean wait_for_completion)
      raises(AdapterInactive);
    void deactivate( in boolean etherealize_objects, in boolean wait_for_completion)
      raises(AdapterInactive);
    State get_state();
  };
  local interface POA
  {
    readonly attribute POAManager the_POAManager;
  };
};
```

Références et activation

- Opérations distinctes :
 - Création de référence

```
local interface POA {  
    // reference creation operations  
    Object create_reference ( in CORBA::RepositoryId intf) raises (WrongPolicy);  
    Object create_reference_with_id ( in ObjectId oid, in CORBA::RepositoryId intf );  
};
```

- Activation (incarnation) d'un objet

```
local interface POA {  
    ObjectId activate_object( in Servant p_servant)  
        raises (ServantAlreadyActive, WrongPolicy);  
    void activate_object_with_id( in ObjectId id, in Servant p_servant)  
        raises (ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy);  
    void deactivate_object( in ObjectId oid)  
        raises (ObjectNotActive, WrongPolicy);  
};
```

- Activation **implicite** par défaut (dans le RootPOA)

Fonctionnement du POA

- Idée intuitive : une table ObjectID -> servant

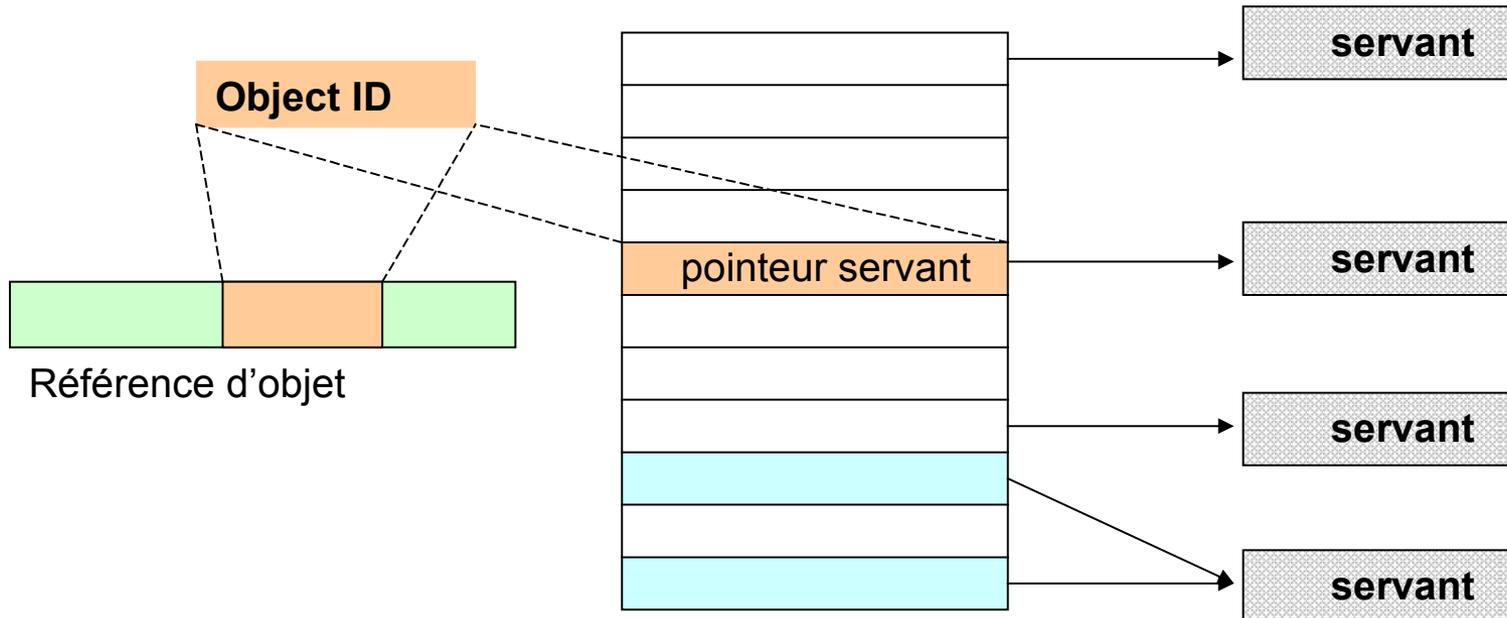
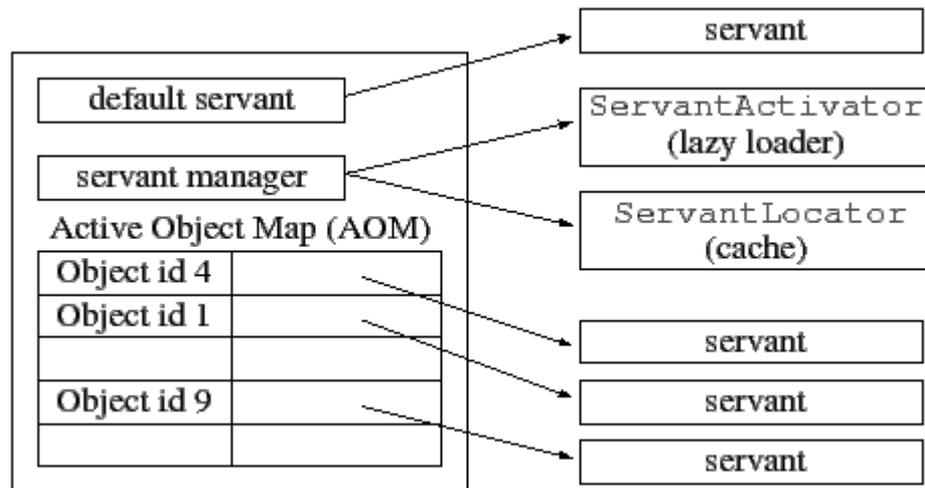


Tableau des objets actifs
POA Active Object Map

Fonctionnement du POA

- Un peu plus qu'une simple table...
 - Servant par défaut – incarne tous les objets du POA
 - Servant activator – instantiation à la demande
 - Servant locator – démultiplexage à la charge du programmeur
 - Cache, équilibrage de charge, forward, etc.



Politiques du POA

- Thread - ThreadPolicy
 - Séquentiel, main thread, multithread
- Durée de vie – LifeSpanPolicy
 - Éphémère, persistant
- Unicité des ObjectID – IdUniquenessPolicy
 - Un servant par objet, un servant pour plusieurs objets (sans état)
- Identification des objets – IdAssignmentPolicy
 - Par l'utilisateur, par le système
- Rétention des associations – ServantRetentionPolicy
 - Retenir l'Id d'un servant dans l'active object map
- Association des requêtes aux servants – RequestProcessingPolicy
 - Active Object Map, default servant, servant manager
- Activation implicite – ImplicitActivationPolicy
 - Activation automatique à la création de la référence

Politiques du POA - Référence

```
// IDL
#pragma prefix "omg.org"
module PortableServer {

    enum ThreadPolicyValue { ORB_CTRL_MODEL, SINGLE_THREAD_MODEL, MAIN_THREAD_MODEL };
    local interface ThreadPolicy : CORBA::Policy { readonly attribute ThreadPolicyValue value; };
    enum LifespanPolicyValue { TRANSIENT, PERSISTENT };
    local interface LifespanPolicy : CORBA::Policy { readonly attribute LifespanPolicyValue value; };
    enum IdUniquenessPolicyValue { UNIQUE_ID, MULTIPLE_ID };
    local interface IdUniquenessPolicy : CORBA::Policy { readonly attribute IdUniquenessPolicyValue value; };
    enum IdAssignmentPolicyValue { USER_ID, SYSTEM_ID };
    local interface IdAssignmentPolicy : CORBA::Policy { readonly attribute IdAssignmentPolicyValue value; };
    enum ImplicitActivationPolicyValue { IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION };
    local interface ImplicitActivationPolicy : CORBA::Policy { readonly attribute ImplicitActivationPolicyValue v; };
    enum ServantRetentionPolicyValue { RETAIN, NON_RETAIN };
    local interface ServantRetentionPolicy : CORBA::Policy { readonly attribute ServantRetentionPolicyValue val; };
    enum RequestProcessingPolicyValue { USE_ACTIVE_OBJECT_MAP_ONLY, USE_DEFAULT_SERVANT, USE_SERVANT_MANAGER };
    local interface RequestProcessingPolicy : CORBA::Policy { readonly attribute RequestProcessingPolicyValue v; };

    local interface POA {
        // Factories for Policy objects
        ThreadPolicy          create_thread_policy          ( in ThreadPolicyValue value);
        LifespanPolicy        create_lifespan_policy        ( in LifespanPolicyValue value);
        IdUniquenessPolicy    create_id_uniqueness_policy   ( in IdUniquenessPolicyValue value);
        IdAssignmentPolicy    create_id_assignment_policy   ( in IdAssignmentPolicyValue value);
        ImplicitActivationPolicy create_implicit_activation_policy ( in ImplicitActivationPolicyValue value);
        ServantRetentionPolicy create_servant_retention_policy ( in ServantRetentionPolicyValue value);
        RequestProcessingPolicy create_request_processing_policy ( in RequestProcessingPolicyValue value);
    };
};
```

Contraintes sur les politiques du POA

- Toutes les combinaisons ne sont pas possibles
 - NON_RETAIN incompatible avec l'utilisation du tableau des objets actifs
 - UNIQUE_ID pas de sens avec NON_RETAIN
 - IMPLICIT_ACTIVATION réclame l'utilisation de SYSTEM_ID et RETAIN
-

Combinaison des politiques du POA

- Création automatique de serviant
 - RETAIN et USE_ACTIVE_OBJECT_MAP_ONLY
- Identique + contrôle de l'activation de serviant
 - RETAIN et USE_SERVANT_MANAGER
- Un serviant gère tous les objets inconnus
 - RETAIN et USE_DEFAULT_SERVANT
- Un serviant par requête
 - NON-RETAIN et USE_SERVANT_MANAGER
- Un serviant pour tous les objets
 - NON-RETAIN et USE_DEFAULT_SERVANT
-

Servant Manager

- Interface **ServantActivator**: politique RETAIN
 - Opération appelée par le POA lorsqu'il a besoin d'incarner/désincarner un objet

```
Servant incarnate ( in ObjectId oid, in POA adapter)  
    raises (ForwardRequest);  
void etherealize ( in ObjectId oid, in POA adapter, in Servant serv,  
    in boolean cleanup_in_progress,  
    in boolean remaining_activations);
```

- Interface **ServantLocator** : politique NON_RETAIN
 - Opération appelée pour chaque requête

```
Servant preinvoke( in ObjectId oid, in POA adapter,  
    in CORBA::Identifier operation, out Cookie the_cookie)  
    raises (ForwardRequest );  
void postinvoke( in ObjectId oid, in POA adapter,  
    in CORBA::Identifier operation, in Cookie the_cookie,  
    in Servant the_servant );
```

Durée de vie, persistance

- Durée de vie de l'objet CORBA par rapport au servant
 - Éphémère (TRANSIENT) : l'objet est détruit à la destruction du POA
 - Persistant (PERSISTENT) : l'objet survit au POA
 - Les références restent valides au redémarrage du serveur
- CORBA ne s'occupe pas de sauvegarder l'état interne de l'objet
 - Seule la **référence** est persistante
- Combinable avec les autres politiques
 - Par exemple ServantManager pour réinstancier automatiquement un servant au démarrage
 - Utilisation habituelle avec un *daemon* tel que orbd

Création d'un POA avec persistance

```
// résolution du RootPOA
POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

// création de la politique PERSISTENT
Policy[] persistentPolicy = new Policy[1];
PersistentPolicy[0] =
    rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT);

// création du nouveau POA
POA persistentPOA = rootPOA.create_POA("childPOA", null,
    persistentPolicy);

// Activation du nouveau POA, i.e. passage de l'état HOLD à ACTIVE
persistentPOA.the_POAManager().activate( );

// enregistrement d'un servent dans le POA persistent
persistentPOA.activate_object(servant);
```

CORBA 3.0

CORBA Component Model

La composition

- La notion de composant est très largement répandue...
 - Composants électroniques
 - Composants d'une voiture
 - Composants d'un meuble
- C'est un ancien rêve de l'informatique
 - Les premiers travaux datent des années 60
- Rôle principal d'un composant : être **composé** !
 - Les composants prônent :
 - la préfabrication
 - la réutilisation

Définition par les propriétés

- Un composant est défini par ses propriétés:
 - 1. c'est une unité de déploiement indépendant
 - 2. c'est une unité de composition par une tierce entité
 - 3. un composant n'a pas d'état persistant
- Remarques
 - -> bien séparé par rapport à l'environnement
 - -> pas de déploiement partiel
 - -> pas d'accès aux détails
 - -> notion de copie non définie
 - Soit le composant est disponible soit il ne l'est pas

De CORBA 2...

- Un modèle orienté objet distribué
 - Hétérogénéité: OMG Interface Definition Language
 - Portabilité: des projections standardisées
 - Interopérabilité: GIOP / IIOP
 - Plusieurs modèles d'invocation: SII, DII et AMI
 - Middleware: ORB, POA, etc.
- Pas de support standard pour l'empaquetage et le déploiement
- Programmation **explicite** des propriétés non fonctionnelles!
 - cycle de vie, (de)activation, service de nomage, de courtier, de notification, de persistance, de transactions, ...
- Pas de notion d'**architecture logicielle**

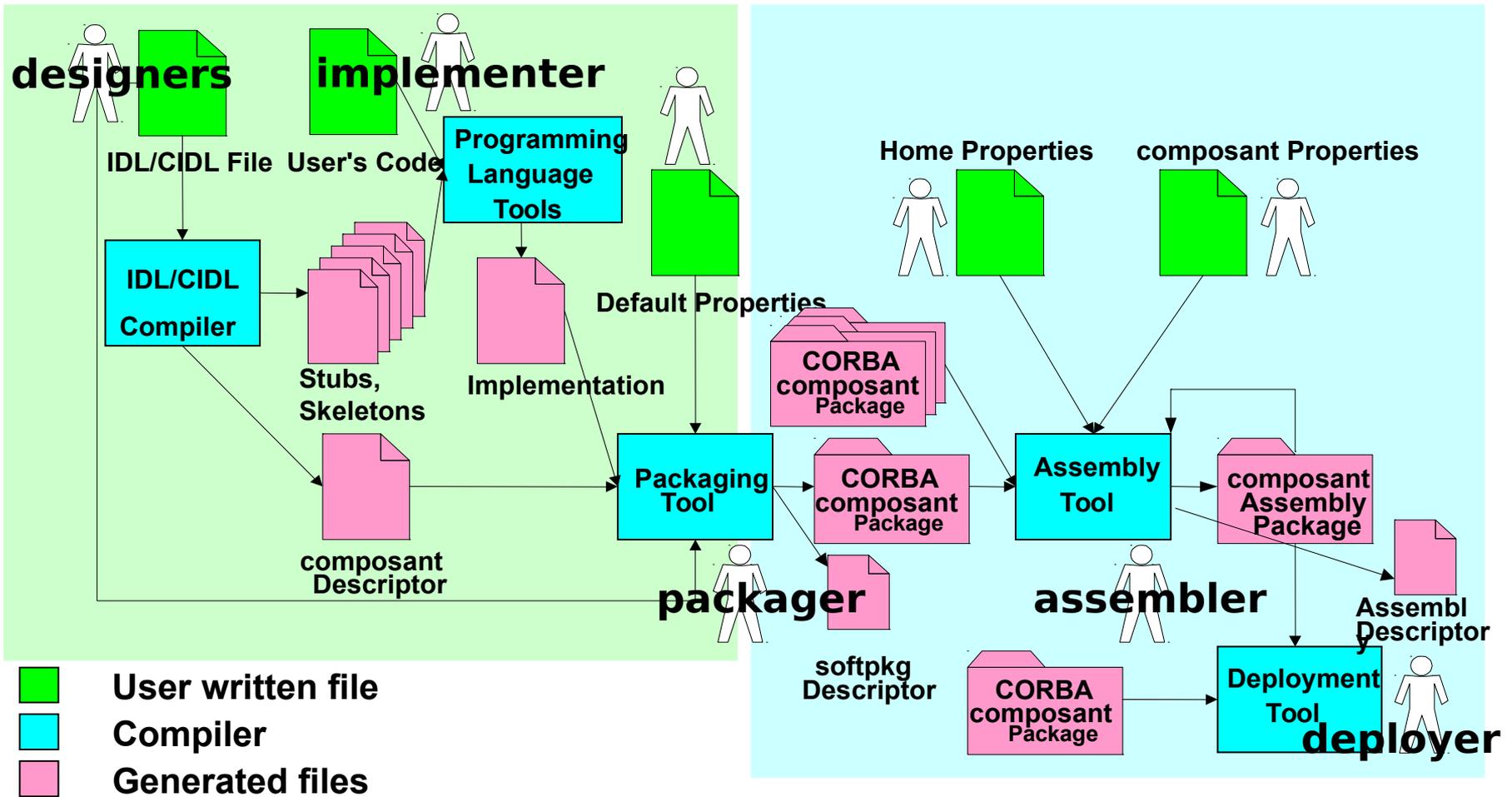
...au modèle de composants CORBA 3

- Un modèle orienté **composant** distribué
 - Une architecture pour définir des composants et leurs interactions
 - Une technologie d’empaquetage pour déployer des binaires exécutable multi-langages
 - Un framework à conteneur pour gérer le cycle de vie, (de)activation, sécurité, transactions, persistance et les évènements
 - Interopérabilité avec Enterprise Java Beans (EJB)
- Le premier modèle de composant industriel multi-langages
 - Multi-langages, multi-OSs, multi-ORBs, multi-vendeurs, etc.
 - CORBA 3.0 – Juillet 2002

Contenu des spécifications CCM

- Un modèle abstrait de composants
 - Étend l'IDL et le modèle objet
- Framework d'implémentation des composants (CIF)
 - Composant Implementation Definition Language (CIDL)
- Un modèle de programmations des conteneurs de composants
 - Vues de l'implémenteur et du client
 - Intégration avec les services de sécurité, de persistance, de transactions et d'évènements
- Des facilités d'empaquetage et de déploiement
 - Fichiers de descriptions en XML
- L'interopérabilité avec EJB via des passerelles
- Des méta-modèles

CCM big picture



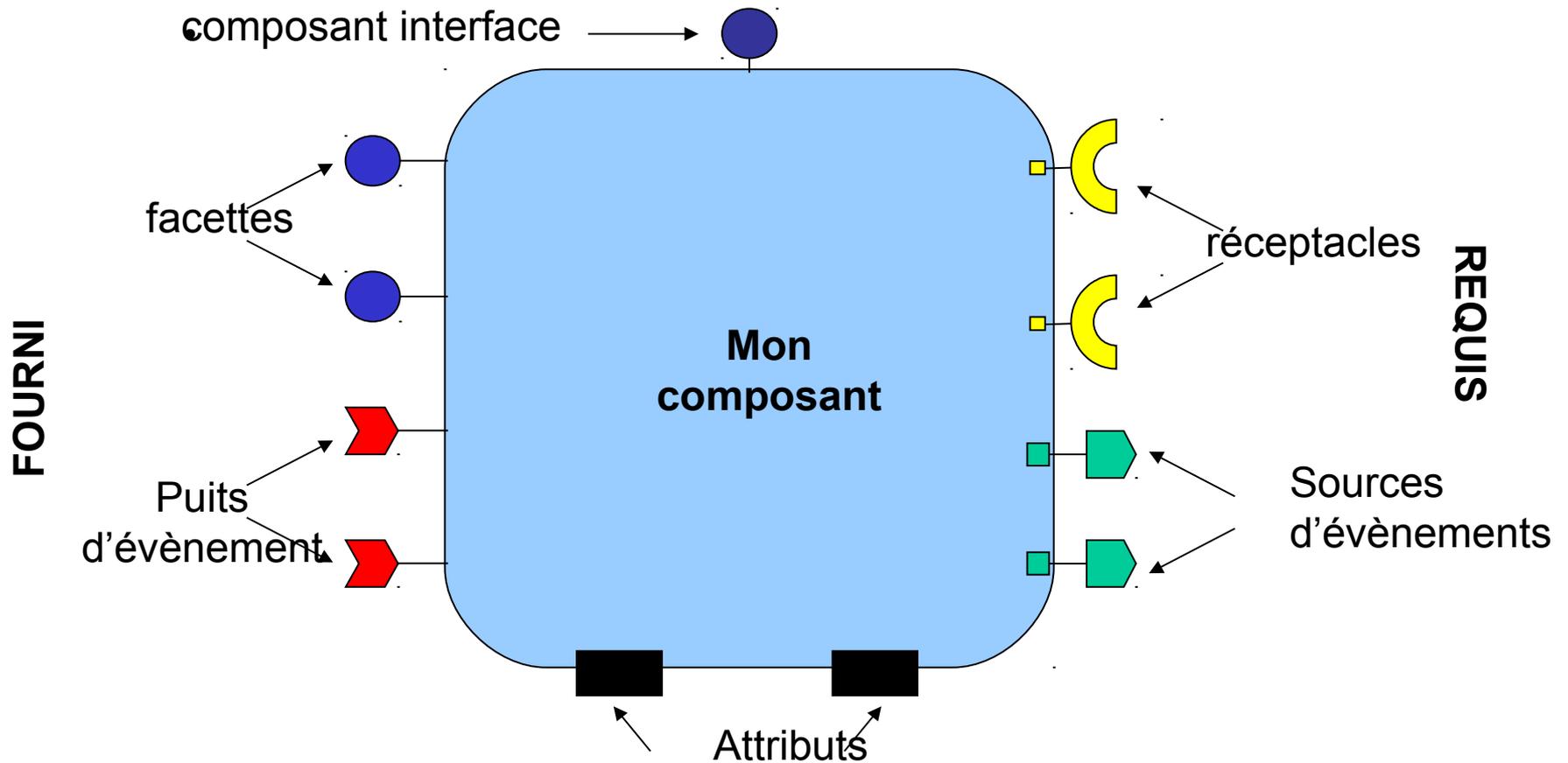
Modèle abstrait des composants

- Il décrit comment un composant CORBA est vu par les autres composants et par les clients
 - Ce qu’offre un composant aux autres composants (*provide*)
 - Ce qu’il demande des autres composants (*use*)
 - Le modèle de collaboration utilisé entre composant
 - Synchrones via les invocations d’opération
 - Asynchrone via la notification d’évènement
 - Quelles propriétés du composant sont configurables
 - Quelles sont les opérations du cycle de vie (*home*)

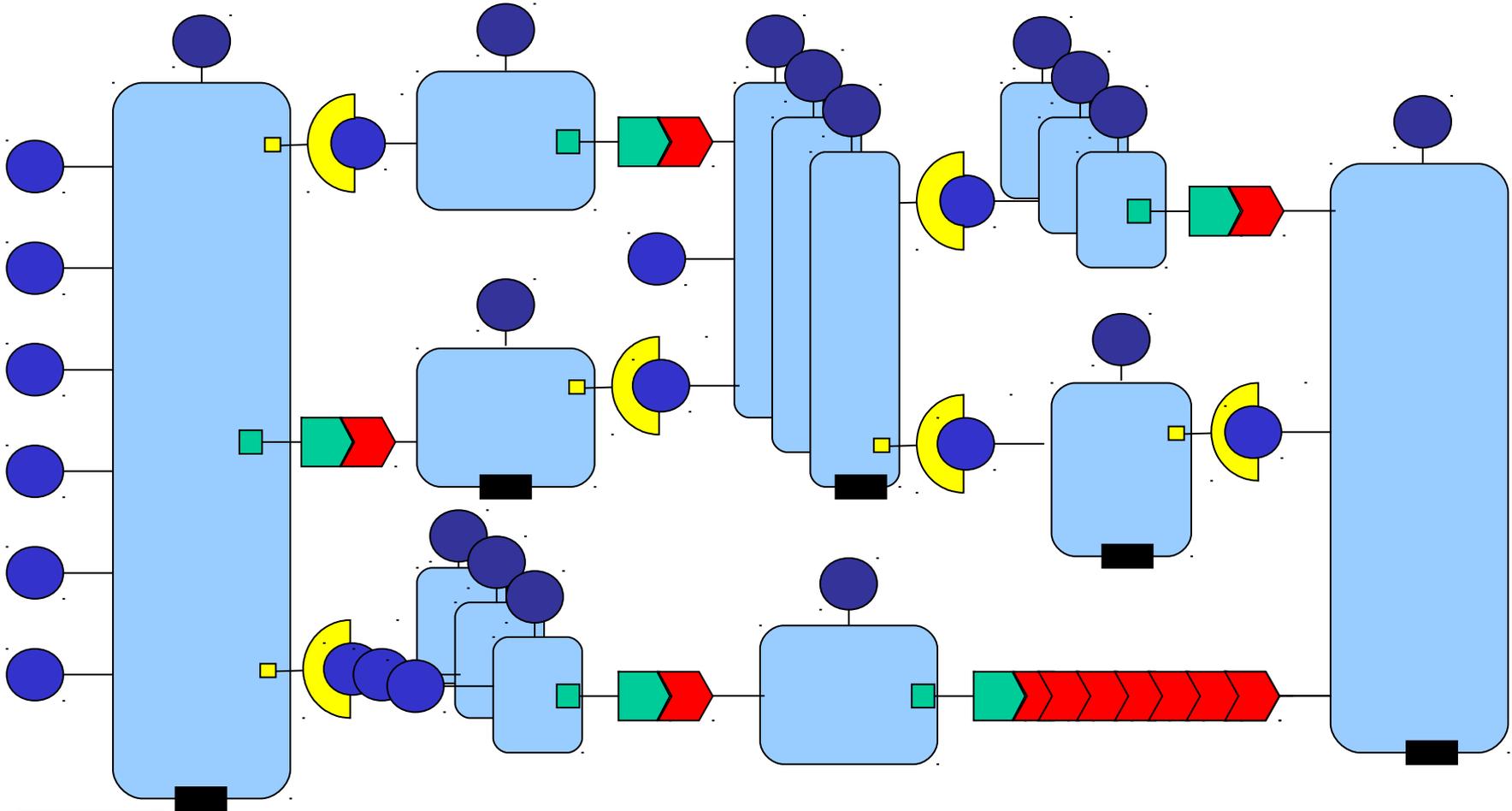
Un composant CORBA

- Un composant est un nouveau meta-type CORBA
 - Extension d'un objet (avec quelques contraintes)
 - Possède une interface et une référence d'objet
 - Une utilisation stylisée des interfaces/objets CORBA
- Fournit des caractéristiques de composant, nommées ports
 - 5 types de port
 - **Attributs** : propriétés configurables
 - **Facettes** : interfaces d'opérations fournies
 - **Réceptacles** : interfaces d'opérations requises
 - **Sources d'évènement** : produit des évènements
 - **Puits d'évènement** : consomme des évènements

Un composant CORBA



Construction d'une application = assemblage



Définition d'un composant

- Composant simple

```
component nom_composant { ...};
```

- Héritage simple de composant

```
component nom_composant : composant_pere { ...};
```

- Peut supporter plusieurs interfaces (supports)

```
component nom_composant supports interface1, interface2  
{ ...};
```

Les attributs

```
component nom_composant
{
  attribute A;
  readonly attribute B;
};
```

- Propriétés configurables nommées
 - Clé vitale pour une réutilisation effective
 - Vise la configuration d'un composant
 - ex.: comportement optionnel, modalité, etc.
 - Peut lever des exceptions
 - Exposées via des accès en lecture / écriture
- Peuvent être configurés
 - Par des mécanismes visuels dans des environnement d'assemblage et/ou de déploiement
 - Par les maisons ou par les implémentations durant l'initialisation
 - Potentiellement non modifiable par la suite

Les facettes

- Interfaces distinctes nommées qui fournissent les fonctionnalités du composant aux clients
- Chaque facette correspond à une vue du composant, c'est à dire à un rôle
- Une facette représente le composant lui-même, pas une chose séparée et contenue dans le composant
- Les facettes ont des références d'objet indépendantes

```
component nom_composant
{
  provides type nom_port;
  ...
};
```

Les réceptacles

- Points distincts et nommés de connexion pour une connectivité potentielle
 - Possibilité de spécialiser par délégation ou de composer des fonctions
 - ~ la base d'un Lego !
- Stocke une référence d'objet simple ou multiple
 - Mais n'est pas destiné à un être un service de mise en relation
- Configuration
 - Statiquement durant l'initialisation ou l'assemblage
 - Dynamiquement à l'exécution

```
component nom_composant
{
  uses type portA;
  uses multiple type portB;
  ...
};
```

Les évènements

- Modèle d'évènement publication / souscription (*publish/subscribe*)
 - Modèle “*push*” seulement
 - Sources (2 types) et puits
- Les évènements sont des ValueTypes
 - Défini par le nouveau meta-type `eventtype`
 - Spécialisation de `valuetype` pour les composants

```
eventtype nom_evenement  
{  
  public string A;  
  ...  
}
```

Les sources d'évènements

- Points de connexion nommés pour la production d'évènement
 - Pousse un `eventtype` spécifié
- Deux types: publieur & émetteur
 - `publishes` = plusieurs clients peuvent souscrire
 - `emits` = seulement un client connecté
- Un client souscrit directement à une source d'évènement
- Le conteneur gère l'accès aux canaux de notification
 - extensibilité, qualité de service, transactionnel, etc.

```
component nom_composant
{
    publishes etype portE;
    emits etype portF;
    ...
};
```

Les puits d'évènements

- Points de connexion nommés dans lesquels des évènements de type spécifié peuvent être « poussés »
- Souscription aux sources d'évènements
 - Potentiellement plusieurs (n vers 1)
- Pas de distinction entre émetteur et publieur
 - Les deux « poussent » dans un puit d'évènement

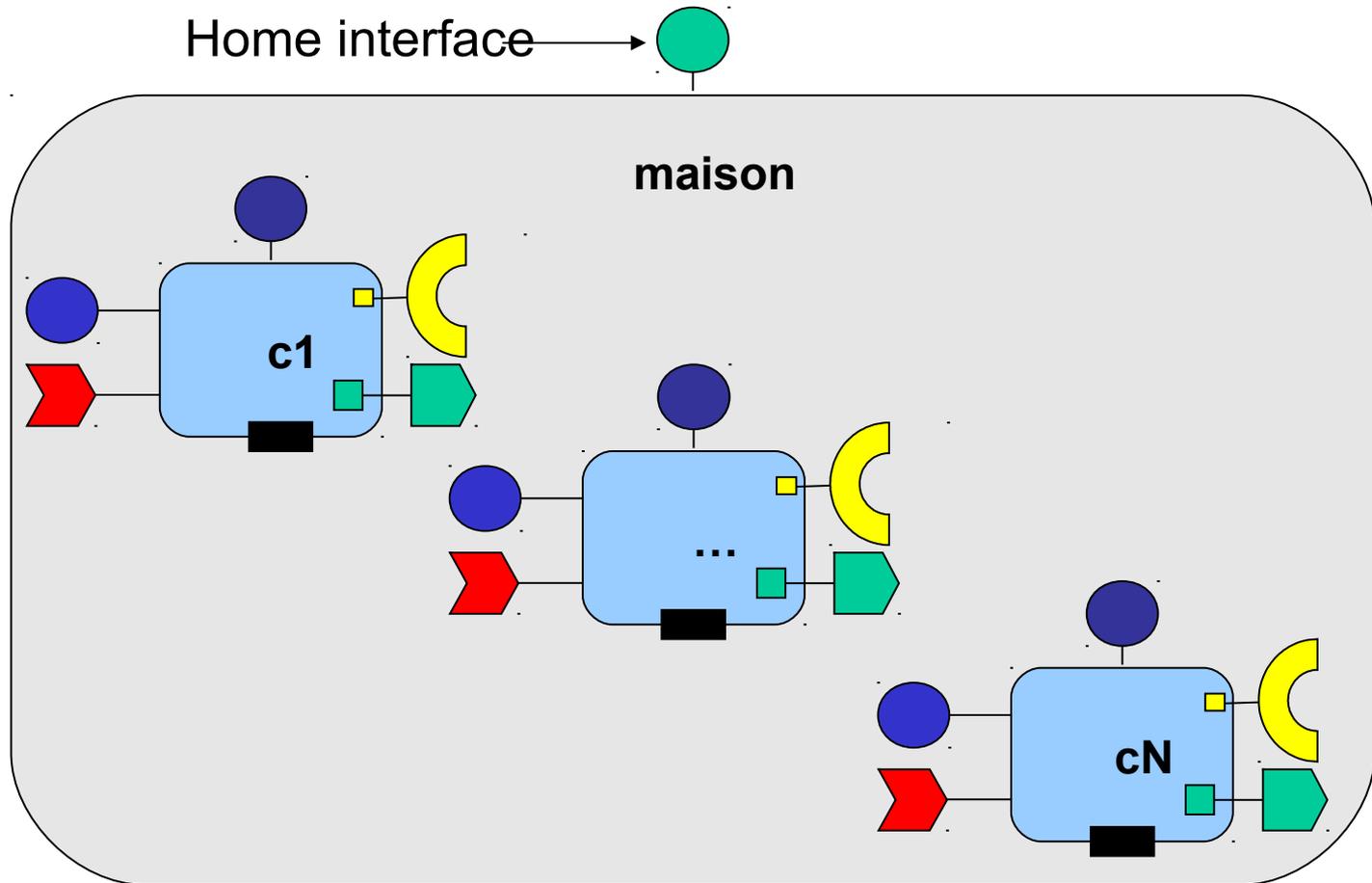
```
component nom_composant  
{  
    consumes etype portE;  
    ...  
};
```

Maison de composant CORBA

- Chaque instance de composant est créée et gérée par une maison unique de composant (*home*)
- Gère un unique type de composant
 - Plus d'une maison peuvent gérer un même type de composant
 - Mais une instance de composant n'est gérée que par une seule instance de maison
- Nouveau meta-type CORBA : **home**
 - Définition d'une maison est distincte de celle d'un composant
 - Une maison a une interface et une référence d'objet
- Est instanciée durant le déploiement

```
home maison manages nom_composant {... };
```

Maison de composant CORBA

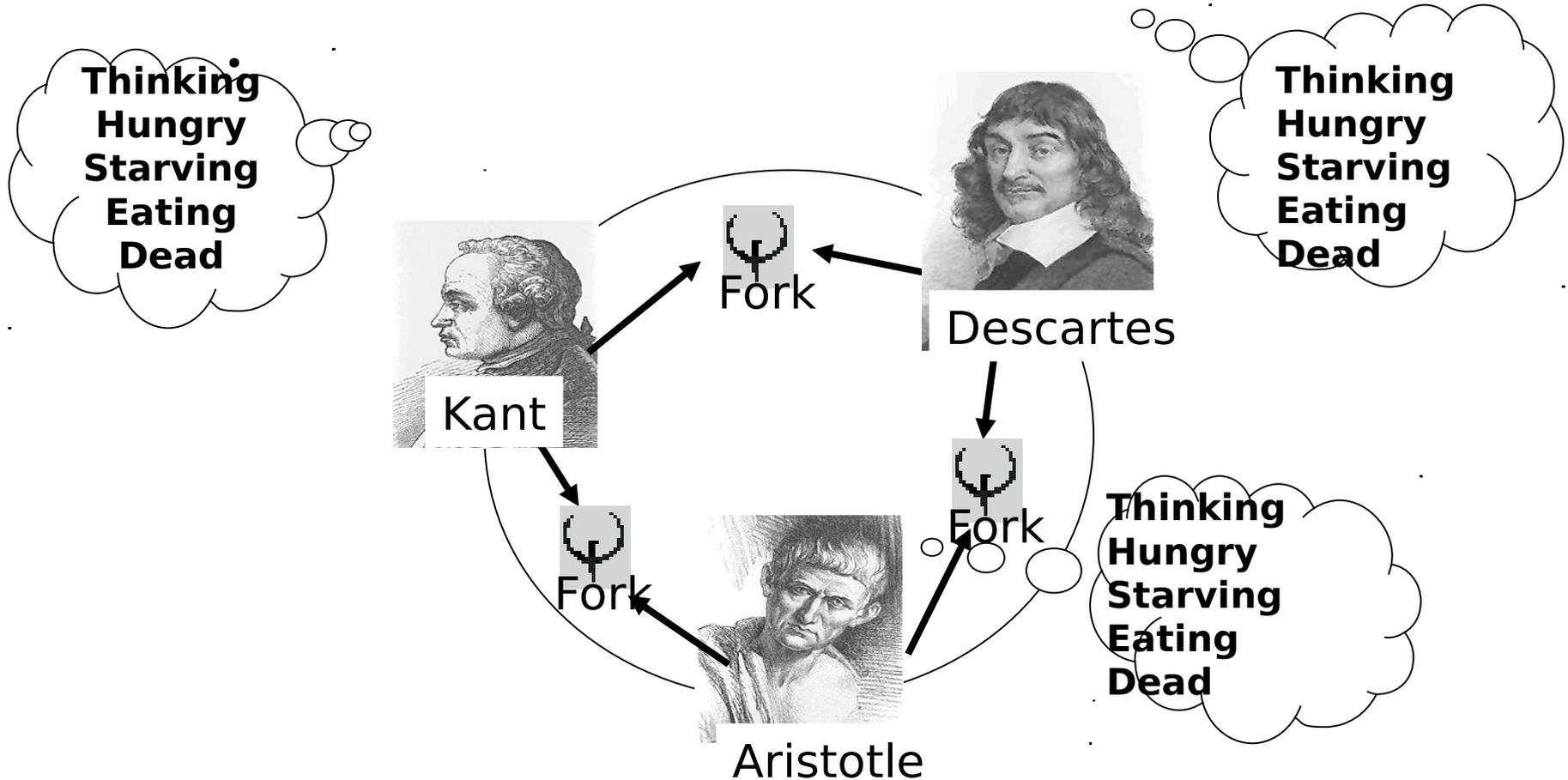


Propriétés des maisons

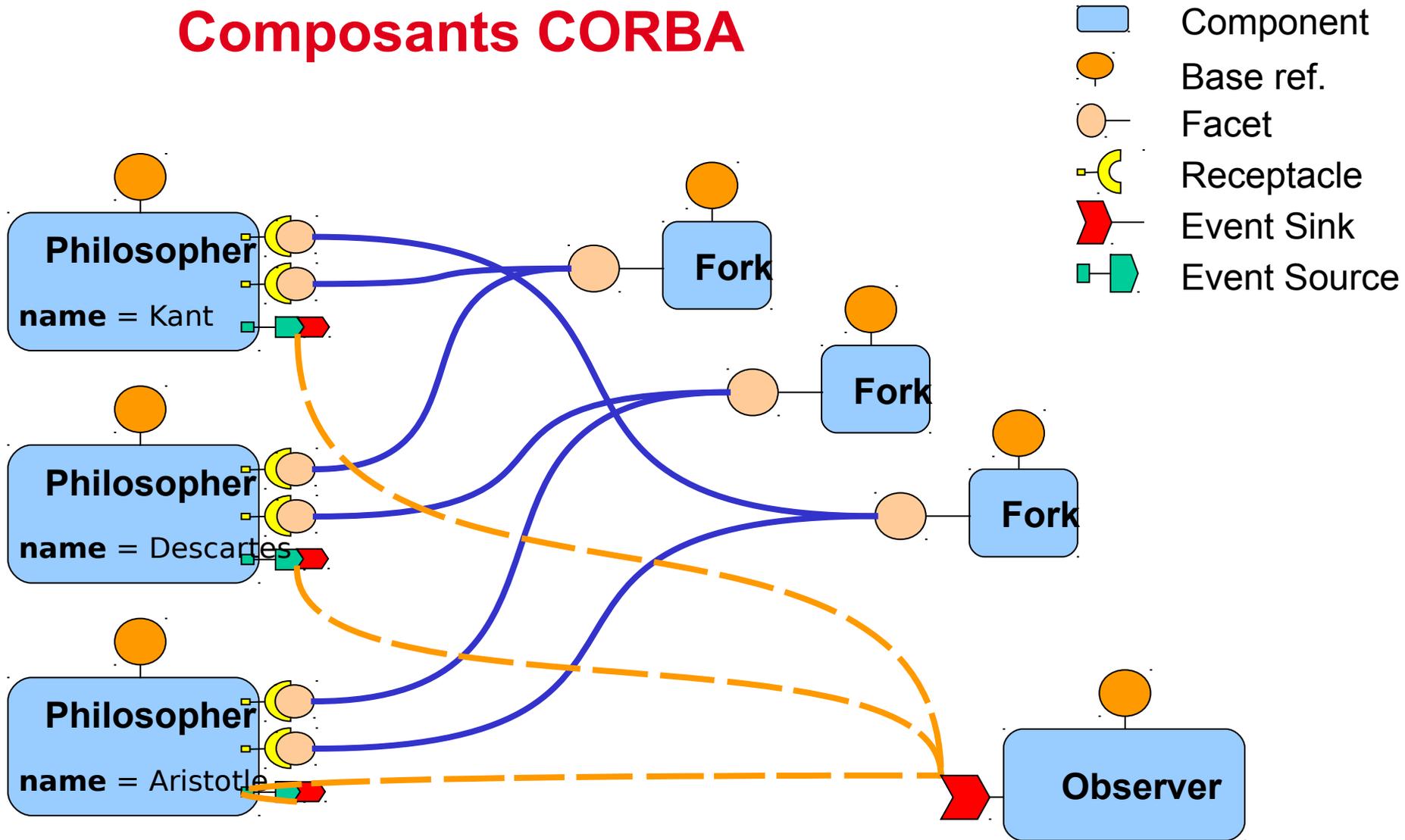
- Permet l'évolution des caractéristiques du cycle de vie ou du type de clé sans changer la définition d'un composant
- Clé primaire optionnelle (`primarykey`) comme identité d'un composant ou comme clé primaire de persistance
- Fournit des opérations standards de création (`factory`) et de recherche (`finder`)
- Extensible par des opérations de niveau utilisateur

```
home maison manages nom_composant primarykey keytype
{
  factory creation(in string name) raises E;
  finder  cherche(in string name) raises F;
  ...
};
```

Exemple : le dîner des philosophes



Composants CORBA



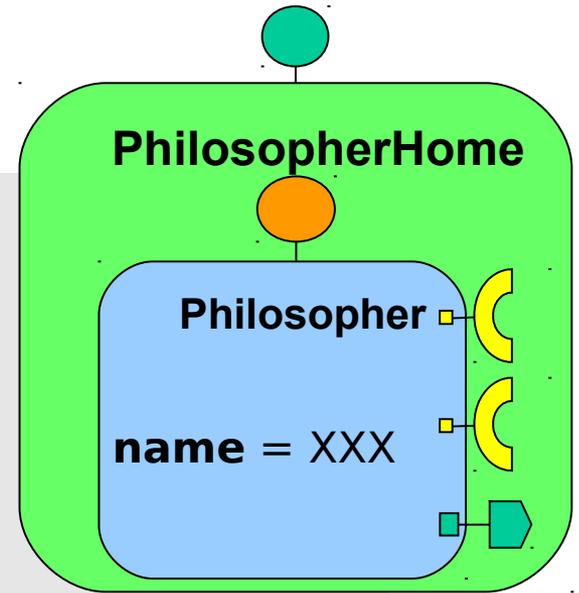
État d'un philosophe

```
enum PhilosopherState
{
    EATING, THINKING, HUNGRY,
    STARVING, DEAD
};

eventtype StatusInfo
{
    public string name;
    public PhilosopherState state;
    public unsigned long ticks_since_last_meal;
    public boolean has_left_fork;
    public boolean has_right_fork;
};
```

Le composant philosophe

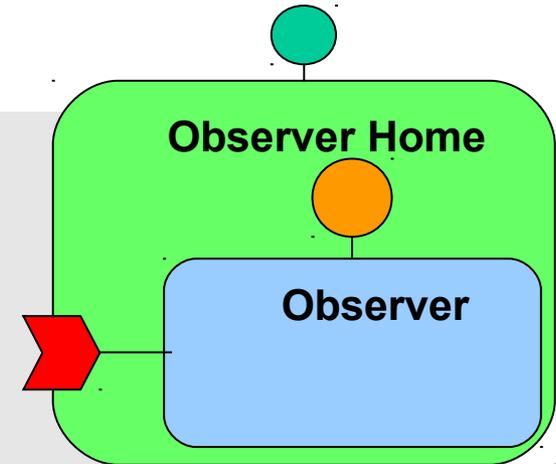
```
component Philosopher {  
  attribute string name;  
  
  // The left fork receptacle.  
  uses Fork left;  
  
  // The right fork receptacle.  
  uses Fork right;  
  
  // The status info event source.  
  publishes StatusInfo info;  
};  
  
home PhilosopherHome manages Philosopher {  
  factory new(in string name);  
};
```



Le composant observateur

```
component Observer
{
  // The status info sink port.
  consumes StatusInfo info;
};

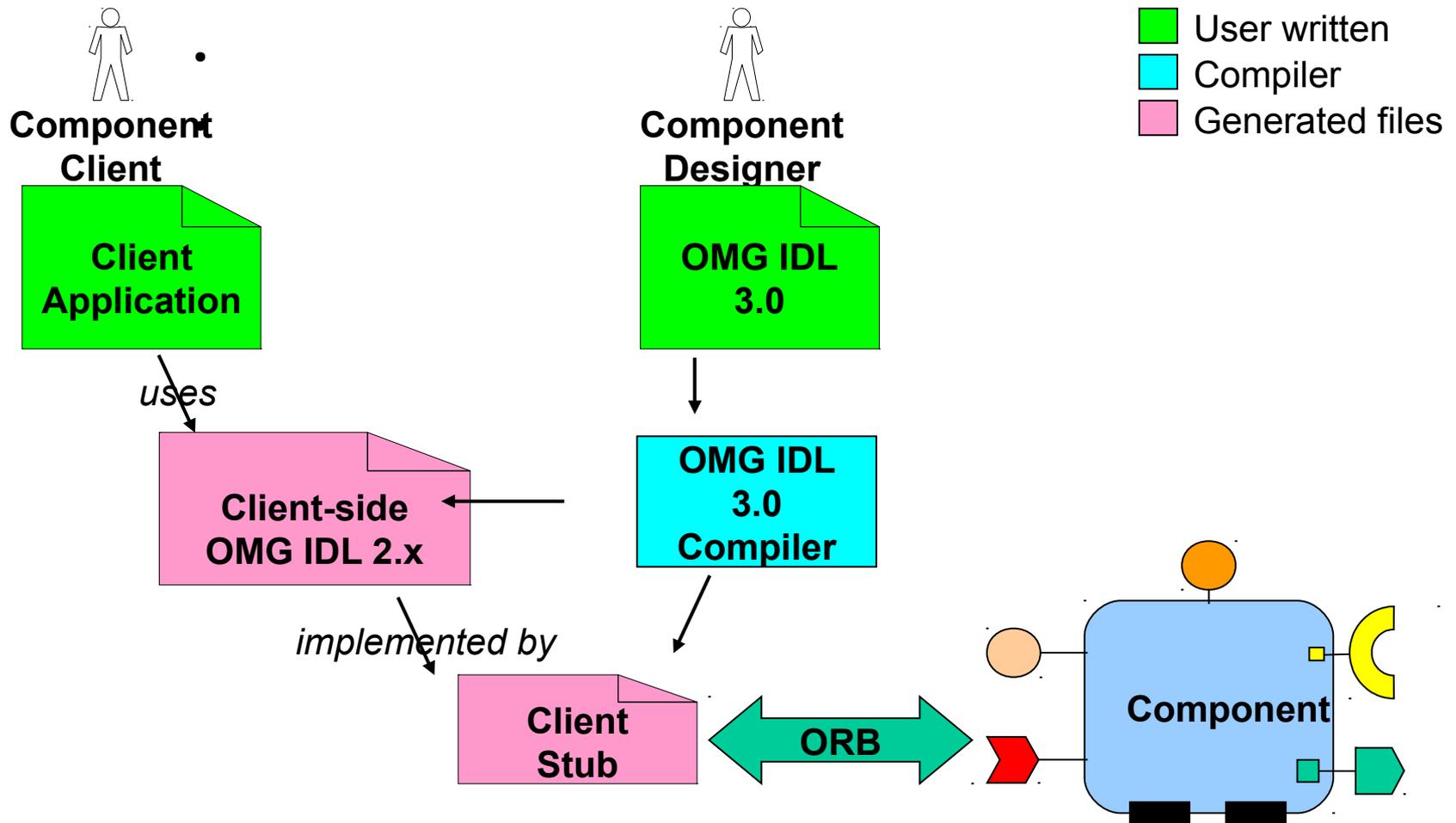
// Home for instantiating observers.
home ObserverHome manages Observer {};
```



Projection côté client

- Deux patrons de conception
 - Fabrique : recherche d'une maison et utilisation pour créer une instance d'un composant
 - Recherche : recherche d'un composant existant via les services de nomage, de courtage ou les opérations de recherche des maisons
- Chaque construction IDL 3.0 à un équivalent en terme d'IDL 2
- Le composant et sa maison sont vus côté client via les projections
- Permet de ne pas changer le langage de projection coté client
 - Les clients peuvent toujours utiliser leurs outils favoris
- Les clients NE sont PAS obligés d'être "component-aware"
 - Ils invoquent juste des opérations sur des interfaces

Projection IDL côté client



Projection des composants

- Exprimé via des extensions à l'IDL 3.0
 - Construction syntaxique pour des patrons de conceptions bien connus
 - Projeté sur les interfaces IDL pour les clients et les implémenteurs
- Exemple :

```
component nom_composant { ... };
```

↓ projeté en

```
interface nom_composant : Components::CCMObject { ... };
```

Projection des opérations

- Facettes

```
provides type portA;
```

↓ projeté en

```
type provide_portA ( );
```

- Réceptacles

```
uses type portB;
```

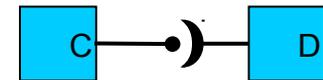
↓ projeté en

```
void connect_portB ( in type conxn )  
  raises ( Components::AlreadyConnected, Components::InvalidConnection );  
type disconnect_portB ( ) raises ( Components::NoConnection );  
type get_connection_portB ( );
```

Connexion de composants

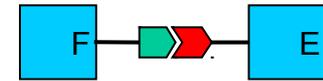
- Facette/réceptacle

```
type_var a = C->provide_portA ();  
D->connect_portB(a);
```

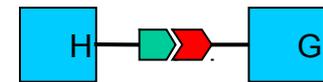


- Source/puit d'évènement

```
etypeConsumer_var p = E->get_consumer_puit();  
F->connect_source(p);
```



```
etypeConsumer_var p = G->get_consumer_puit();  
H->subscribe_source(p);
```



CIDL – Component Implementation Definition Language

- Décrit une composition de composant
 - Entité agrégé qui décrit tous les artefacts requis pour implémenter un composant et sa maison
- Gère la persistance d'un composant
 - Basé sur le OMG Persistent State Definition Language (PSDL)
 - Liens entre des types de stockage et les exécuteurs segmentés
- Génère des squelettes d'exécuteur fournissant
 - La segmentation des exécuteurs des composants
 - Une implémentation par défaut des opérations de callback
 - La persistance de l'état d'un composant
-

Composition CCM

- Entité agrégée décrivant les artefacts requis pour implémenter un composant et sa maison
 - Nom de la composition
 - Catégories de cycle de vie des composants
 - Service, session, process, entity
 - Un type de maison de composant (1)
 - Le type du composant à implémenter est défini implicitement
 - Une définition d'exécuteur de la maison (2)
 - Une définition d'exécuteur du composant (3)
 - Une définition de délégation
 - Une définition pour un proxy de la maison
 - Une association à un espace de stockage abstrait

```
composition <categorie> nom_composition {  
    home executor nom_executeur_maison {           // (2)  
        implements nom_type_maison;                 // (1)  
        manages nom_executeur;                       // (3)  
    }  
};
```

Exécuteur

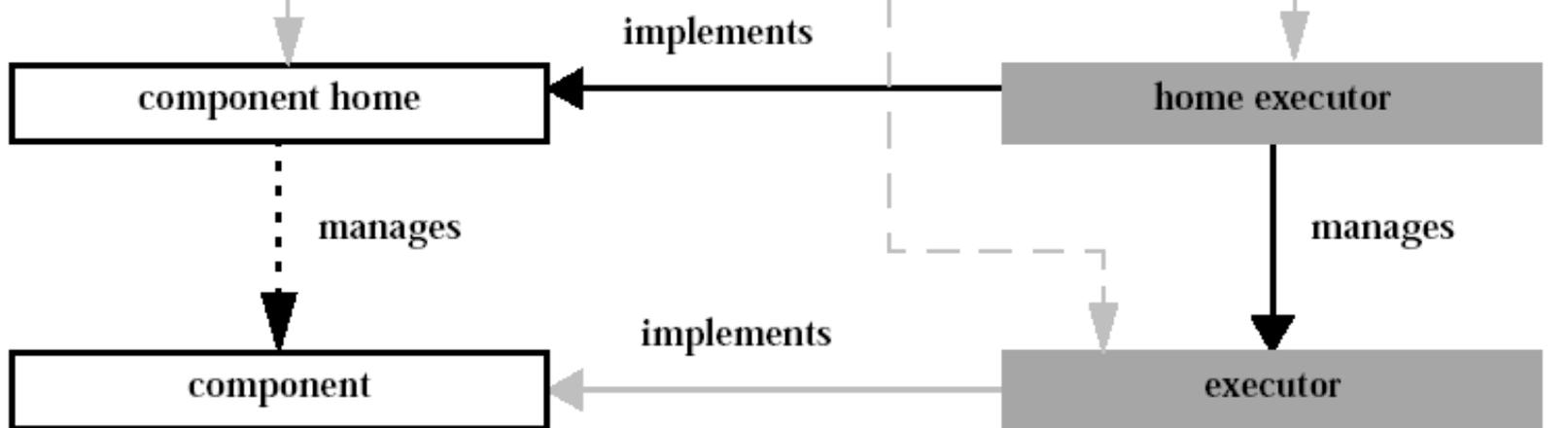
- Artefact de programmation implémentant une abstraction.
- Dans les langages objets, un exécuteur = un objet
- Exécuteur de maison, exécuteur de composant

```
composition <categorie> nom_composition
{
  home executor nom_executeur_maison
  {
    implements nom_type_maison ;
    manages nom_executeur;
  }
};
```

```

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

```

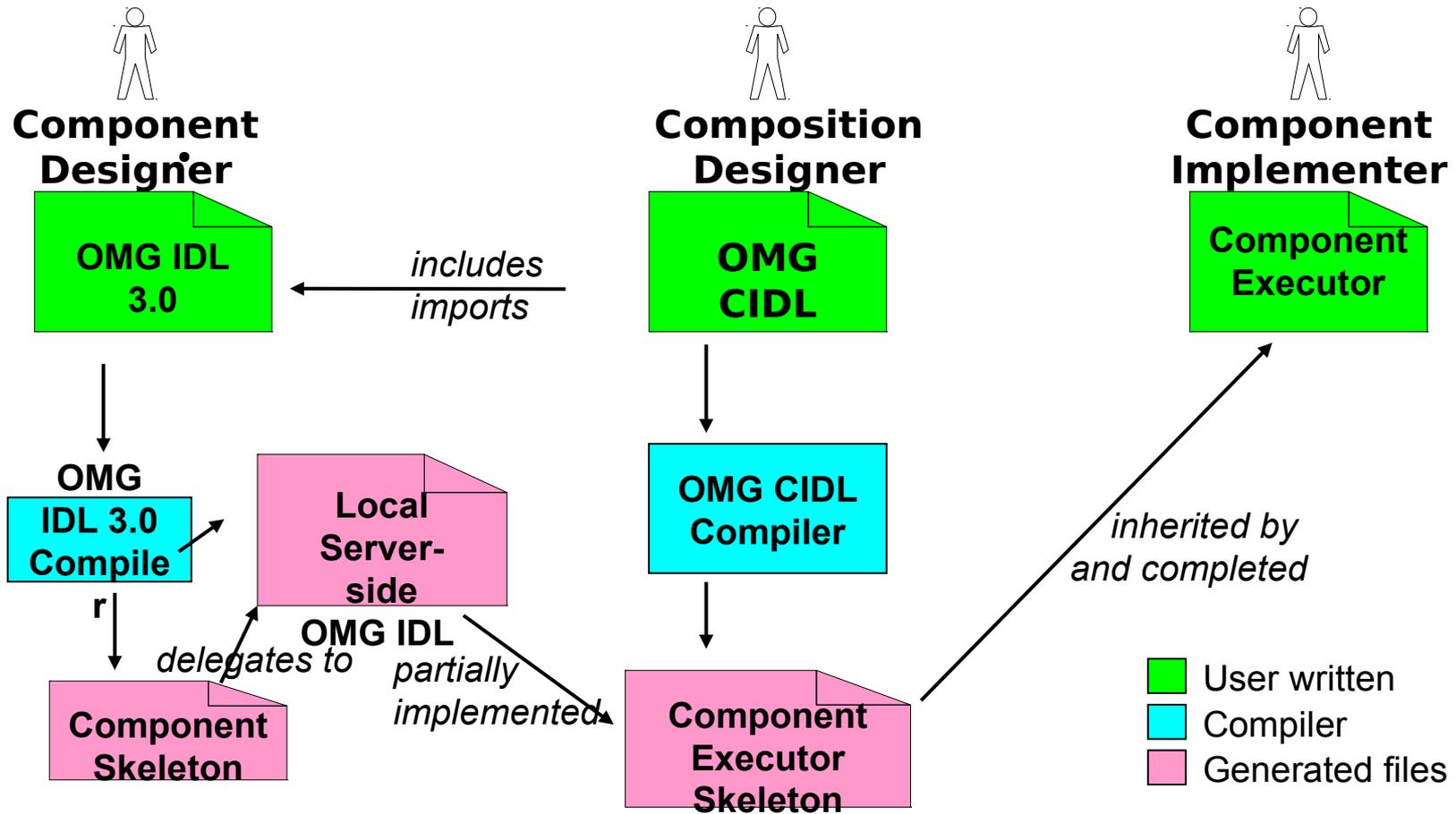


CIDL

IDL

- explicitly defined in composition
- implicitly defined by composition
- explicitly defined elsewhere in IDL/CIDL

Compilation CIDL



CIDL pour l'observateur

```
#include <philo.idl>
// or import DiningPhilosophers;

composition session ForkManagerSessionComposition
{
  home executor ForHomeImpl
  {
    implements DiningPhilosophers::ForkHome;
    manages ForkManagerImpl;
  };
};
```

Implémentation de la maison

```
package DiningPhilosophers.monolithic;
import DiningPhilosophers.*;
public class ForkHomeImpl extends org.omg.CORBA.LocalObject
    implements CCM_ForkHome {
    public ForkHomeImpl(){}
    public org.omg.Components.EnterpriseComponent create()
    {
        return new ForkManagerImpl();
    }
    public static org.omg.Components.HomeExecutorBase create_home()
    {
        return new ForkHomeImpl();
    }
}
```

Implémentation du composant

```
package DiningPhilosophers.monolithic;
import DiningPhilosophers.*;
public class ForkManagerImpl extends org.omg.CORBA.LocalObject
    implements CCM_ForkManager, CCM_Fork,
        org.omg.Components.SessionComponent
{
    private boolean available_;
    public CCM_Fork get_the_fork() { // From CCM_ForkManager
        return this; // Returns an implementation of the Fork facet
    }
    public void get() throws InUse { // From CCM_Fork
        if (! available_) throw new InUse();
        available_ = false;
    }
    public void release() { // From CCM_Fork
        if (available_) return;
        available_ = true;
    }
}
```

Travail à faire

Retour sur l'application bancaire

- Modèle plus réaliste:
 - Asynchronisme des requêtes
 - Encapsulation – l'interface des comptes bancaires n'est accessible qu'au client et à la banque, pas aux autres banques
 - De l'extérieur : numéro de compte
 - Intermédiaire interbancaire
 - Assure la sincérité des échanges
- Propriétés souhaitables
 - Persistance
 - Transactions

Intermédiaire, asynchronisme

- Architecture générale
 - Entité interbancaire : Interbank
 - Maintient un historique des transactions
 - Les banques
 - Se connectent à Interbank
 - Enregistrent un callback pour recevoir des transactions
 - Les clients
 - Se connectent à leur banque
 - Interface privée

Intermédiaire, asynchronisme

- Mettre en place l'architecture générale
 - Types de données échangées (transaction)
 - Interfaces IDL pour :
 - Interbank – le service interbancaire
 - BankTransaction – l'interface de callback pour recevoir des transactions
 - BankCustomer – l'interface destinée aux clients (ouverture de compte, dépôts, retraits, solde)

Virement interbancaire

- Virement asynchrone
 - Pas perdu, même si le serveur destinataire ne tourne pas
- Déroulement de la transaction
 - Banque A -> B, service interbancaire I
 - message : A->I ; puis I -> B
 - une fois exécuté : confirmation B ->I (inscription dans l'historique) ; confirmation I->A (débit du compte)

Observateurs

- Écrire des mini-clients (Java ou C++) pour
 - Réaliser les opérations élémentaires des clients
 - Lancer des ordres de virement
 - Interroger la liste de transactions du service interbancaire

Persistence

- Le service interbancaire doit être persistant
 - Référence constante
 - Instanciation automatique au démarrage
 - Restauration de l'état au démarrage
- Implémentez la persistance à l'aide d'un POA doté de la politique PERSISTENT et du serveur d'application `orbd` et de l'outil `servertool` de Java IDL
- Suivez le tutoriel Oracle :
<http://docs.oracle.com/javase/7/docs/technotes/guides/idl/jidlExample2.html>

Default servant

- Alternative à la persistance
 - Pour l'InterBank, on peut aussi avoir une référence non-persistante, mais un **default servant**
 - On gère alors l'objet comme un **singleton**
 - L'objet est seul dans son POA

Servant Activator

- La banque ne crée pas nécessairement un objet CORBA pour chaque compte existant
 - Créer un POA doté d'un Servant Activator activant les objets à la demande
 - Implémenter les méthodes incarnate/etherealize pour gérer correctement l'état à l'instanciation

À vous de jouer !

inria
informatics mathematics

<http://dept-info.labri.fr/~denis/>