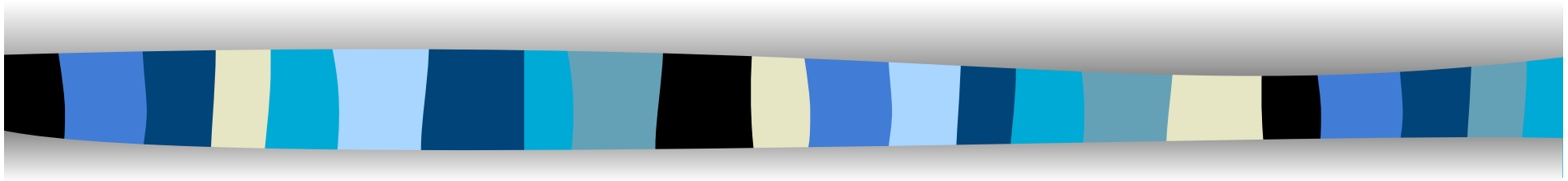
Intégration des objets et des Services



Mounia Fredj & Mahmoud Nassar

ENSIAS

Partie 1 : Intégration des objets

Objectifs du cours

- Etude des concepts de base de CORBA
- Construction d'une application répartie au dessus du bus CORBA
  - exemple d'application (Java)

Bibliographie

M. Geib, C. Gransart, P. Merle,
"Corba : des concepts à la pratique", Masson, 1999.

D. Acremann, G. Moujeard, L. Rousset, "Développer avec Corba en Java et C++", Campus Press, 2° édition, 2000.

R. Orfali and D. Harkey
"Client/Server Programming with Java and CORBA ", Second edition.

B. Traverson
Polycopié cours : " Environnements répartis objets - CORBA ", Janvier 2002.

URLs

- http://www.omg.org/ (OMG Site)
- http://www.orbacus.com/ (IONA – ORBacus Site)
- http://www.cs.wustl.edu/~schmidt/report-doc.html

Plan

1. **Introduction à Corba**
   - Contexte et Motivations
   - Principes fondamentaux de Corba
2. **Modèles, composants et architecture**
3. **Le cycle de développement CORBA**
   - Schéma général de développement
   - Un exemple bancaire
   - Les produits
4. **Mise en œuvre des objets Corba**
   - Référence d'objet – Références initiales
   - Mise en œuvre côté client / côté serveur (héritage, délégation)
   - Exemple d'un service : le service de nommage
5. **Le langage OMG-IDL**
6. **Le mapping Java**
7. **L'adaptateur d'objets POA**

I. Introduction à Corba

Contexte et Motivations

Principes fondamentaux de Corba

CORBA

- Spécification d'un **modèle de composants logiciels répartis** et des **interfaces de l'intergiciel associé**

  CORBA pour Common Object Request Broker Architecture

- Fait partie d'un large ensemble de spécifications définies par le consortium international Object Management Group (OMG)
  - **Object Management Architecture (OMA)**
    - Des services communs
      - Nommage, courtage, notification, transactions, persistance, sécurité, …
    - Des interfaces orientées métiers / domaines
      - Air Traffic Control, Gene Expression, Software Radio, Workflow Management, …
  - **Model Driven Architecture (MDA)**
    - Des technologies d'ingénierie dirigée par les modèles
      - Unified Modeling Language (UML)
      - Meta Object Facilities (MOF)
      - XML Metadata Interchange (XMI)

Contexte : environnement **réparti**

- Un système est dit **réparti** :
  - s'il gère plusieurs ressources distantes reliées par un réseau de communication
- Une application est dite **répartie** :
  - si elle a besoin pour s'exécuter de ressources distantes.
    - Ces ressources : données, traitements ou utilisateurs.
- A opposer aux systèmes « centralisés »
  - Ordinateur central (mainframe)
  - Ordinateur personnel (personal computer)

Attraits de la répartition

- **Organisationnels**
  - Décentralisation des responsabilités
  - Découpage en unités autonomes et mobiles
- **Amélioration des performances**
  - Réduction des coûts de communication
  - Partage des charges entre processeurs
- **Fiabilité et disponibilité**
  - Individualisation des défaillances
  - duplication

ENSIAS

Contraintes de la répartition

- **Pas de vision globale instantanée**
  - Délais de transmission dus aux communications
  - Indéterminisme dû au parallélisme

- **Programmation et administration plus difficiles**
  - Mise au point (gestion des erreurs, suivi des exécutions)
  - Installation, configuration, surveillance et maintenance

Caractéristiques des applications réparties

- **Hétérogénéité** matériel, logiciel et communication
  - faire communiquer des systèmes et intégrer des composants d'origines diverses
- **Portabilité**
  - application développée sur une machine peut s'exécuter sur une autre machine sans réécriture
- **Interopérabilité**
  - des composants d'une appli. répartie doivent communiquer même s'ils s'exécutent sur des systèmes hétérogènes

Types d'environnements

- serveurs de BD (ex: Oracle)
- serveurs de transactions (Encina Transarc/IBM)
- serveurs de groupware (ex: Lotus notes, Overapps)
- **serveurs d'objets** (communicant par ORB)
- serveurs Web (HTTP, CGI, Applet…)
- …

Techniques existantes (1/3)

- Traditionnels **RPC**
  - Supportent seulement l'intégration "procédurale" des applications
  - Ne fournit ni une abstraction des objets, ni la communication par messages asynchrones, ni l'invocation dynamique
  - Ne gère pas l'héritage des interfaces

- Windows **COM/DCOM** (méthode propriétaire)
  - Traditionnellement limité aux applications bureautiques
  - Ne gère pas la distribution des applications hétérogènes

Techniques existantes (2/3)

- **Java RMI** (adapté aux applications Java)
  - Peut être étendu à d'autres langages (ex: C ou C++) en utilisant des passerelles à travers Java Native Interface (JNI), ou RMI/IIOP
  - Bien adapté à toutes les applications Java à cause de son intégration couplée
  - Inclus dans Java Virtual Machine
  - Intégré dans les serveurs d'applications J2EE : IBM WebSphere, BEA WebLogic, Sun iPlanet, etc.

Techniques existantes (3/3)

- **Les Web services**
  - Composants logiciels accessibles via le web
  - WSDL (Web Service Description Language) : standard de description des services en XML
  - SOAP (Simple Object Access Protocol) : protocole étendant XML pour le codage des appels de web méthodes
  - UDDI (Universal Description, Discovery and Integration) : Référentiel de recherche de services web selon des mots-clefs de catégorisation des services
  - Plusieurs implantations IBM WSTK, Microsoft .NET

SOAP ?

- SOAP protocole XML permettant la communication entre composants logiciels et applications en s'appuyant sur des protocoles standards de type http, smtp, …
- SOAP est un protocole de transmission de messages
- SOAP est adapté à la communication entre applications
- SOAP est un format d'échange de messages
- SOAP est conçu pour fonctionner sur l'Internet
- SOAP est indépendant des plates-formes et des langages
- SOAP est basé sur XML
- SOAP est simple et extensible

OK, et CORBA alors?

- CORBA prend aussi en compte tous ces points
  - mais exige de compiler et distribuer des stubs clients pour chaque type de clients
  - pas toujours pratique pour un grand nombre de combinaisons de plates-formes et de langages ou lors de fourniture de services à des clients anonymes au travers d'Internet
- SOAP est un protocole basé sur XML
  - En conséquence, particulièrement prolixe
  - CORBA, au travers de IIOP (Internet Inter-ORB Protocol), le battra en performance car les opérations de conversion et de déconversion (marshalling et demarshalling) dans CORBA sont plus efficaces et il y a moins de données sur le réseau.

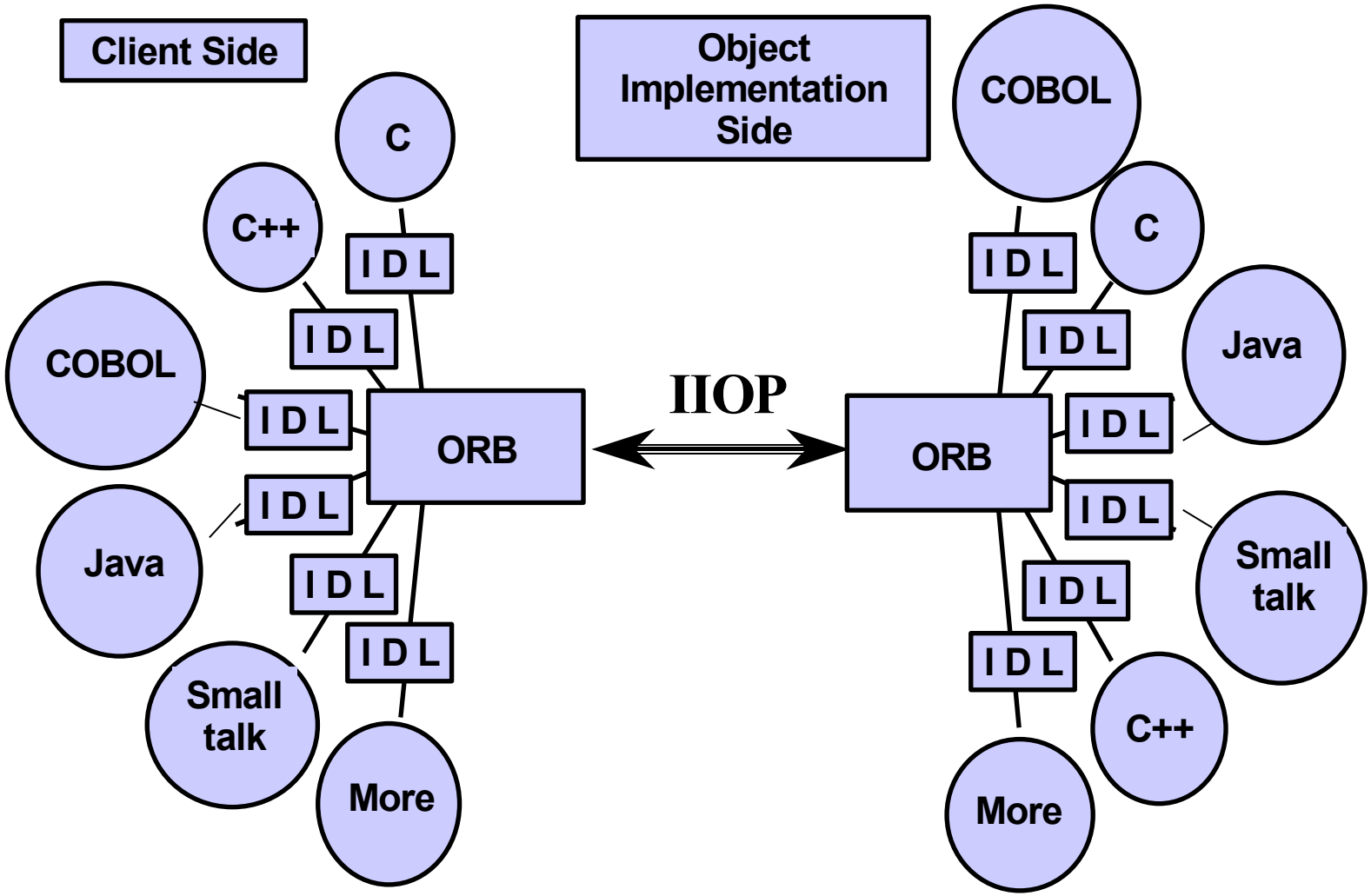
Source : *Journal du Net*

Motivations

- CORBA offre :
  - un plus haut niveau d'intégration que l'envoi traditionnel de données brutes et non typées (ex: paquets TCP)
  - une programmation distribuée selon une approche orientée objet
  - une base pour une collaboration de haut niveau des objets répartis
  - Une intégration des codes déjà existants
  - Une indépendance de plate-formes matérielles et logicielles (hétérogénéité et interopérabilité)

⇒ **facilite** la construction d'applications en environnement réparti

Intégration du code existant en multi-langage



Conclusion

Corba : permettre à des applications actives sur différentes machines d'un environnement **réparti hétérogène** de coopérer au travers d'objets **répartis**

- **Environnement réparti**
  - décentralisation, performance, fiabilité, disponibilité.
  - offre "middleware" sur le marché.
- **Environnement ouvert**
  - pour la portabilité et l'interopérabilité entre applications.
- **Approche objet**
  - pour l'extensibilité et l'évolutivité des applications.

=> CORBA

ENSIAS

I. Introduction à Corba

Contexte et Motivations

Principes fondamentaux de Corba

ENSIAS

L'Object Management Group (1)

- CORBA : une **norme**, un ensemble de spécifications et de recommandations rédigées par l'OMG (Object Management Group)
- OMG
  - Consortium créé en 1989, à but non lucratif
  - actuellement plus de 850 membres
    - Constructeurs : Sun, H.P., DEC, IBM, …
    - Editeurs d'environnements systèmes : Novell, Microsoft, …
    - Editeurs de logiciels : Borland, ObjectDesign, Iona,..
    - Produits et BD : Lotus, Oracle, Informix, O2, …
    - Industriels : Boeing, Alcatel, Thomson, …
    - Institutions universitaires

L'Object Management Group (2)

- Objectifs
  - Promouvoir la **technologie objet** dans le développement du logiciel
  - Etablir des **spécifications** pour l'intégration d'applications réparties hétérogènes
  - Offrir un **environnement logiciel** pour supporter :
    - La réutilisation et la portabilité des composants logiciels
    - L'hétérogénéité et l'interopérabilité entre les différents langages et les environnements informatiques (machines, OS)

L'Object Management Group (3)

- Proposition d'un cadre
  - basé sur 2 paradigmes
    - l'architecture répartie
    - les objets

⇒ systèmes répartis objet

  - **OMA** (Object Management Architecture) : une vision globale d'un système modulaire et réparti
  - **CORBA** (Common Object Request Broker Architecture)

Principes de modélisation en CORBA

- Séparation des interfaces et leur implantation
  - Les clients dépendent des interfaces, non des implantations
- Transparence de la localisation de l'objet
  - L'utilisation d'un service est orthogonale à sa localisation
- Transparence d'accès
  - On invoque des opérations sur des objets
- Interfaces typées
  - Les références des objets sont typées par les interfaces
- Support de l'héritage multiple
  - L'héritage étend, fait évoluer, et spécialise le comportement

L'Object Management Architecture

- OMA : Object Management Architecture
  - définit un modèle général pour les systèmes distribués orientés objet
  - classifie les objets intervenants dans l'application
  - spécifie les directives techniques générales qui doivent être suivies pour chaque composante du système
- OMA : une architecture composée
  - d'un ***modèle objet*** commun à toutes les composantes
  - d'un ***modèle de référence***

OMA : le modèle objet (1)

- Le Modèle Objet :
  - introduit les concepts de base caractérisant les objets d'un système et permettant d'avoir une sémantique commune à tous les objets
  - définit les objets pour représenter les entités ayant un état et un comportement
    - Le comportement d'un objet est implanté au moyen d'opérations qui lui sont appliquées
    - Chaque objet est une instance d'un type donné. Des relations d'héritage peuvent être définies entre les types

OMA : modèle objet (2)

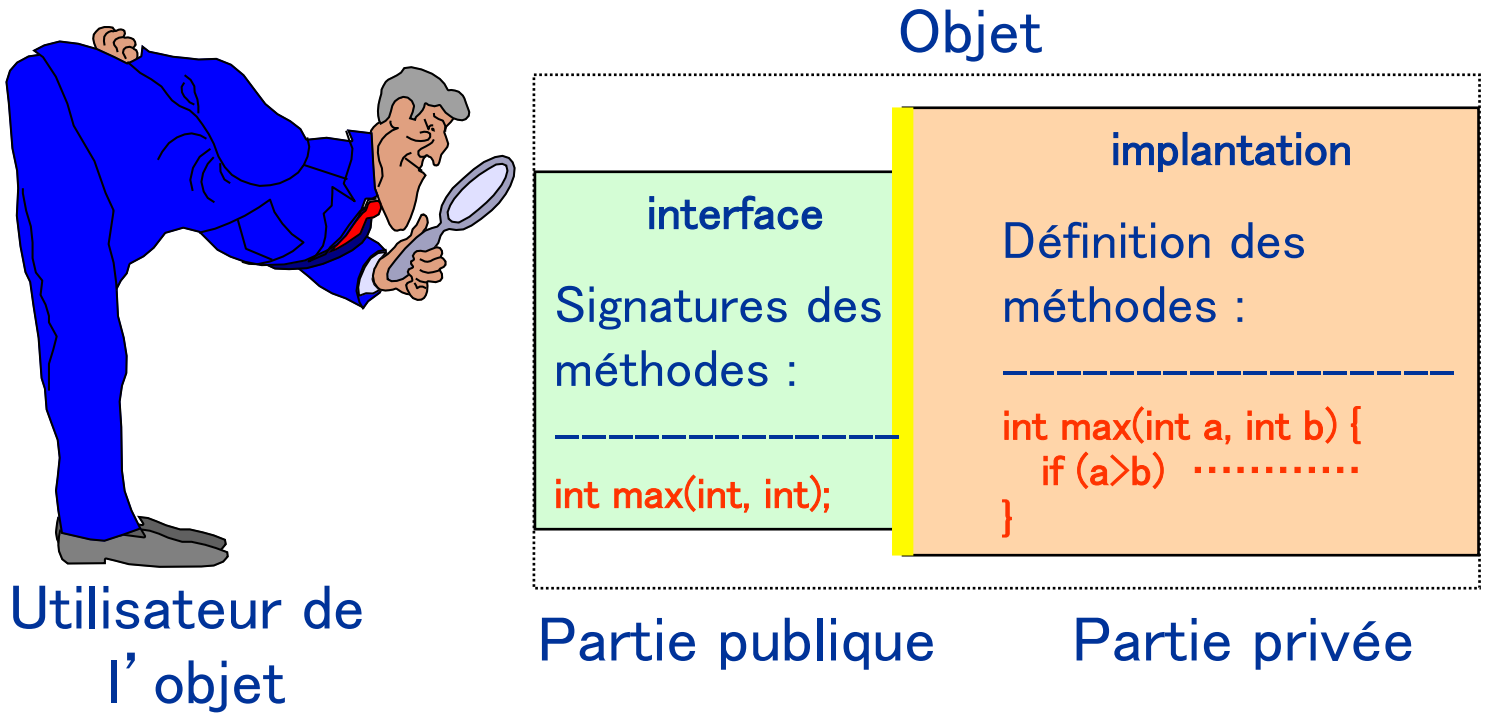
- Modèle orienté objet Client/Serveur
  - chaque application peut exporter certaines de ces fonctionnalités (services) sous la forme d'objets Corba
  - l'interaction entre applications est matérialisée par les invocations à distance des méthodes des objets ainsi définis
  - la notion Client/Serveur n'intervient que lors de l'utilisation d'un objet
    - l'application implantant l'objet est le serveur
    - l'application utilisant l'objet est le client
  - une application peut être à la fois cliente et serveur

OMA : principes du modèle objet

- Principes du Modèle Objet :
  - Isolation des demandeurs de service (client) et des fournisseurs (serveurs) par une interface bien définie
  - Côté Client
    - Création et identité d'un objet
    - Requêtes et opérations
    - Types et signatures
  - Côté serveur
    - Implantation (méthodes et activation)

L'interface d'objet ?

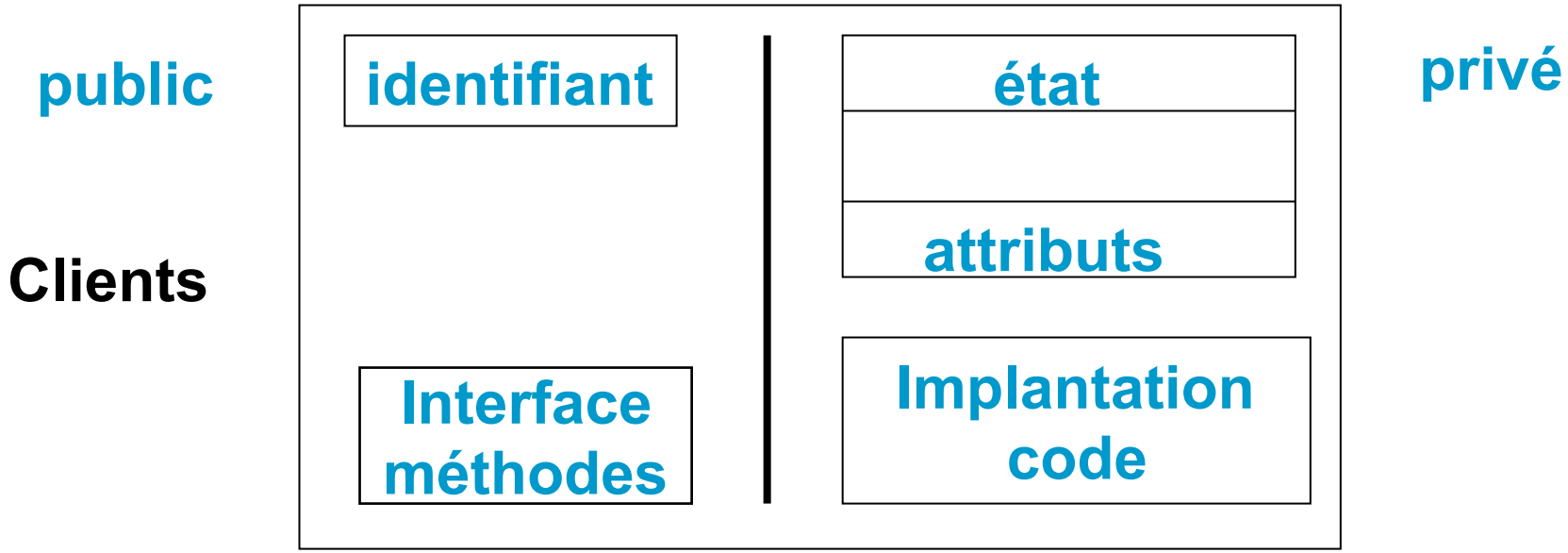
- Unité représentant un point d'accès à un objet



Reflète les services offerts par un objet

OMA : la sémantique de l'objet (1)

- Un objet est une entité :
  - identifiable
  - qui encapsule les données et les traitements associés
  - qui fournit un ou plusieurs services aux clients



OMA : la sémantique de l'objet (2)

- Les requêtes
  - il s'agit d'un événement auquel on associe une opération, un objet, un ou plusieurs paramètres, et un éventuel contexte de la demande
  - mécanisme d'invocation d'une opération ou d'accès à un attribut
  - elles correspondent à la *demande de service* d'un client. Un *résultat d'exécution* est retourné au client
  - Une *référence d'objet* est une valeur qui désigne un objet particulier

OMA : la sémantique de l'objet (3)

- Création et destruction d'objets
  - du point de vue du client, il n'y a pas de mécanisme spécial pour la création et la destruction d'objets
  - c'est le résultat d'exécution de requêtes
  - le résultat de la création d'objet est révélé au client sous forme de référence d'objet qui désigne l'objet nouvellement créé
- Types
  - Un type est une entité identifiable avec un prédicat associé défini sur des valeurs
  - Les types sont utilisés dans les signatures pour restreindre un paramètre ou un résultat

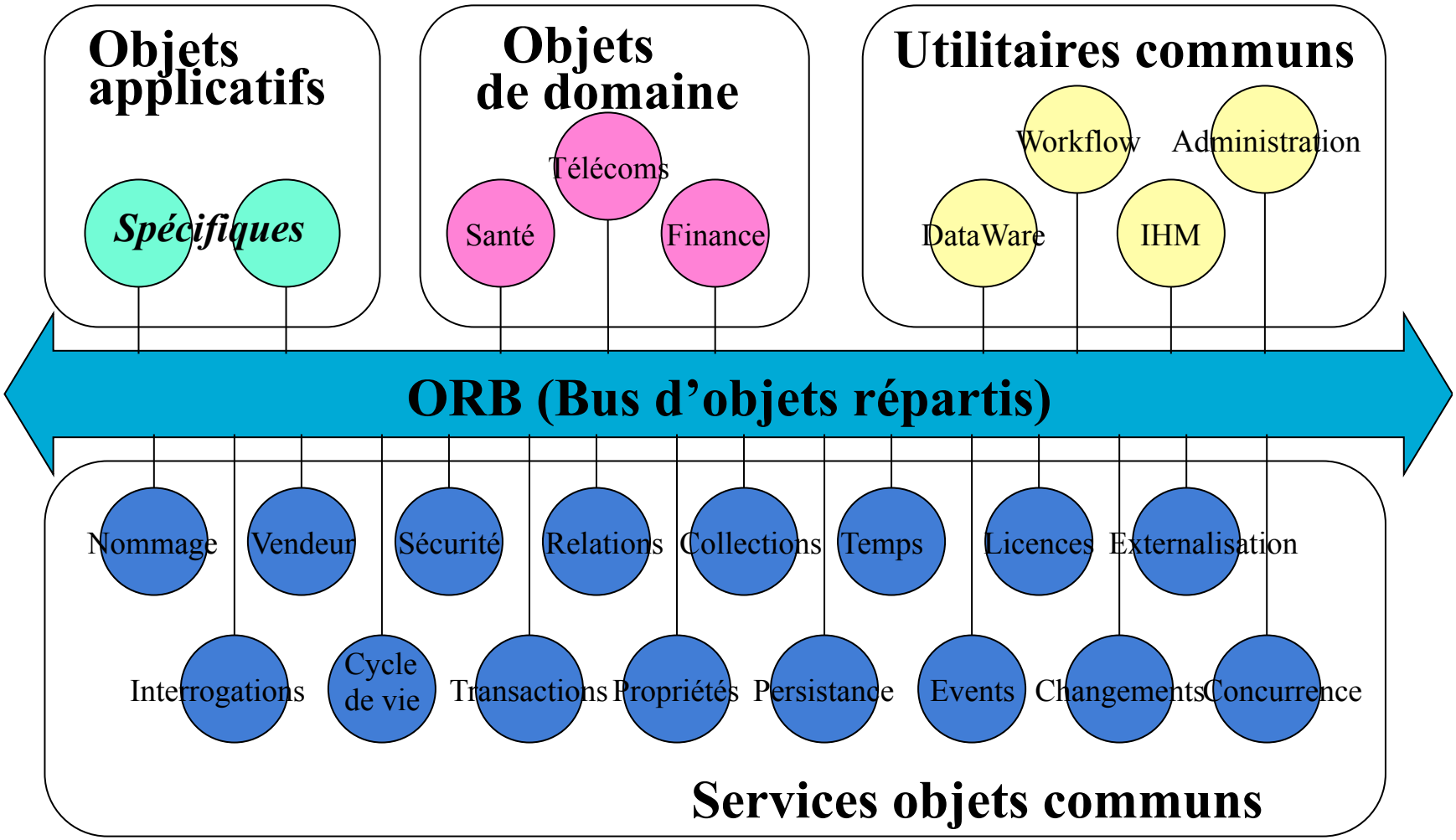
OMA : la sémantique de l'objet (4)

- Les interfaces
  - ce sont les descriptions des opérations possibles que le client peut demander à un objet
  - elles sont spécifiées en **IDL**. L'héritage d'interface fournit les mécanismes de composition permettant à un objet de supporter plusieurs interfaces
- Les opérations
  - Ce sont des entités identifiables (par un identificateur d'opération) qui définissent les services qui peuvent être demandés
  - Une opération a une signature qui décrit les paramètres, les résultats, les exceptions, le contexte et une indication sur la sémantique d'exécution

OMA : modèle de référence (1)

- Le modèle de référence identifie et classifie les différents types d'objets constituant un système réparti objet :
  - un bus à objets répartis : ORB (Object Request Broker)
    - transport des requêtes vers les objets
  - des services objet communs (Common Object Services)
    - Services orientés systèmes pour les développeurs
  - des utilitaires communs (Common Object Facilities)
    - Services orientés utilisateurs
  - des interfaces de domaines (Domain Objects) = objets métiers
    - « interopérabilité sémantique »
  - des interfaces d'applications (Application objects): objets applicatifs

OMA : modèle de référence (2)

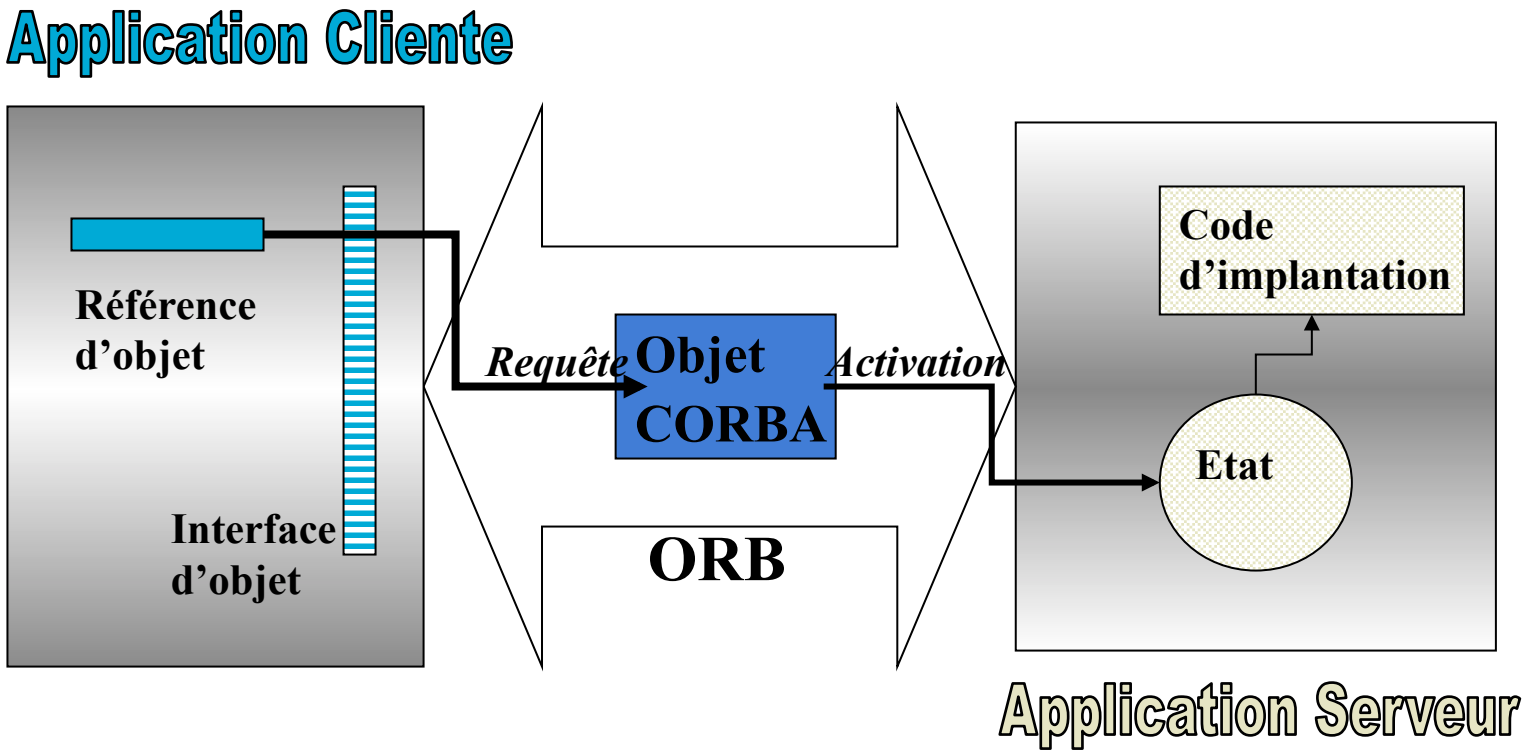


OMA : l'ORB (1)

- ORB : *bus logiciel* au cœur de l'architecture globale
  - composante la plus importante
  - sa fonction est de faciliter la communication entre les objets d'un système réparti
  - prend en charge le dialogue entre les objets serveurs et les différents clients qui s'y connectent
  - C'est la première composante permettant l'interopérabilité et la portabilité des applications

OMA : l'ORB (2)

- il permet aux clients d'émettre des requêtes aux objets pour obtenir un certain service
  - de façon indépendante de la distance, de la localisation, du langage et de la plate-forme de l'objet serveur
  - De la façon la plus transparente possible



Services objets communs (1)
(Object Services)

- Spécifications d'interfaces IDL
- Leurs fonctionnalités peuvent être étendues ou spécialisées par héritage
- Interfaces indépendantes des domaines d'application
- Objectif : étendre les fonctions de l'ORB
- Très utiles aux programmeurs
- Ne sont pas obligatoires pour :
  - avoir une mise en œuvre compatible CORBA
  - développer une application CORBA

Services objets communs (2)

- Fonctions systèmes accessibles sous forme d'objets
  - Outils destinés au développeur d'application répartie indépendants du domaine d'application
- Ils centralisent des services répétitifs que l'on retrouve dans la plupart des développements
- Exemples :
  - une grande partie des applications réparties ont besoin de règles de sécurité et de protection pour les données transmises sur le réseau
  - l'OMG a défini un service Sécurité
- de nombreux services génériques ont été spécifiés

ENSIAS

Services objets communs (3)

- Services de recherche d'objets
  - Service Nommage (Naming service)
    - pour retrouver un objet par un nom
    - service de « pages blanches »
  - Service Vendeur (Trader)
    - pour retrouver un objet par des propriétés
    - service de « pages jaunes »

Services objets communs (4)

- Services de communications asynchrones
  - Events, Notification, Messaging
- Services de sûreté de fonctionnement
  - Security, Transactions, Concurrence
- Services concernant la vie des objets
  - Life Cycle, Property, Relationship, Externalization, Persistent Object, Query, Collection, Versionning, Time, Licencing

Utilitaires communs (Common Facilities)

- Spécifications d'interfaces IDL
  - Leurs fonctionnalités peuvent être étendues ou spécialisées par héritage
- Ensemble de services de plus haut niveau fournissant des fonctionnalités utiles dans de nombreuses applications
- Indépendants du domaine d'application
- **Exemples** :
  - gestionnaire d'impression
  - outils de gestion documentaire
  - module de messagerie, …
  - IHM, administration, workflow, etc.

ENSIAS

Interfaces de domaines (Domain Objects) (1)

- Objets utilitaires destinés à un secteur d'activité donné (ils constituent des spécialisations métier)
- Leur **objectif** : assurer l'interopérabilité sémantique entre les SI d'entreprises d'un même métier (BOF : Business Object Frameworks)
- **Exemples** :
  - La santé, la finance, Telecom, domaine médical, le commerce électronique
  - …

  ⇒ si création d'un objet de gestion des cartes bancaires: il sera utilisé par l'ensemble des entreprises du secteur financier

Les interfaces de domaines (2)

- Standardisés par l'OMG, leurs fonctionnalités peuvent être étendues ou spécialisées par héritage
- Spécifiques à un domaine d'application
  - Telecom : AVS (Audio/Video Streams), Telecom Log Service, CORBA/TMN
  - Finance : Currency Specification, General Ledger, Party Management
  - Médical : Person ID Specification, Lexicon Query, Resource Access Decision Facility, Clinical Observation Access Service
  - ...

ENSIAS

Interfaces d'application (Application Objets)

- Ce sont des interfaces développées en IDL spécifiquement pour une application répartie donnée
- Elles ne sont pas standardisées par l'OMG puisqu'elles sont spécifiques aux applications
- Possibilité de standardisation pour des objets émergents

Conclusion

- OMG: consortium qui définit les standards CORBA, OMA, CORBAservices et CORBAfacilities.
- Spécifications consolidées pour l'ORB, encore peu stables pour les CORBAservices et en cours pour les CORBAfacilities.
- Produits disponibles chez plusieurs éditeurs de logiciels, constructeurs informatiques et dans le domaine public.

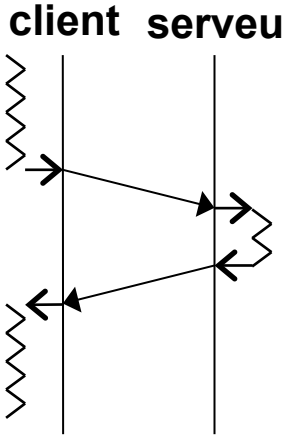
II. Modèles, Composants et Architecture

L'architecture CORBA

- CORBA : tout système réparti s'appuyant sur l'OMA, l'architecture définie par l'OMG
- Son rôle : implanter les fonctionnalités de l'ORB
  - CORBA détaille les interfaces et les caractéristiques de l 'ORB
- Caractéristiques :
  - Transparence totale des invocations
    - un utilisateur a toujours l'impression de faire un appel local
  - Activation automatique et transparente des objets

Modèle d'interaction C/S

Modèle **d'invocation** client/serveur



client : composant qui fait appel à un service

serveur : composant qui fournit un service

=> définition de l'interface décrivant ces services

Le modèle d'invocation possède les caractéristiques suivantes :

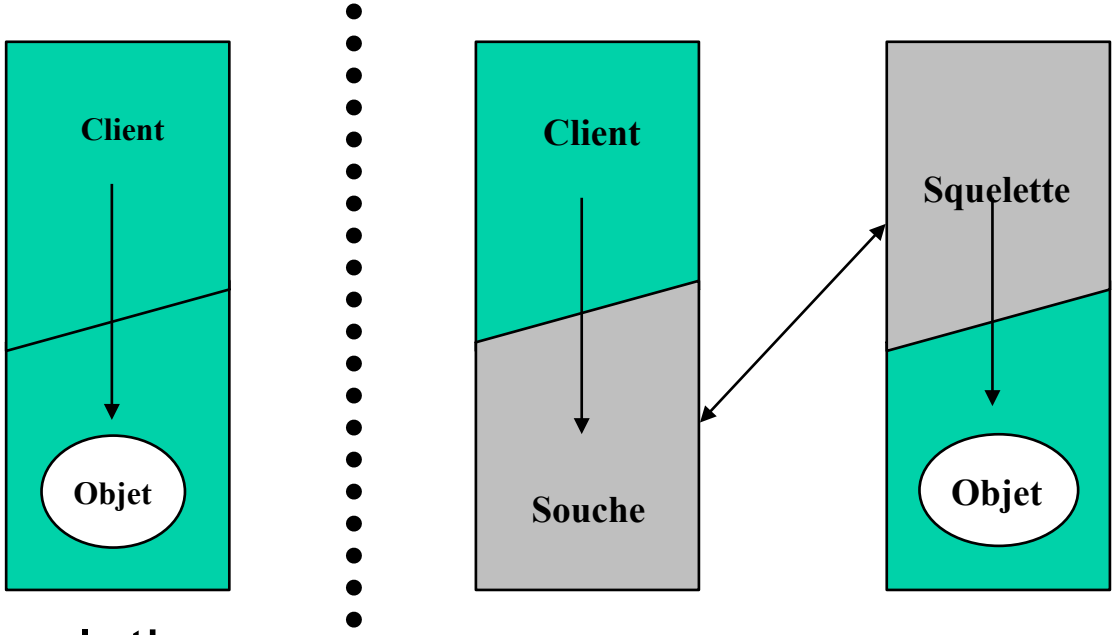
- présence simultanée du client et du serveur,
- appel bloquant pour le client.

ENSIAS

Modèle « Invocation Statique »

Communication entre C/S s'appuie sur des **adaptateurs réseau** dérivés de la définition de l'interface.

Rôle des adaptateurs réseau



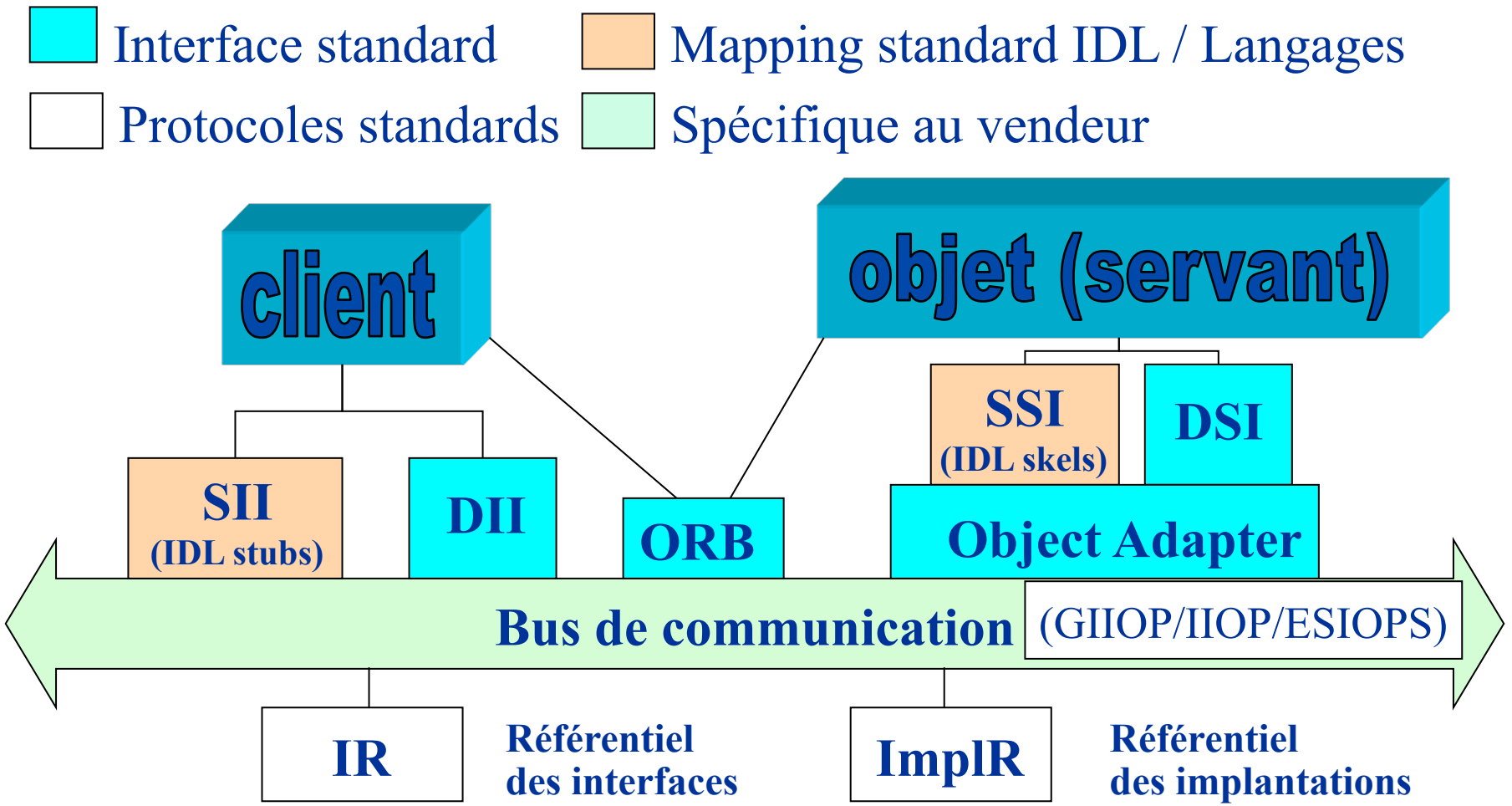
solution centralisée

solution répartie

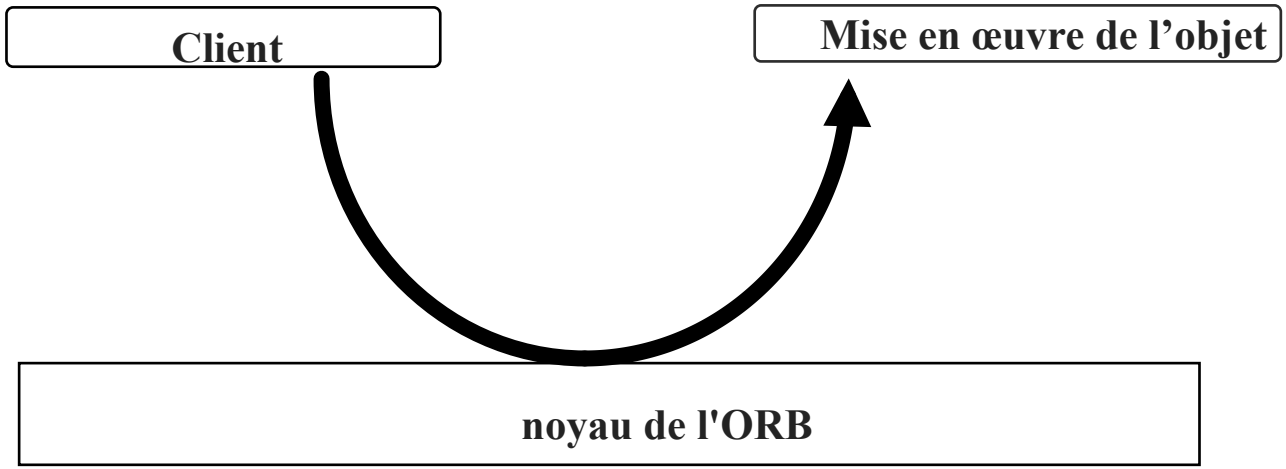
Composants de l'architecture CORBA

- Composants de CORBA :
  - ORB : courtier de requêtes
  - BOA (et POA) : l'adaptateur d'objets
  - SII/SSI : mécanisme d'interfaçage statique
  - DII/DSI : mécanisme d'interfaçage dynamique
  - IOR : identification des objets
  - IFR : un référentiel d'interfaces
  - ImpIR : un référentiel d'implantation

L'architecture CORBA



L'ORB



Noyau de l'ORB : assure le transport des requêtes entre objets de la façon la plus transparente possible.

Les composants de l'ORB

- ORB : Object Request Broker core (négociateur)
  - localisation des objets
  - noyau de transport des requêtes (protocoles GIOP, IIOP) vers un objet
  - transport des résultats vers le client en milieu hétérogène

Les composants de l'ORB

IDL : Interface Definition Language

- langage de définition des interfaces entre clients et serveurs d'objets.
- résout le problème de la portabilité.

IIOP : Internet Inter-ORB Protocol

- protocole de communication de type requête-réponse.
- résout le problème de l'interopérabilité.
  - GIOP (Global Inter-Orb Protocol) ➔ IIOP (Internet IOP) au dessus de TCP/IP

Les composants de l'ORB

- Interface Definition Language
  - langage de spécification d'interfaces (orienté objet)
  - fortement typé
  - indépendant de tout langage de programmation
  - héritage multiple
  - traduction automatique des descriptions dans divers langages de programmation

  ⇒ langage de spécification et non d'implantation

Les composants de l'ORB

- Exemple

```
module UnService {
  typedef unsigned Long EntierPositif;
  typedef sequence<Positif> desEntiersPositifs;

interface Premier {
  boolean est_premier(in EntierPositif nombre) ;
  desEntiersPositifs nombres_premiers(in
  EntierPositif nombre) ;
  } ;
} ;
```

Les composants de l'ORB

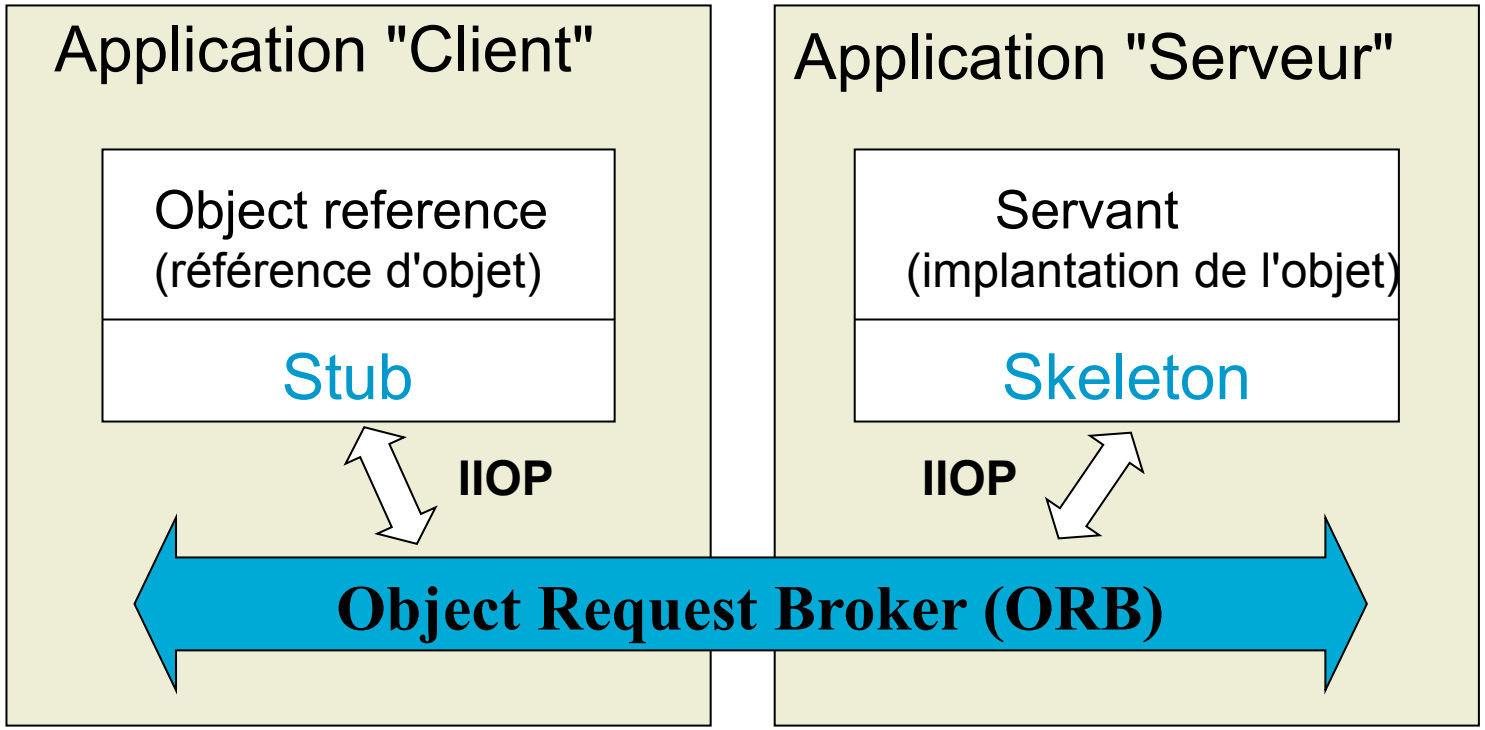
- L’interface du bus ORB : fournit les primitives de base
  - l’initialisation,
  - le paramétrage de l’environnement CORBA,
  - la gestion des références,
  - gestion de la mémoire (allocation, libération, …)
- Exemple :
  - primitive pour convertir une référence d’objet en chaîne de caractères (et une autre pour l’opération inverse).
- Primitives utilisées aussi bien par les applications clientes que par les objets serveurs.

Les composants de l'ORB

```
Module CORBA {
   ORB ORB_init(in string argv, in ORBid orb);
   pseudo interface ORB {
      string object_to_string (in Object obj);
      Object string_to_object (in string str);
      BOA BOA_init(in string argv,in BOAid boa);
      Object resolve_initial_refs(in ObjectId ident);
        create_{NVList, NamedValue, Context);
   };
…
};
```

CORBA : les composants (1)

- Les modules stub (souche) pour la partie cliente et skeleton (squelette) au niveau du serveur :
  - prennent en charge les communications entre client et serveur



Le dialogue CORBA : client, ORB et serveur

CORBA : les composants (2)

- Un stub est un représentant local d'un objet distant :
  - Il connaît la localisation de l'objet
  - Il réalise l'empaquetage et le dépaquetage des invocations et des résultats
  - Il fait appel au bus ORB pour réaliser la communication
  - le code des stubs est généré à partir de la description de l'interface IDL de l'objet

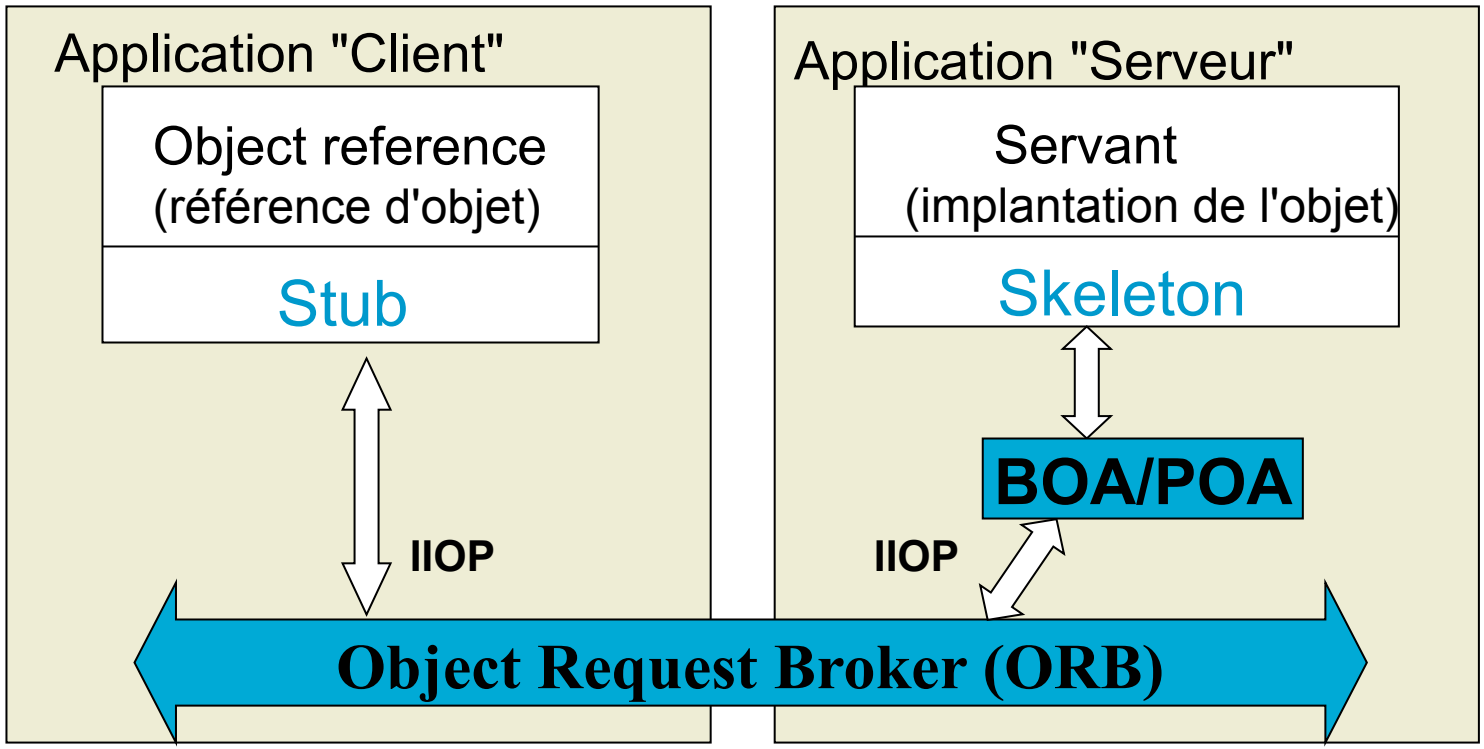
CORBA : les composants (3)

- Un skeleton fournit les interfaces statiques de chaque service exporté par le serveur
  - il est créé en utilisant un compilateur IDL (comme pour le stub)
  - il traite les requêtes provenant des stubs ou DII
  - l'existence d'un squelette n'implique pas l'existence du stub client correspondant (utilisation de l'interface dynamique pour le client)

CORBA : les composants (4)

- L'adaptateur d'objets (Object Adapter) : support d'exécution
  - permet l'adaptation à la nature des objets implantés
  - gère les différents objets qui se situent à l'intérieur de l'application serveur
    - transmission des demandes émises par les applications clientes vers un objet particulier
    - activation/désactivation des objets
    - …
  - actuellement, 2 types d'adaptateurs existent :
    - **BOA (Basic Object Adapter)** : issu des anciennes versions de la norme
    - **POA (Portable Object Adapter) :** défini dans CORBA 2.2

CORBA : les composants (5)



Positionnement de l'adaptateur d'objets dans CORBA

CORBA : les composants (6)

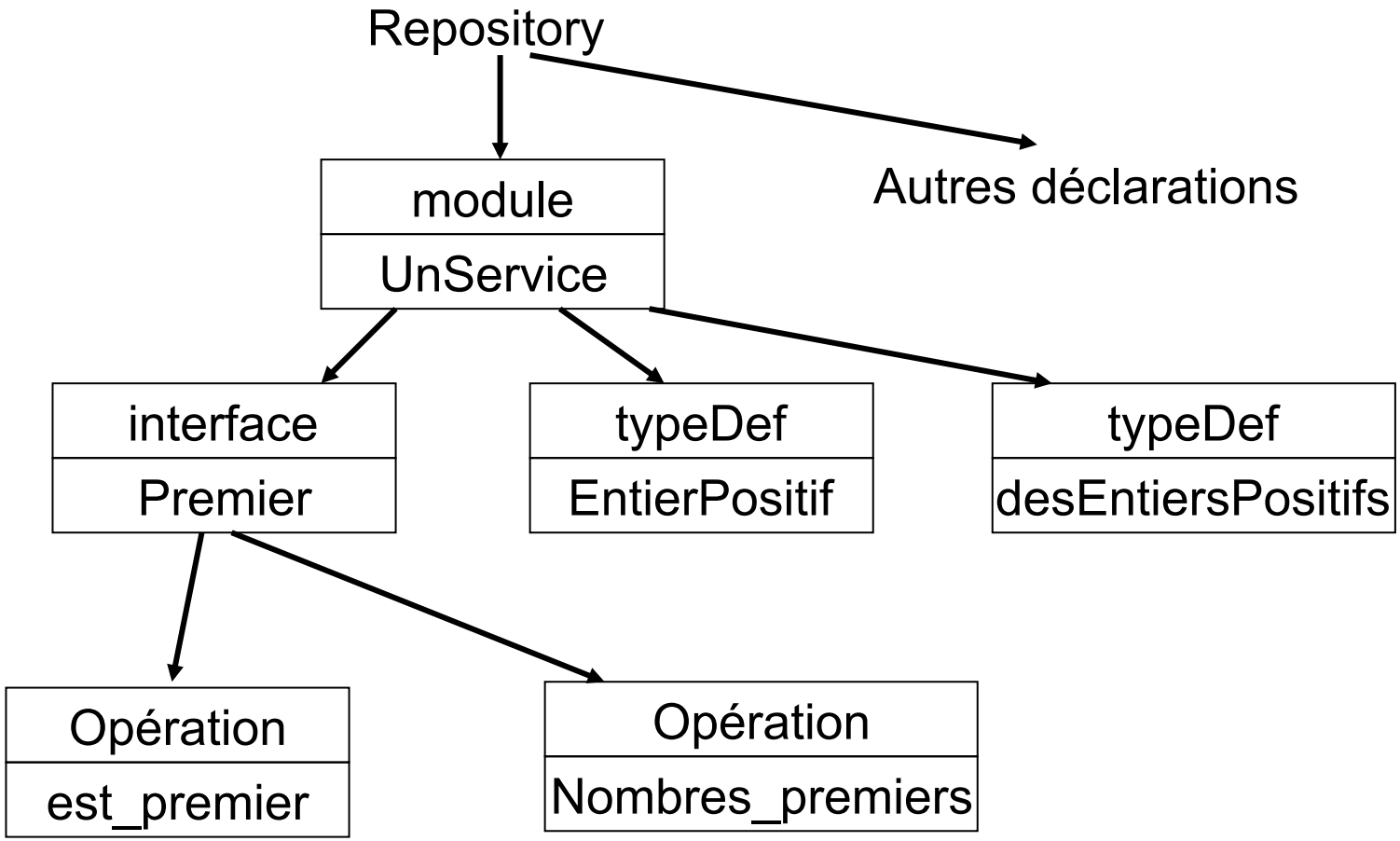
```
Module CORBA {

   pseudo interface BOA {
      void dispose(in Object obj);
      void change_implementation(in Object obj, in
                                 ImplementationDef impl);
      void impl_is_ready(in ImplementationDef impl);
      void deactive_impl(in ImplementationDef impl);
      void obj_is_ready(in Object obj, in

        ImplementationDef impl);
      void deactibe_obj (in Object obj);
   };
};
```

CORBA : les composants (7)

- IR : Interface Repository
  - stocke toutes les définitions IDL des interfaces des objets du bus, des méthodes qu'ils supportent et des paramètres qu'ils nécessitent
  - les informations contenues dans l'IR : métadonnées
    - accès offert aux applications qui peuvent exploiter dynamiquement (càd durant l 'exécution) les interfaces des objets CORBA
- Graphe d'objets « concepts » d'IDL
  - ModuleDef, InterfaceDef, OperationDef, AttributeDef, TypeDef, …
  - Par navigation dans ce graphe ou à partir d'une référence d'objet, on peut retrouver le contenu d'une interface, la signature d'une opération, ...

CORBA : les composants (8)

Exemple d'un graphe d'objets dans IR



ENSIAS

CORBA : les composants (9)

- ImplR : le référentiel d'implantations
  - contient les informations nécessaires décrivant l'implantation des objets
    - le nom des exécutables contenant le code des objets, la politique d'activation de ces exécutables, les droits d 'accès aux serveurs et à leurs objets
  - permet à l'ORB de localiser et d'activer les différentes implantations des objets de l'IR
  - interrogation de la base de données
    - implantation de la méthode *get_implementation* de chaque objet

Les mécanismes dynamiques (1)

- Invocation dynamique (DII)
  - permet à une application cliente d'invoquer dynamiquement les opérations des objets sans utiliser les stubs IDL prégénérés
  - l 'application cliente découvre les fonctionnalités des objets en consultant le IR :
    - elle peut alors construire dynamiquement des requêtes via la souche générique offerte par le DII
    - le client spécifie l'objet qui est invoqué, l'opération à exécuter, l'ensemble des paramètres pour l'opération

Les mécanismes dynamiques (2)

- L'interface dynamique DSI
  - est l'équivalent du DII
  - offre un squelette générique pour les applications serveurs
  - fournit un mécanisme de liaison dynamique pour les serveurs ayant besoin de gérer des appels de méthodes pour des objets n'ayant pas de squelettes compilés (skeleton)
  - regarde les valeurs des paramètres dans le message entrant pour voir à qui est destiné le message : détermination de l'objet et de la méthode cibles

III. Cycle de développement Corba

Schéma général de développement

Exemple d'application bancaire

Les produits

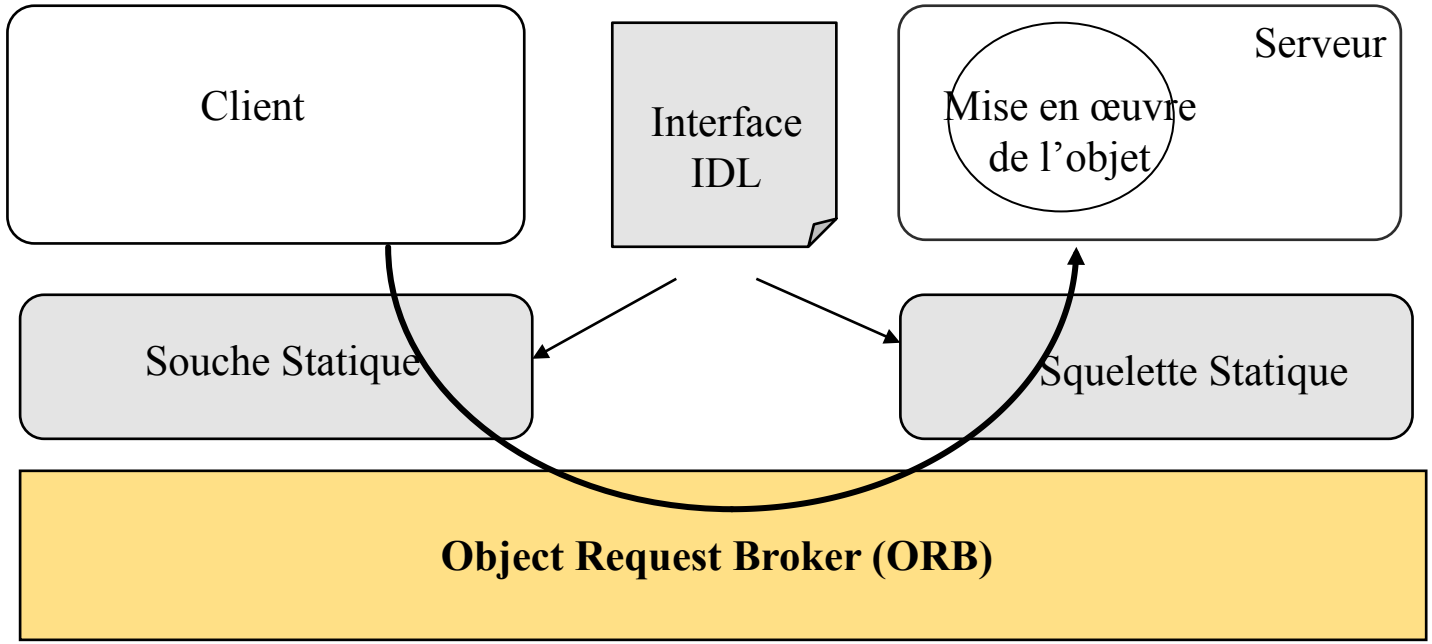
Schéma général de développement

- Ecrire la description de l'interface (langage IDL : Interface Definition Language) et la compiler.

- Ecrire le code du serveur (en particulier, l'objet qui met en œuvre l'interface en utilisant un langage de programmation C, C++, Java...) et le compiler.

- Ecrire le code du client (en utilisant un langage identique ou ≠ de celui utilisé pour le serveur) et le compiler.

1ère vue du schéma de développement



Les adaptateurs réseau, souche et squelette font appel aux fonctions de communication offertes par l'ORB

Définition de l'interface

- L'interface possède un **nom** et contient la définition des **types** (notamment des exceptions), des attributs et des opérations.

- Les opérations sont décrites avec leurs paramètres. Chaque paramètre est précédé par son type et son sens de passage.

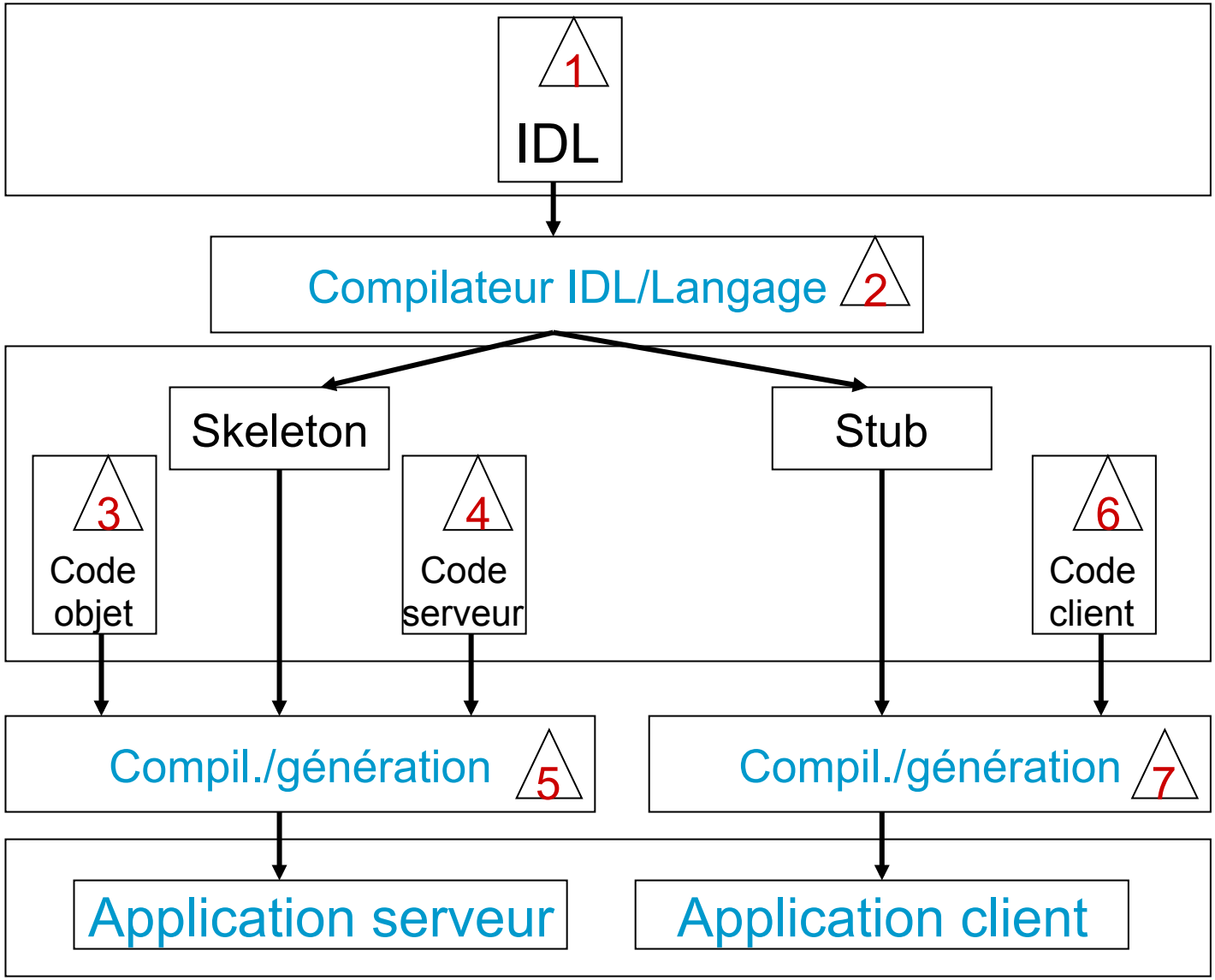
Etapes du cycle de développement CORBA (1)

1. Ecriture de l'interface de l'objet en IDL
2. Compilation de l'IDL
   - Génération des modules stub et skeleton
3. Implantation de l'objet
   - Dérivation d'une classe depuis le skeleton généré
4. Rédaction ou génération du code de l'application serveur
   - Ce code sert à lancer l'objet et à le mettre à disposition des différentes applications clientes

Etapes du cycle de développement CORBA (2)

5. Compilation du serveur
   - Génération de l'exécutable de l'application serveur avec liaison au module ORB
6. Réalisation de l'application cliente
   - Cette application se connecte à l'objet distant pour lui demander l'exécution de méthodes
7. Compilation du client
   - Tout comme le serveur, cette application doit inclure l'ORB

Etapes du cycle de développement CORBA (3)



Réalisation côté serveur

- Réaliser l'objet qui met en œuvre l'interface
- Réaliser le code de l'application serveur en 4 étapes :

a- Initialiser l'environnement d'exécution du serveur
  - Initialiser l'ORB et l'adaptateur d'objets (POA)

b- Rendre l'objet publique
  - Exporter sa référence d'objet.

c- Attendre les requêtes des clients
  - Activer l'adaptateur d'objets et l'ORB.

d- Arrêter l'environnement d'exécution du serveur
  - Arrêter l'ORB et l'adaptateur d'objets.

Réalisation côté client

- Réaliser le code de l'application cliente en 4 étapes :

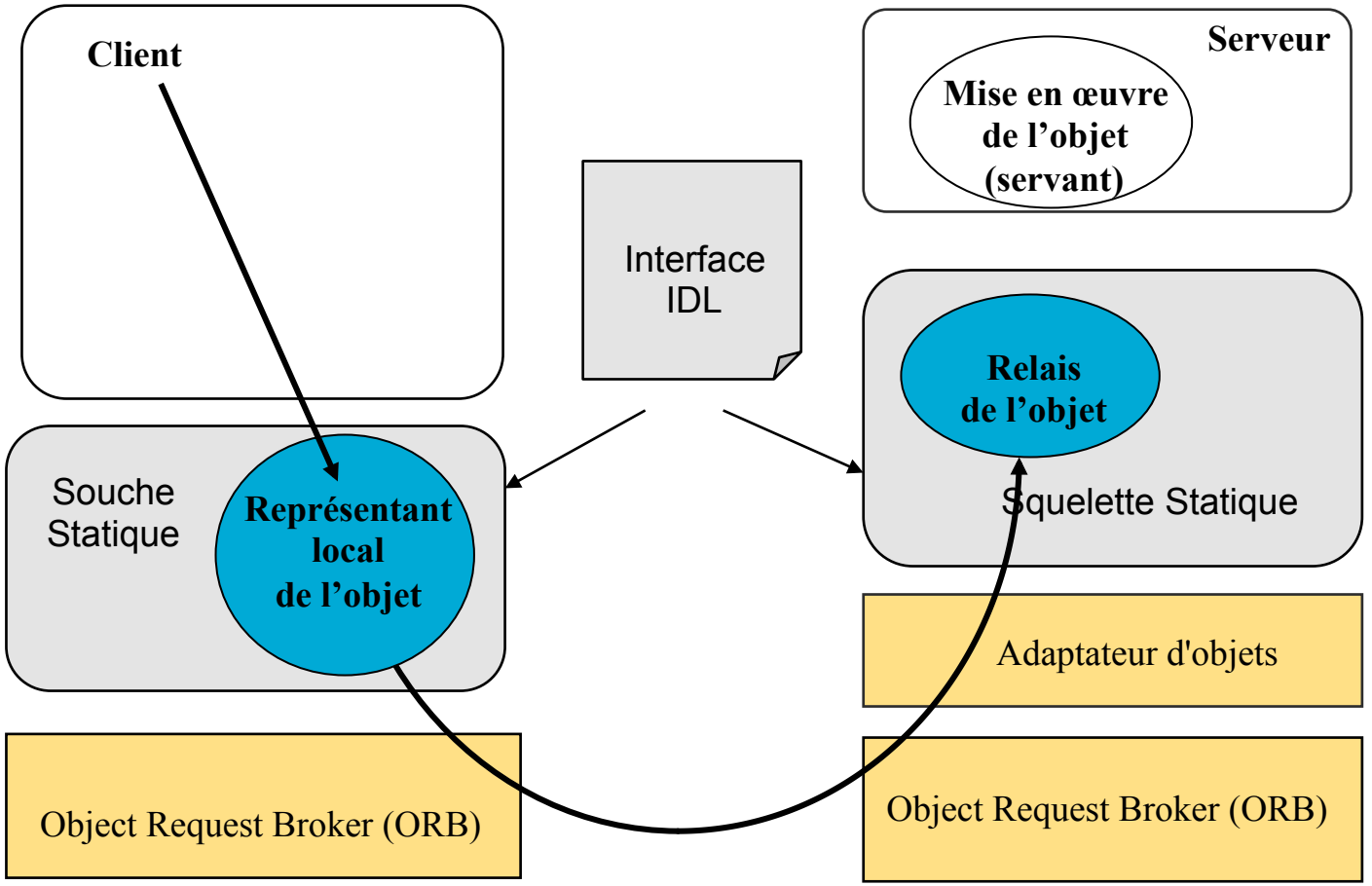
a- Initialiser l'environnement d'exécution du client : initialiser l'ORB.

b- Se lier à l'objet de mise en œuvre distant : Récupérer la référence de l'objet à invoquer et la convertir en une référence vers 1 objet local représentant de l'objet distant.

c- Faire appel aux opérations : Appeler les fonctions de l'interface par des appels locaux sur le représentant local de l'objet distant.

d- Arrêter l'environnement d'exécution du client : Arrêter l'ORB.

2ème vue du schéma de développement



III. Cycle de développement Corba

Schéma général de développement

Exemple d'application bancaire

Les produits

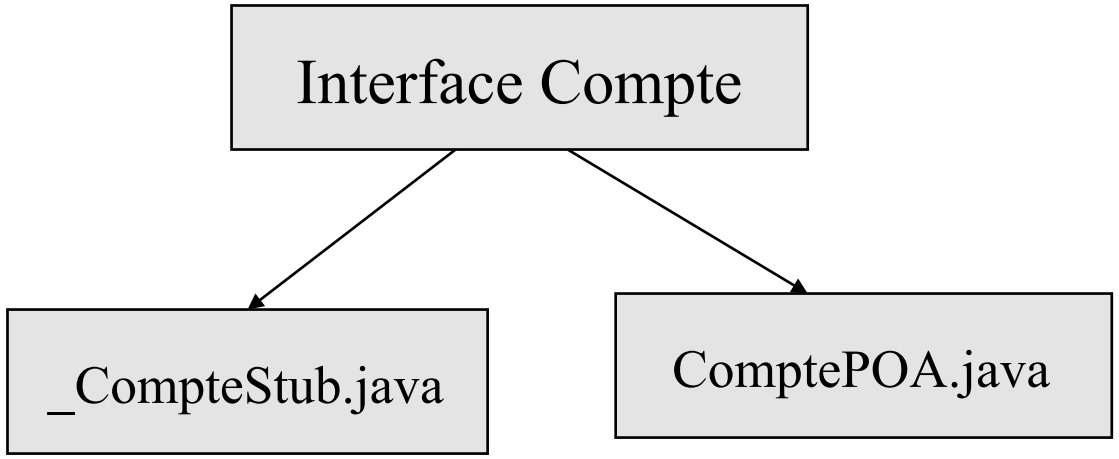
Exemple : application bancaire (ORBACUS)

=> réaliser une opération de crédit sur un compte bancaire

1. La définition IDL pour créditer un compte

```
interface Compte
   { void crediter(in float somme_credit);   } ;
```

2. La compilation IDL



Classe « représentant local » Classe « relais »

Serveur bancaire (1)

3. Réaliser l'objet de mise en œuvre (CompteImpl.java)

```
// réalisation de la classe de mise en œuvre
❶public class CompteImpl  extends ComptePOA {

   private float solde;
   CompteImpl (float montant_init) {
         solde = montant_init;          }

❷ public void crediter (float montant_credit) {
         solde =solde + montant_credit;          }
}
```

Serveur bancaire (2)

4. Réaliser l'application serveur (serveur_banque.java)

a : Initialiser l'environnement d'exécution du serveur

```
public static void main(String args[]) {
org.omg.CORBA.ORB orb = null;
    try {
     // Initialisation de l'ORB
❶ orb = ORB.init(args, null);
    // Initialisation du POA
❷ org.omg.PortableServer.POA rootPOA =
    org.omg.PortableServer.POAHelper.narrow(
          orb.resolve_initial_references("RootPOA"));
    // Initialisation du gerant de POA
❸ org.omg.PortableServer.POAManager gerant = rootPOA.the_POAManager();
    } catch(SystemException ex) {
    …
```

Serveur bancaire (3)

b : Rendre l'objet publique

```
...
// creation de l'objet de mise en œuvre (servant)
❶ CompteImpl unCompteImpl = new CompteImpl(500);
 // Creation de la reference CORBA
❷ Compte unCompte = unCompteImpl._this(orb);
 // Copie de la reference dans un fichier

❸ String ref = orb.object_to_string(unCompte);
  String refFile = "Compte.ref";
  FileOutputStream file = new FileOutputStream(refFile);
  PrintWriter out = new PrintWriter(file);
  out.println(ref);
  file.close();
...
```

Serveur bancaire (4)

c: Attendre les requêtes des clients

```
…
try {
      // Activation du gerant de POA
 ❶ gerant.activate();
    // Activation de l'ORB
 ❷ orb.run();
    } catch(SystemException ex) {
        System.err.println(" -- Erreur CORBA -- \n" + ex.getMessage());
        ex.printStackTrace();
        return;
      }
...
```

Serveur bancaire (5)

d : Arrêter l'environnement d'exécution du serveur

```
…
try {
   // Terminaison de l'ORB
❶  orb.shutdown(true);
} catch(SystemException ex) {
 …
```

5. Compiler le serveur
   - compilation Java classique qui consiste en la génération de codes intermédiaires " .class ".

Client bancaire (1)

6. Réaliser l'application cliente (client_banque.java)

a . Initialiser l'environnement d'exécution du client

```
public static void main(String args[])  {
    org.omg.CORBA.ORB orb = null;
    // initialisation de l 'ORB
     orb = ORB.init(args, null);
    }
    catch(SystemException ex)
    {
       System.err.println(" -- Erreur CORBA -- \n" + ex.getMessage());
       ex.printStackTrace();
       return null;
    }
```

Client bancaire (2)

b. Se lier à l'objet distant

```
...
    String refFile = "Compte.ref";
    FileInputStream file = new FileInputStream(refFile);
    BufferedReader in = new BufferedReader(new InputStreamReader(file));
    ref = in.readLine();
    file.close();
❶   org.omg.CORBA.Object obj = orb.string_to_object(ref);
❷   Compte compte = CompteHelper.narrow(obj);
...
```

Client bancaire (3)

c . Faire appel aux opérations

```
...
    // Appel vers les opérations distantes
    try {
        System.out.println( "  Indiquer le montant du credit : ");
        montant_credit=lire_float();
        compte.crediter(montant_credit);
    } catch(SystemException ex) {
        System.err.println(" -- Erreur CORBA -- \n" + ex.getMessage());
        ex.printStackTrace();
        return;
    }
...
```

Client bancaire (4)

d . Arrêter l'environnement d'exécution du client

```
…
try {
   // Terminaison de l'ORB
   orb.shutdown(true);
} catch(SystemException ex) {
  …
```

7 . Compiler le client

Compilation Java

- Liste des fichiers .java :
  - développés : client_banque.java, serveur_banque.java, CompteImpl.java.
  - générés automatiquement : _CompteStub.java, Compte.java, CompteHelper.java, CompteHolder.java, CompteOperations.java, ComptePOA.java.

- Génération des fichiers .class:
  - À partir des fichiers .java développés et générés automatiquement

(ex) `javac -d classes *.java`

III. Cycle de développement Corba

Schéma général de développement
Exemple d'application bancaire
Les produits

Les produits : Offre CORBA

- ORBs d'éditeurs logiciels
  - Orbix de IONA Technologies,
  - Visibroker de Borland,
  - Object Broker de BEA Systems
  - ORBacus de Object Oriented Concepts
- ORBs de constructeurs
  - Component Broker de IBM,
  - ORB Plus de HP.
- ORBs du domaine public
  - …

Critères d'évaluation

- Projections vers des langages
  - C, C++, Smalltalk, Cobol, Ada, Java, Lisp
- Protocole d'interopérabilité IIOP
- Interfaces dynamiques DII et DSI
- Intégration COM
- Services et utilitaires CORBA
- Utilitaires de développement, d'administration, ...

ENSIAS

Conclusion

- Une complète interopérabilité
  - entre objets et applications basés sur CORBA : grâce au protocole IIOP (Internet Inter-ORB Protocol)
- L'intégration aux systèmes existants
  - l 'architecture de CORBA est ouverte
- Flexibilité du développement
  - grâce à la dissociation des parties *définition* et *implantation* d 'un objet : le client ne voit que l'interface
- Le choix du fournisseur
  - choix de l'ORB (il en existe plusieurs dizaines sur le marché, respectant la norme CORBA à des degrés divers)

ENSIAS

IV. Mise en œuvre des objets Corba

Référence d'objet

Mise en œuvre côté client/ côté serveur

Le service de noms

ENSIAS

Référence d'objet Corba

- Chaque objet distant est connu du monde extérieur au moyen de sa référence d'objet
- Une référence d'objet peut être convertie en chaîne de caractères pour permettre son échange
- Le format de la référence dépend du protocole de communication

L'IOR

- IOR : Interoperable Object Reference
  - séquence unique d'octets identifiant l'objet chez le serveur
  - Exemple d'IOR « stringifiée » (sous forme d'une chaîne de caractères) :

IOR:000000000000001049444c3a5472697669616c3a312e3000000000100000000000000007c0001
02000000000d3135322e38312e342e313130000004800000025abacab3131303033383632313336
005f526f6f74504f410000cafebabe3bd5b87800000000000000000000010000001000000 2c0000000
0000100010000000400010020000101090001010005010001000101090000000200010100050100 01

Composition d'une référence d'Objet

IIOP :

- version : numéro de version du protocole IIOP
- AdHost : adresse du site (nom DNS ou adresse IP)
- N°Port : numéro du port (TCP)
- ObjectKey : clé unique identifiant l'objet dans le serveur

- La référence d'objet contient l'info. nécessaire à l'aiguillage de la requête du client au serveur

Caractéristiques des références d'objet

- La référence d'objet : information nécessaire pour désigner un objet avec un ORB
  - info. de localisation des objets dépendante du protocole de communication utilisé
    - ex. dans IIOP : nom de la machine et le port TCP/IP du serveur où est localisé l'objet
  - deux implantations d'ORB peuvent différer par leur choix de la représentation des références d'objets

Echanger une référence d'objet

- L'ORB est défini par l'OMG comme un pseudo-objet avec une interface IDL,
  - Cette interface est convertie, en Java, en une classe qui permet l'accès aux services de l 'ORB :

    org.omg.CORBA.ORB
- Elle permet la conversion de la référence d'objet en chaîne de caractères pour permettre son échange.
- Les références d'objet:
  - fournies par les objets serveurs
  - clés d'accès pour les clients
- Mécanisme de liaison : 2 solutions pour obtenir une référence d'objet
  - par sa mise à disposition par l'objet serveur,
  - au moyen du service de désignation (annuaire)
    - nécessite la connaissance de sa référence d'objet !

Les références initiales

- Références initiales : l'ensemble des références constituant les points de départ pour une application CORBA et accessible à l'initialisation de l'ORB
  - ex : réf. vers l'annuaire / rootPOA/…
- L'interface de l'ORB fournit 2 fonctions qui permettent d'obtenir les références initiales :
  - `ObjectIdList list_initial_services() :`
    - retourne la liste des id des services connus par l'ORB comme réf. initiales
  - `resolve_initial_references( in ObjectId Ident) :`
    - retourne la réf. d'obj CORBA associée au service identifié

IV. Mise en œuvre des objets Corba

Référence d'objet

Mise en œuvre côté client/ côté serveur

Le service de noms

ENSIAS

Mise en œuvre de l'objet : côté client

- Pour échanger des objets de tout type, l'ORB échange des objets **génériques** plutôt que les objets de l'utilisateur
- Tous les objets CORBA (donc les interfaces IDL) héritent dans la projection Java, de la classe **org.omg.CORBA.Object**
- Au moyen de cet héritage, tout objet défini par l'utilisateur peut être utilisé comme objet générique.
- Pour convertir un objet générique vers un objet de l'utilisateur, on doit d'abord s'assurer de la validité de la conversion.
  - Cette tâche est confiée à une fonction de spécialisation **narrow**

```
org.omg.CORBA.Object  obj;
Compte unCompte = CompteHelper.narrow(obj); lien
```

Exemple Banque

- Liste des fichiers .java générés automatiquement par la compilation Java :

CompteOperations.java : déf. de l'interface contenant la signature des méthodes de l'objet Compte

Compte.java : déf. de l'interface Compte, qui hérite de CompteOperations

_CompteStub.java : Stub de l'objet Compte (représente l'objet distant)

CompteHelper.java : classe regroupant méthodes aidant à l'utilisation des objets Compte

CompteHolder.java : ens. d'outils prenant en charge le passage de paramètres avec les méthodes de l'objet Compte

ComptePOA.java : skeleton de l'objet Compte qui implante CompteOperations

ENSIAS

Exemple Banque (suite)

- Liste des fichiers .java développés :

CompteImpl.java : classe dérivée de ComptePOA

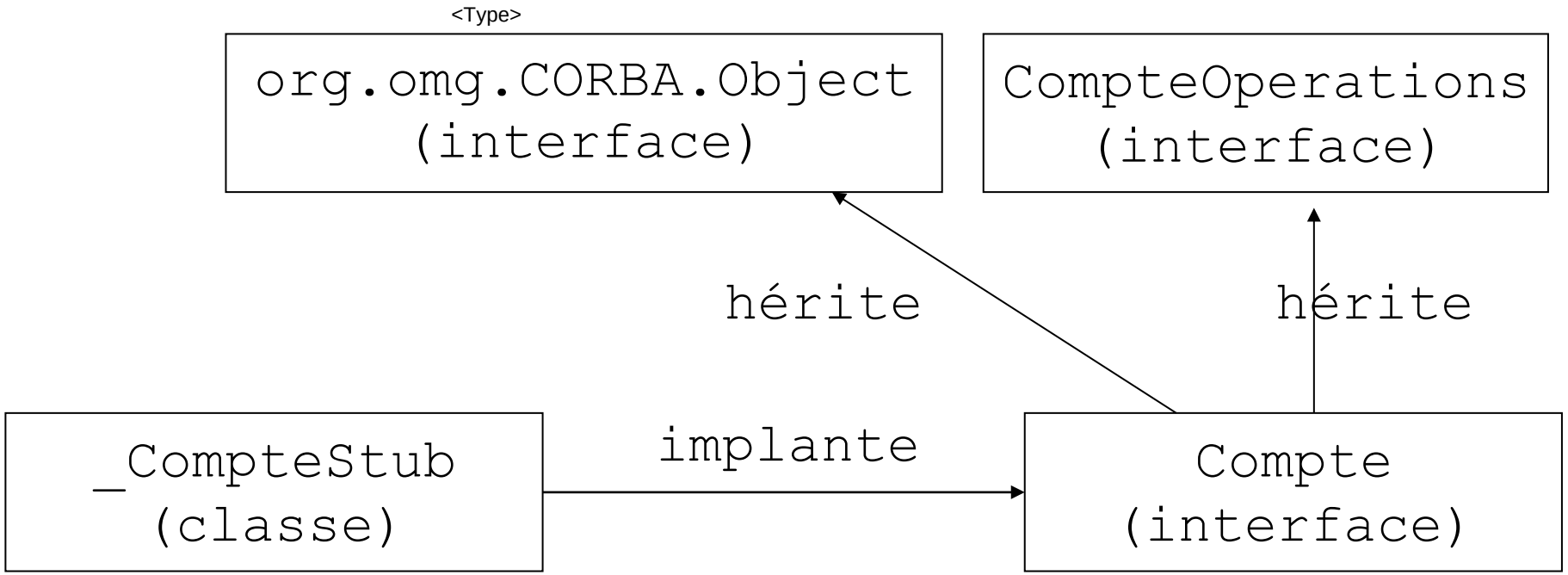
serveur_banque.java : application serveur Compte

client_banque.java : application cliente Compte

ENSIAS

Hiérarchie de classes : côté client

(Java)

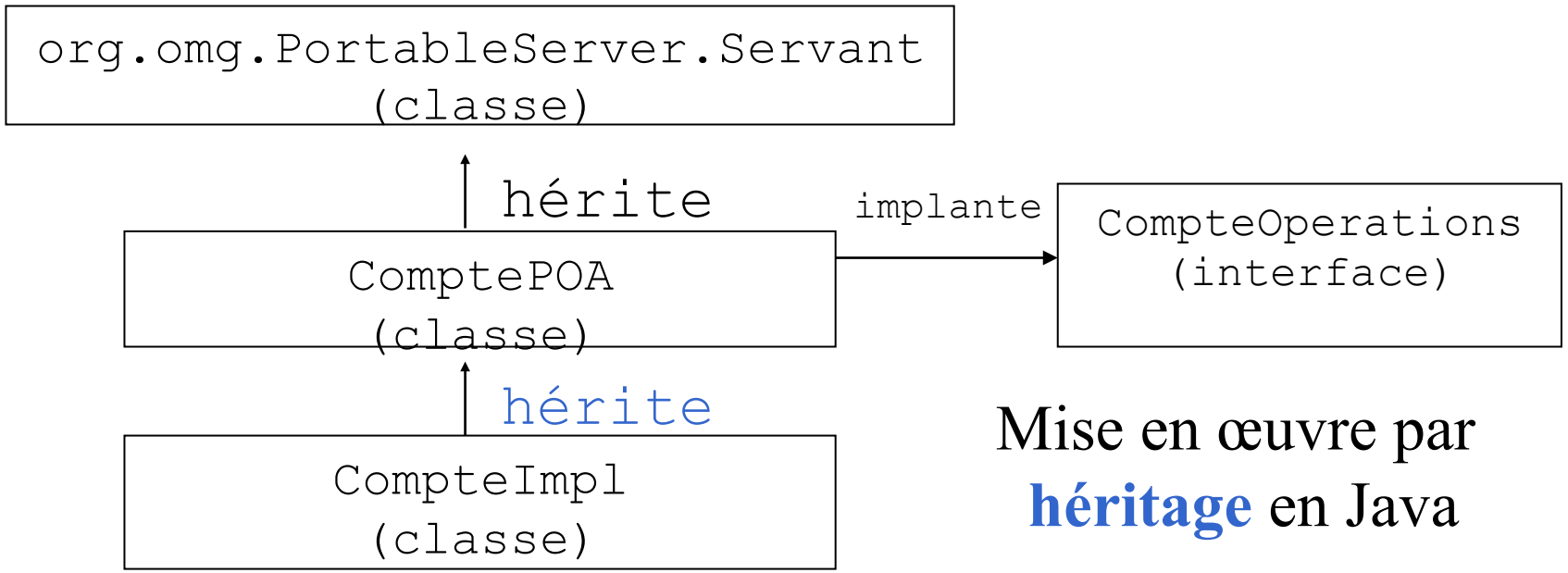


Mise en œuvre de l'objet : côté serveur

- Deux stratégies de mises en œuvre existent dans la spécification CORBA :
  - Mise en œuvre par héritage
  - Mise en œuvre par délégation

Hiérarchie de classes côté serveur (Mise en œuvre par héritage)

La classe de l'objet de mise en œuvre hérite de la classe skeleton générée par la compilation de l'IDL.



Mise en œuvre par **héritage** en Java

Programmation en Java

- On remarque que la classe org.omg.CORBA.Object n'apparaît pas. Elle peut être obtenue par la méthode _this générée automatiquement de la classe CompтePOA.

↳ Besoin d'hériter de cette classe à la déclaration du servant ainsi que les 2 initialisations.

```
public class CompteImpl extends ComptePOA
      {…    }
```

Serveur

```
CompteImpl unCompteImpl = new CompteImpl();
  Compte unCompte = unCompteImpl._this(orb);
```

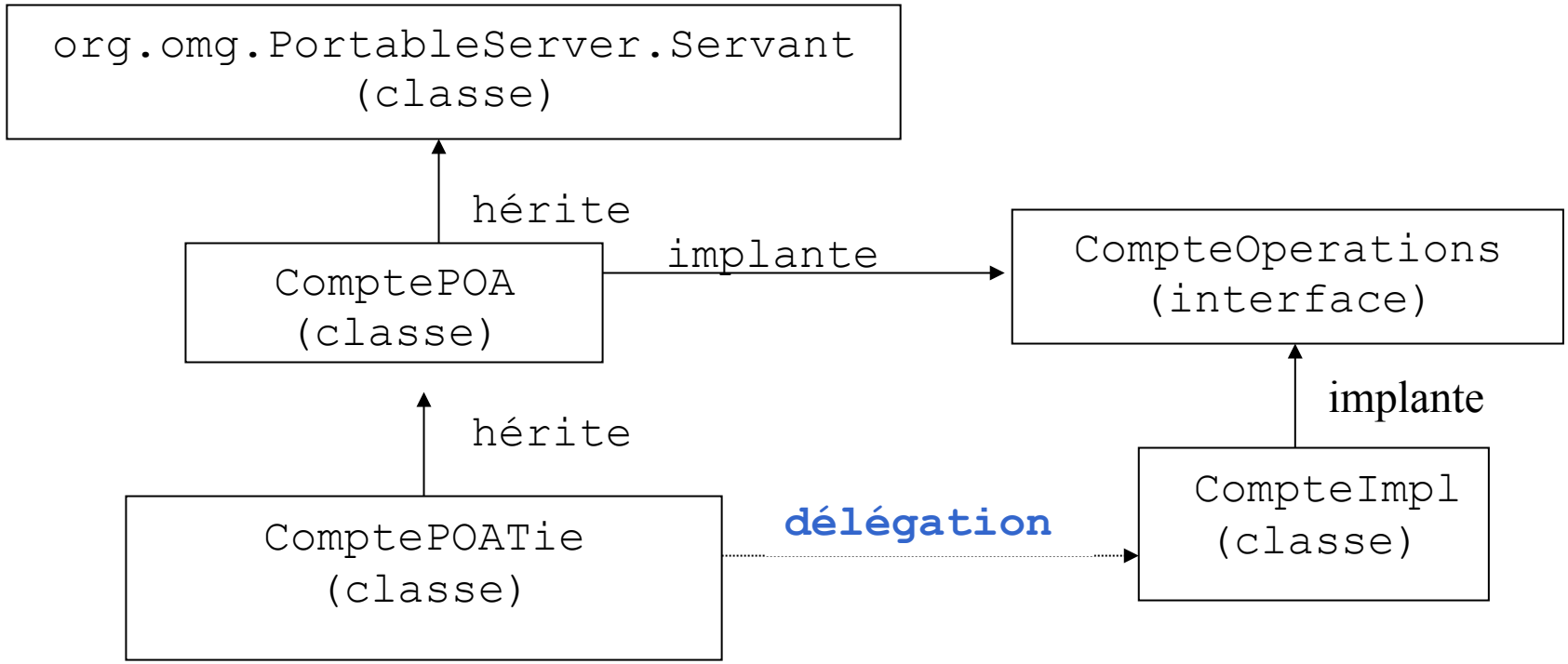
Mise en œuvre par délégation

- Si l'objet de mise en œuvre hérite lui même d'autres classes, il y aura héritage multiple : pb en java !

- S'il n'est pas possible ou souhaitable de changer la hiérarchie de classes de l'objet de mise en œuvre, on peut utiliser une classe de délégation (appelée **tie**) qui hérite ou réalise les méthodes du skeleton et qui redirige les appels de l'interface vers l'objet de mise en œuvre (**classe déléguée**).

Hiérarchie de classes côté serveur

Mise en œuvre par délégation en Java



L'objet de délégation

- L'objet de délégation est automatiquement généré à partir de la description IDL.
- l'objet de délégation doit être ***initialisé*** et ***connecté*** à l'adaptateur Réseau POA.
- Il faut ensuite ***exporter*** au client la référence de l'objet de délégation.

Programmation en Java

Objet de mise en œuvre

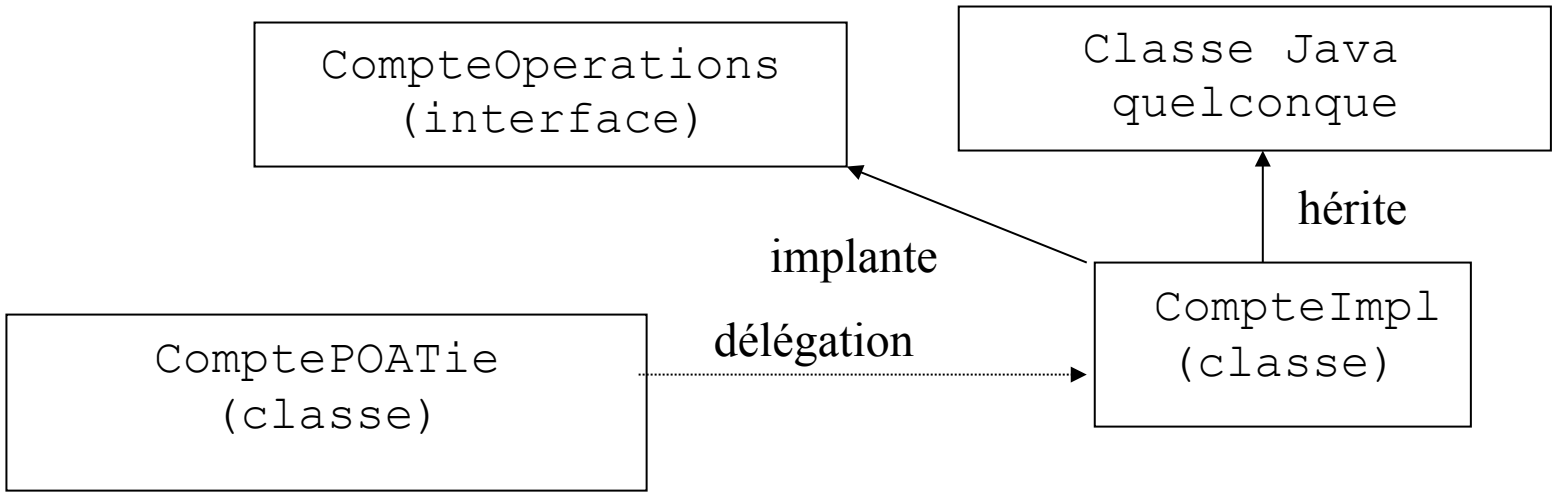
```
public class CompteImpl implements CompteOperations
{…}
```

Serveur

```
// instanciation de la classe déléguée
CompteImpl unCompte_delegue = new CompteImpl();
// instanciation de l'objet Tie et passage de
l'objet délégué
ComptePOATie unCompteImpl = new ComptePOATie(unCompte_delegue);
Compte unCompte = unCompteImpl._this(orb);
```

Conclusion mise en œuvre

- Mise en œuvre par héritage
  - plus performant :
    - ne nécessite pas la création d'une instance supplémentaire
    - l'appel de méthode est direct
- Mise en œuvre par délégation
  - plus souple :
    - permet le choix de la classe ancêtre => réutiliser des objets non Corba



IV. Mise en œuvre des objets Corba

Référence d'objet
Mise en œuvre côté client/ côté serveur
Le service de noms

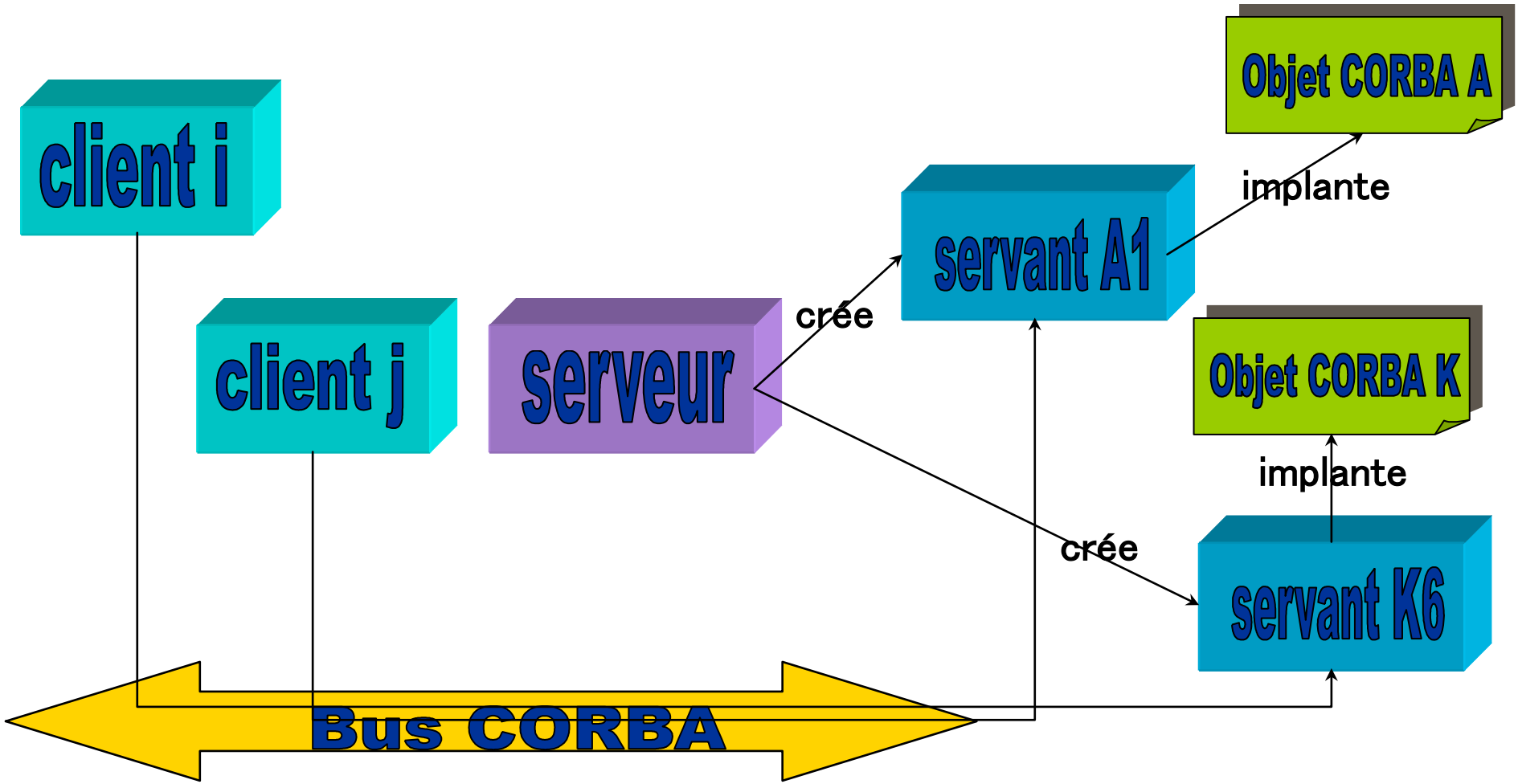
ENSIAS

Objet CORBA Versus Objet d'implantation (Servant)

- Objet CORBA :
  - entité virtuelle (n'existe pas en tant que composant réel) qui peut être localisée par un bus ORB et qui reçoit les requêtes clients qui lui sont délivrées.
- Servant :
  - entité réelle programmée en un langage, implante un objet CORBA et évolue dans le contexte d'un serveur.

⟹ Pour qu'un objet CORBA traite des requêtes, il a besoin d'un **Servant** qui concrétise son existence et réponde aux requêtes.

Objet CORBA Versus
Objet d'implantation (Servant)

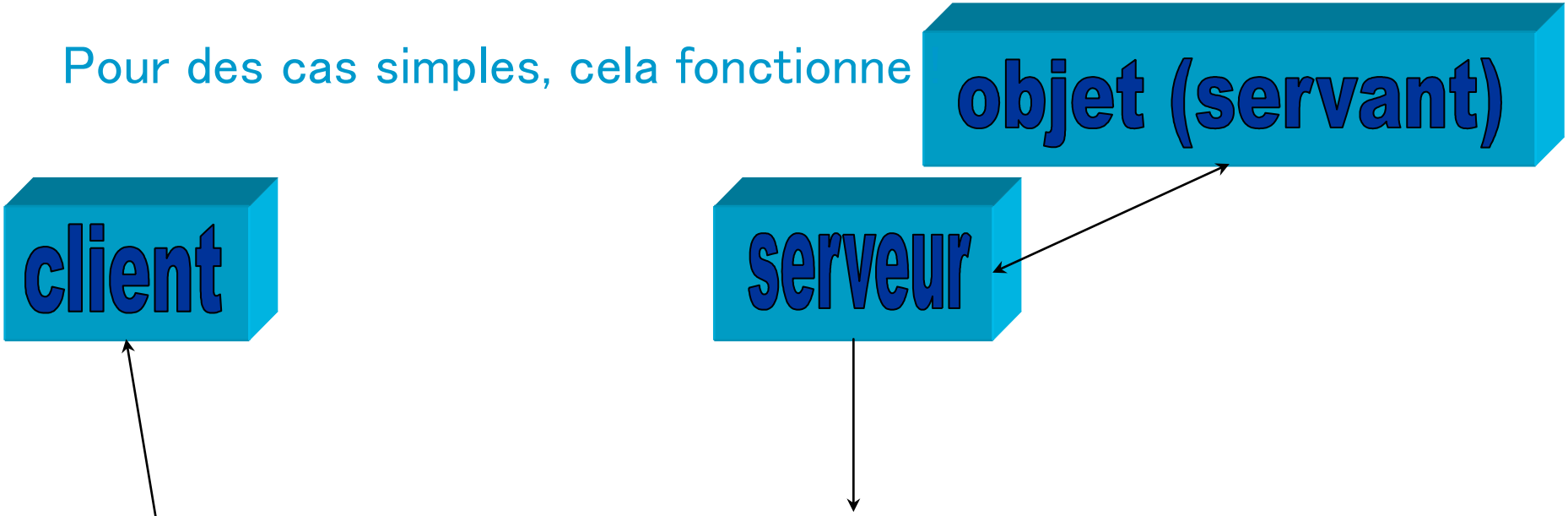


Invocation de méthodes d'un servant

- Diffusion de l'IOR non gérée par le bus
  - object_to_string (côté serveur) et string_to_object (côté client) de CORBA::ORB
  - fichier partagé (local ex : c:\lab-works\object.ref, web ex : http://www.mywebserver/object.ref,…)
  - fichier répliqué sur les clients
- Problèmes
  - Utilisation de plusieurs objets servants
  - L'application serveur est inconnue
  - Gestion des fichiers à la charge du programmeur (mise à jour, écritures concurrentes, réplication, nettoyage …)

Un serveur hébergeant un seul servant

Pour des cas simples, cela fonctionne



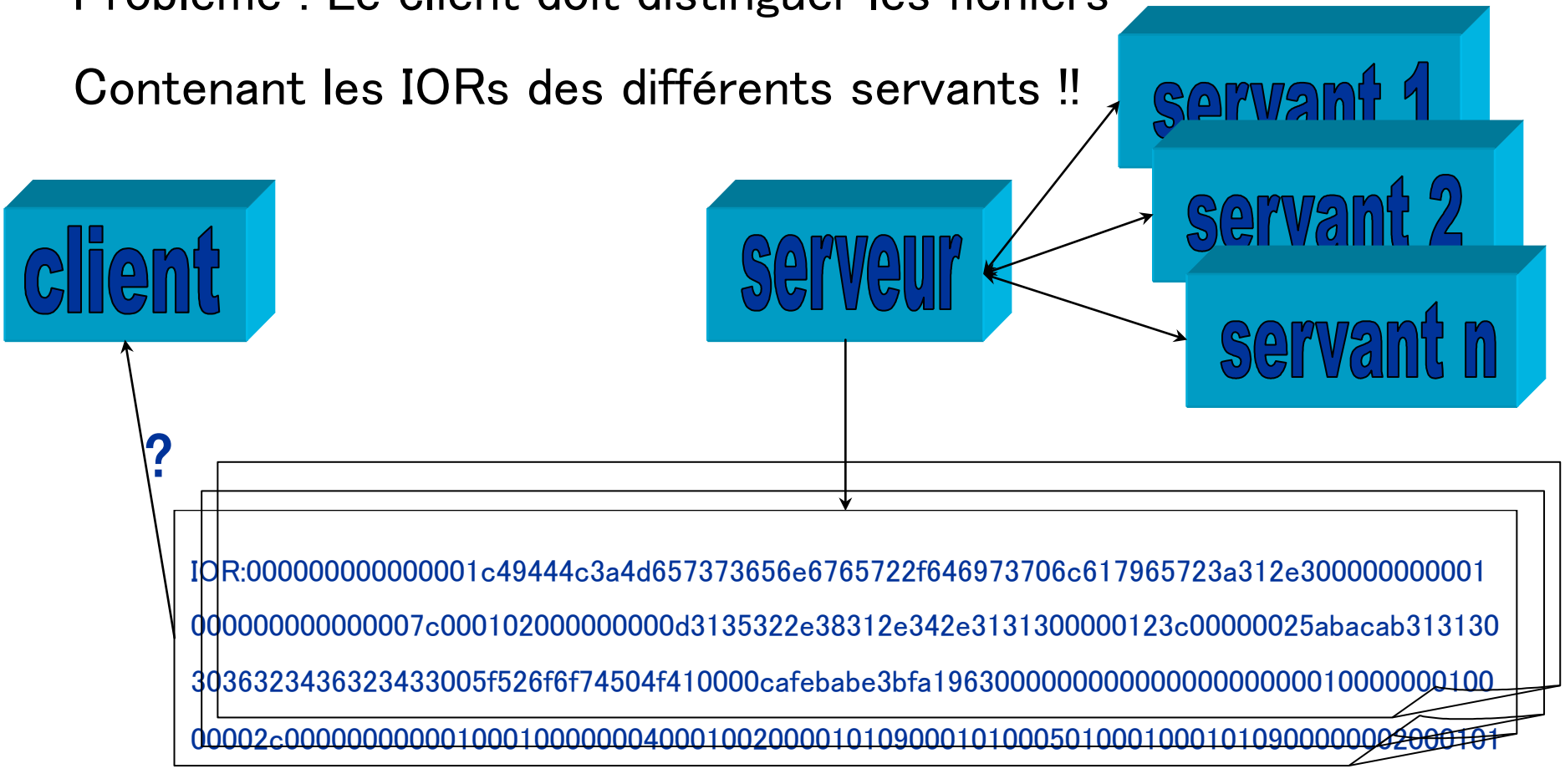
IOR:000000000000001c49444c3a4d657373656e6765722f646973706c617965723a312e300000000001
000000000000007c00010200000000d3135322e38312e342e3131300000123c00000025abacab313130
303632343632343300 5f526f6f74504f410000cafebabe3bfa1963000000000000000000000010000000100
00002c000000000001000100000004000100200001010900010100050100010001010900000002000101
0005010001

Fichier partagé ou le serveur « stringifie » l'IOR du servant

Problème de gestion des fichiers : un serveur hébergeant plusieurs servants

Problème : Le client doit distinguer les fichiers

Contenant les IORs des différents servants !!

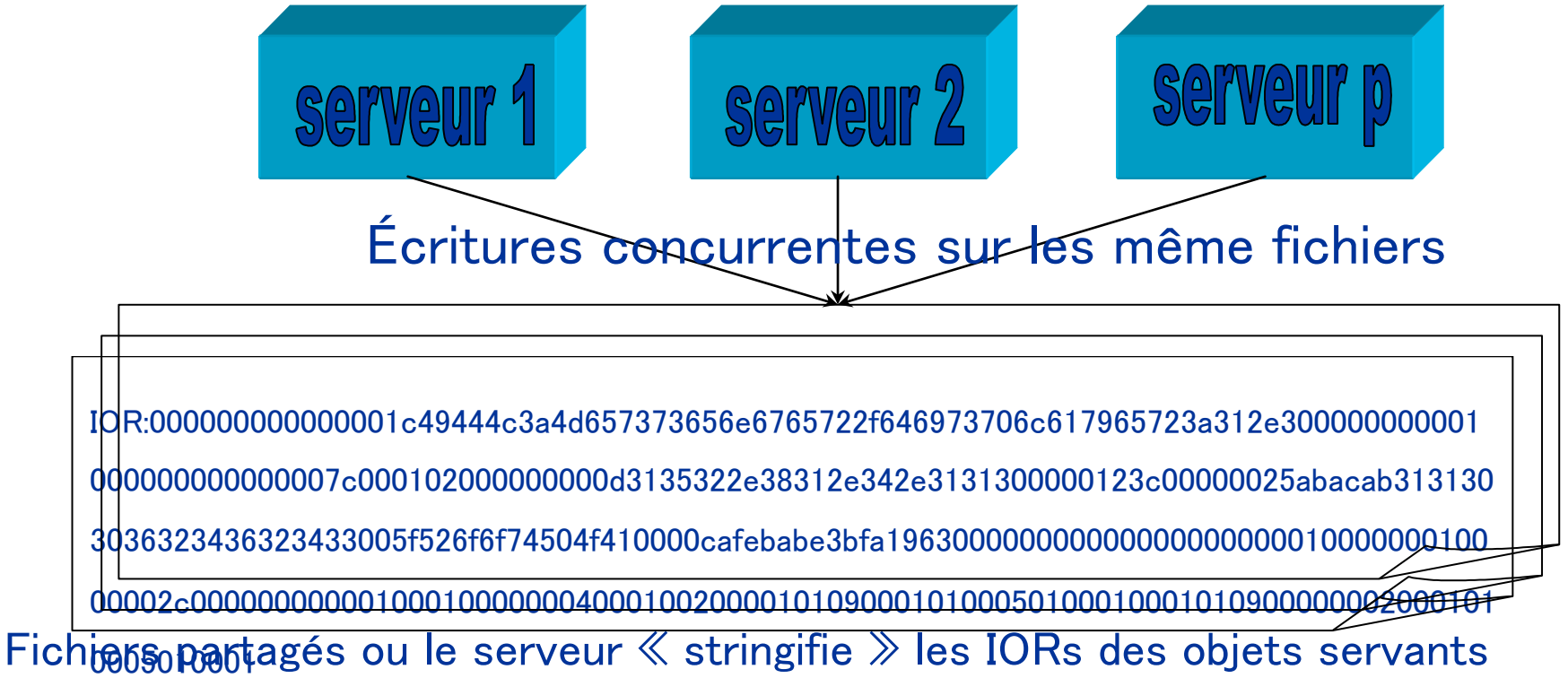


Fichiers partagés ou le serveur « stringifie » les IORs des objets servants

ENSIAS

Problème de gestion des fichiers : plusieurs instances du même serveur

Problème : Les instances du serveur ne doivent pas manipuler les même fichiers d'IORs !!



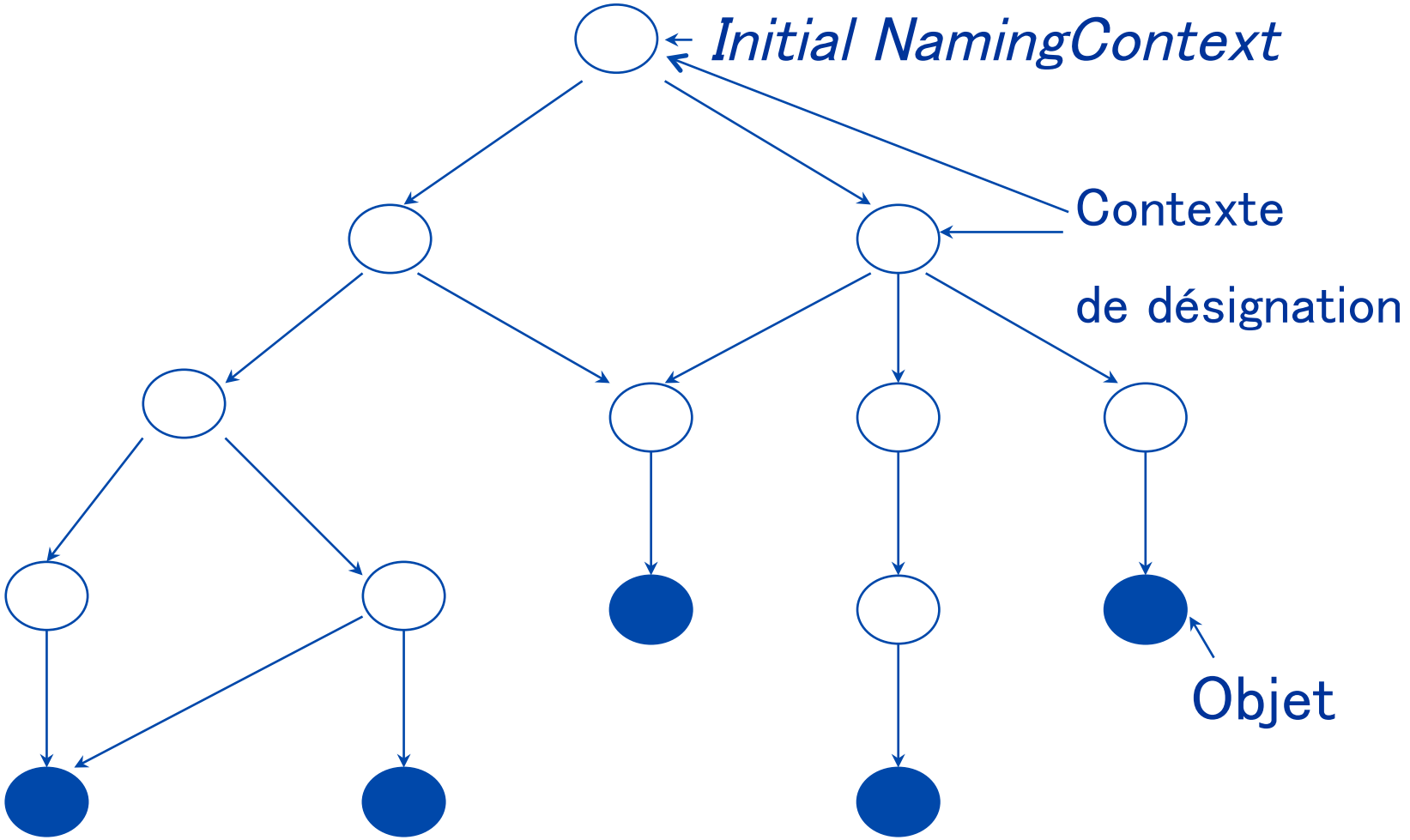
Utilité du service de Nommage

- Partage de références entre applications
- Utilisation d'un service standard
- Equivalent aux pages blanches, DNS
- Associer un nom à une référence d'objet CORBA
- Retrouver la référence d'objet
- Naviguer à l'intérieur d'un contexte de noms

Principes

- Le service d'annuaire permet d'associer les références d'objets à des noms symboliques
- Les noms et les références ainsi liés sont appelés des liaisons (nom, référence).
- L'espace de désignation est organisé sous forme de graphe hiérarchique de façon similaire à un système de fichiers.
  - Équivalent à la notion du chemin d'accès (répertoire/ sous répertoire : Structure de graphe)
    - Objet ~ fichier
    - Contexte ~ répertoire

Hiérarchie de l'espace de nommage



Chemin et nœud

- L'ensemble des nœuds suivis pour atteindre une référence d'objet depuis la racine de l'espace de désignation est appelé **chemin** (Name)
- Un chemin est représenté en IDL sous forme d'une séquence de **nœuds** (NameComponent)

```
Typedef sequence <nameComponent> Name
```

- Un nœud est caractérisé par son nom (identifiant le composant) et une sorte (description supplémentaire permettant la classification)
- Un nœud est représenté en IDL sous forme d'une structure

```
struct NameComponent
{       string id;
        string : kind;
};
```

ENSIAS

Les contextes de désignation

- Le module **CosNaming** définit
  - association nom - référence d'objet
  - graphes de répertoires : **NamingContext**
  - chemin d'accès : **Name**
- L'interface « CosNaming::NamingContext » du service de Nommage modélise les contextes de désignation

  Elle permet de :
  - Gérer les contextes (créer de nvx contextes, les lier à des contextes existants, les détruire)
  - Utiliser les contextes (ajouter, modifier et retrouver des liaisons (nom,référence))
  - Parcourir les contextes

Opérations principales du NamingContext

- Gestion du contexte
  - ajouter un contexte : bind_context
  - Créer un contexte : new_context
  - détruire le contexte : destroy
- Utiliser les contextes
  - Ajouter une liaison (nom, référence) : bind
  - Modifier une liaison : rebind
  - résoudre une liaison : resolve
  - détruire une liaison : unbind
- Parcourir les contextes
  - lister le contenu d'un contexte : list

Association

- Une association (Binding) permet de déterminer si un nœud de l'arbre de désignation correspond à un contexte de désignation ou à un objet.
- Une association est représentée en IDL par une structure :

```
enum BindingType {nobject, ncontext};
struct Binding
        {
                Name binding_name;
                BindingType binding_type;
        };
```

Exemples d'opérations du contexte de Nommage

```
interface NamingContext {
    void bind(in Name n, in Object obj) raises(NotFound, ...);
    void rebind(in Name n, in Object obj) raises(NotFound, ...);
    void bind_context(in Name n, in NamingContext nc) raises(...);
    void rebind_context(in Name n, in NamingContext nc) raises(...);
    Object resolve (in Name n) raises(NotFound, ...);
    void unbind(in Name n) raises(NotFound, ...);
    NamingContext new_context();
    NamingContext bind_new_context(in Name n) raises(...); //~mkdir
   void destroy() raises(NotEmpty); //~rmdir
   // Autres déclarations
   };
```

Les exceptions du contexte de Nommage

```
interface NamingContext {
    enum NotFoundReason {missing_node, not_context, not_object };
    exception NotFound {   // Pas de référence associée au nom
            NotFoundReason why; Name rest_of_name; };
    exception CannotProceed {   // Impossibilité d'effectuer l'opération
            NamingContext cxt; Name rest_of_name; };
    exception InvalidName { };  // Nom contenant des caractères
                                // invalides
    exception AlreadyBound { }; // Nom déjà utilisé
    exception NotEmpty { };     // Destruction d'un contexte non vide
    // . . .
};
```

Obtenir le service de Nommage

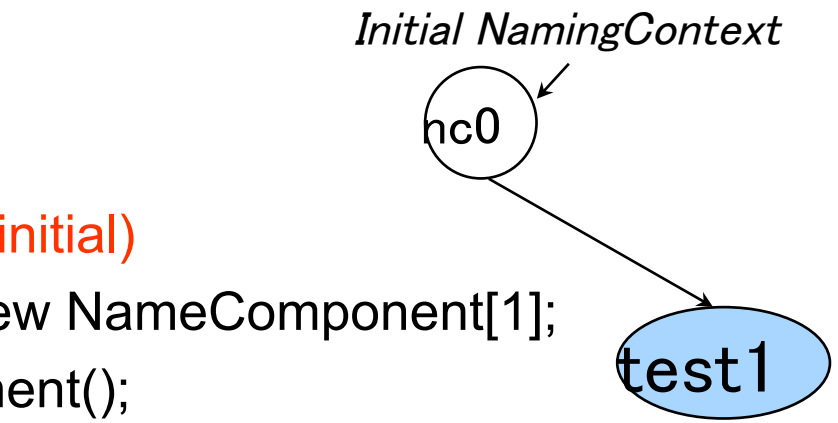
- Comment retrouver la référence de ce service ?
  - C'est un « objet » du bus CORBA
  - Souvent lancé sous forme d'un programme exécutable
- Du nom NameService fixé par configuration du produit utilisé
- Quelque soit le langage, le scénario est
  - a. opération resolve_initial_references de CORBA::ORB
  - b. conversion en CosNaming::NamingContext
    - Le NamingContext initial

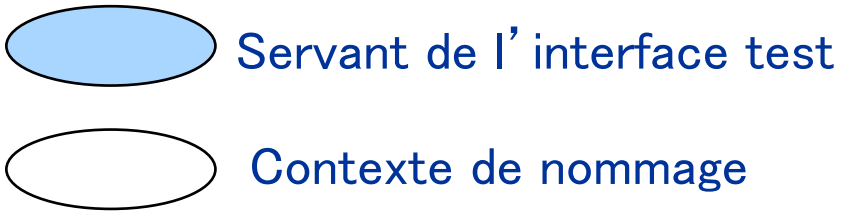
Obtenir le service de Nommage en Java

```
// Retrouver la référence de l'objet « NameService »

    org.omg.CORBA.Object nsObj = null;
    nsObj = orb.resolve_initial_references("NameService");
    …

// La convertir en une référence d'un objet de type
    CosNaming::NamingContext
    // (nc0 est le NamingContext Initial)
    org.omg.CosNaming.NamingContext nc0 = null;
    nc0 =
          org.omg.CosNaming.NamingContextHelper.narrow(nsObj);
```

Créer un chemin simple en Java

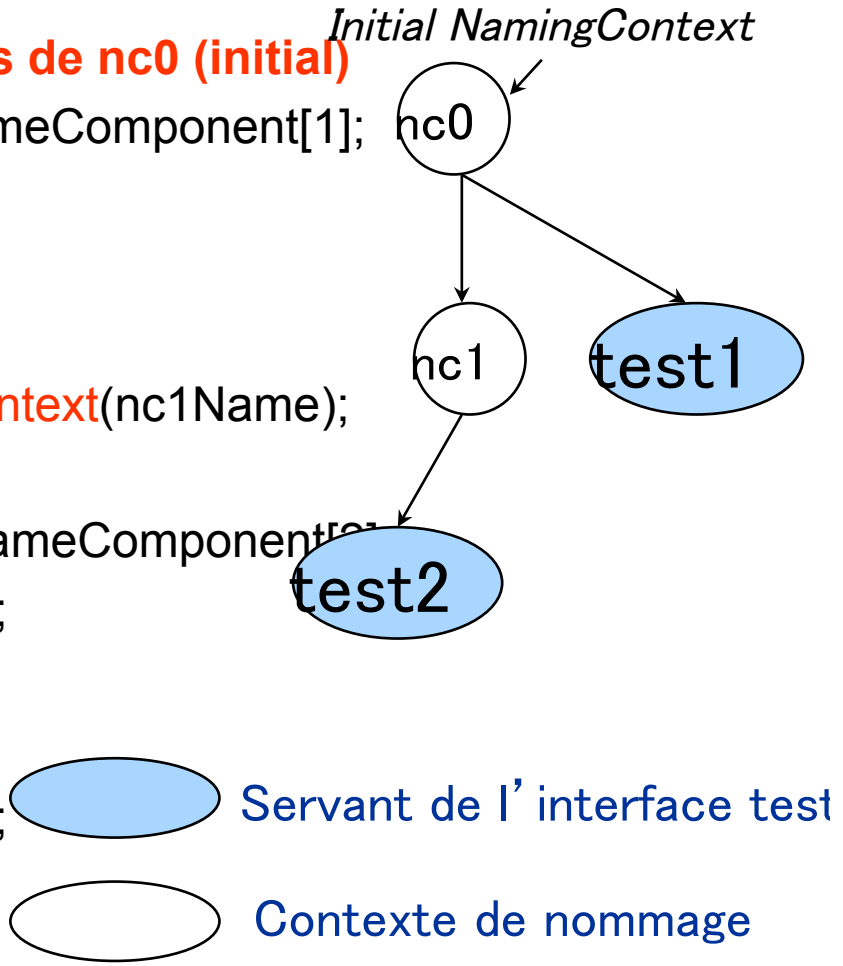
```
// création du servant test1
    test1 = new test_impl(..);


    // liaison de test1 au contexte nc0 (initial)
    NameComponent[] test1Name = new NameComponent[1];
    test1Name[0] = new NameComponent();
    test1Name[0].id = "test1";
    test1Name[0].kind = "";
    nc0.bind(test1Name, test1);
```





Créer un chemin Composé en Java

```
// création du servant test2
    test2 = new test_impl(..);
    // création du contexte nc1 comme fils de nc0 (initial)
    NameComponent[] nc1Name = new NameComponent[1];
    nc1Name[0] = new NameComponent();
    nc1Name[0].id = "nc1";
    nc1Name[0].kind = "";
    NamingContext nc1 = nc0.bind_new_context(nc1Name);
    // liaison de test2 au contexte nc1
    NameComponent[] test2Name = new NameComponent[2];
    test2Name[0] = new NameComponent();
    test2Name[0].id = "nc1";
    test2Name[0].kind = "";
    test2Name[1] = new NameComponent();
    test2Name[1].id = "test2";
    test2Name[1].kind = "";
    nc1.bind(test2Name, test2);
```



Enregistrer un objet en Java

```
// Créer un chemin (par exemple simple)
    NameComponent[] test1Name = new NameComponent[1];
    test1Name[0] = new NameComponent();
    test1Name[0].id = "test1";
    test1Name[0].kind = "";

// création du servant test1
    test1 = new test_impl(..);

// Enregistrer l'objet
    nc0.bind(test1Name, test1);
   // nc0.unbind (test1Name); pour détruire l'association
…
```

ENSIAS

Retrouver un objet en Java

```
// Créer un chemin (par exemple simple)
    NameComponent[] test1Name = new NameComponent[1];
    test1Name[0] = new NameComponent();
    test1Name[0].id = "test1";
    test1Name[0].kind = "";

// Retrouver l'objet
    org.omg.CORBA.Object testObj = nc0.resolve(test1Name);

// Convertir la référence
    test Test = testHelper.narrow(testObj);
```

ENSIAS

Avantages/Inconvénients

- Avantages
  - Noms symboliques + conviviaux et + signifiants que IOR
  - Les liaisons (nom, référence) permettent l'indépendance à la localisation de l'objet (seul le nom symbolique est manipulé par les clients).
  - Organisation hiérarchique et classification possible (utile si le nb d'objets est grand).

- Inconvénients
  - Coût de consultation et de recherche dans l'annuaire.
  - Les liaisons (nom, référence) peuvent être obsolètes.
  - Ne permet pas une interopérabilité complète : l'accès au SN passe par l'ORB (service resolve_initial_services)

V. Interface Definition Language (IDL)

OMG-IDL : le langage d'interfaces

- Le langage IDL a été défini :
  - pour permettre la communication entre les applications clientes et les objets CORBA
    - Abstraction des langages de programmation utilisés
    - Description des interfaces des objets par la définition d'un comportement ou contrat IDL
  - IDL décrit uniquement les interfaces des objets
    - Séparation de l'interface et de son implantation
    - il n'inclut aucune structure algorithmique et ne permet pas la définition de variables

⇒ Serveur implante une interface, Client utilise l'objet serveur à travers l'interface (les clients ne voient pas détails d'implantation)

Caractéristiques du langage IDL

- IDL a été fortement inspiré de C++
  - enrichi de caractéristiques propres aux mécanismes d'invocation des objets distribués
- IDL est un langage déclaratif
  - non un langage de programmation
  - offre des types de base tels que *long*, *double* et *boolean*, et des types construits tels que *struct*, *union*, *sequence* et *string*.

Caractéristiques du langage IDL (suite)

- La spécification des interfaces :
  - correspond à la 1° étape de l'implantation d'une application objet distribuée
  - indépendante du langage d'implantation
  - est utilisée pour :
    - permettre la génération des **souches** IDL côté client et des **squelettes** IDL côté serveur
    - peupler le répertoire d'interfaces utilisé lors de l'invocation dynamique

Une description IDL

- Eléments d'une description IDL
  - Module, Interface, Constante, Type, Attribut, Opération, Exception
- Possibilité de définir macros ou directives d'inclusion (#include)
- Mot-clé interface : le seul à permettre la définition d'objets Corba
- Interface : essentiellement {attributs et opérations}

Forme générale d'une description IDL

- Description IDL :
  - Fichier texte de suffixe .idl
  - Sa structure a une syntaxe précise :

```
<types>
<constantes>
<exceptions>
<modules>
        <types>
        <constantes>
        <exceptions>
        <modules>
        <interfaces>
                <types>
                <constantes>
                <exceptions>
                <attributs>
                <opérations>
```

Types, const, exceptions déclarés globalement, localement à un module ou à une interface

Les descriptions du langage IDL (1)

- Constantes

  `const <type> <nom> = <valeur> ;`

  `<type>` : n'importe quel type sauf `any`.

  `<valeur>` : expression arith. ou logique.

  exemple : `const double PI = 3.14159 ;`

  `const string c = « message »;`

- Nouveaux types

  `typedef <déclaration> <nom> ;`

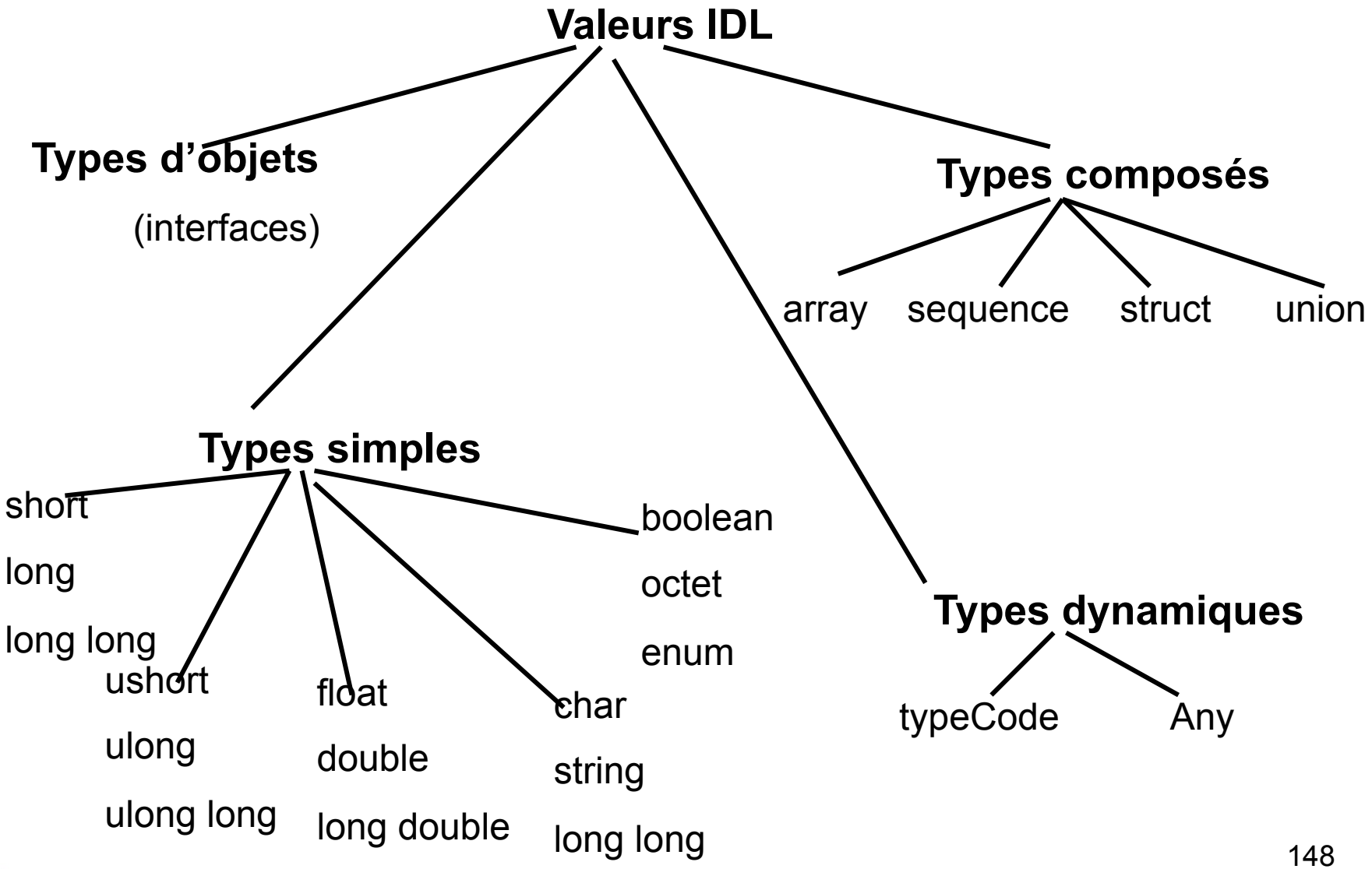
  exemple : `typedef unsigned short Annee ;`

- Types énumérés

  `enum <nom> { <V1>, … , <V2> } ;`

  exemple : `enum JourSemaine {Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche} ;`

Les descriptions du langage IDL (2)



Les descriptions du langage IDL (3)

- **Les métatypes de données** :
  - types de données contenant une valeur et une information décrivant le type de celle-ci
  - **TypeCode** :
    - type de données permettant de stocker la description de tout autre type de données IDL
    - Exemple : un TypeCode décrivant une structure contiendra le nom de la structure et la liste de tous les champs
  - **Any** :
    - sert à stocker n'importe quelle valeur IDL et son TypeCode associé
    - Employé lorsqu'il est nécessaire de transporter des valeurs dont le type n'est pas encore connu à la conception
    - Exemple : définition d'une pile générique

Les descriptions du langage IDL (3)

- Structures

```
struct <nom> {<déclarations>};
```

exemple :
```
struct  MaDate {
  Jour le_jour ;
  Mois le_mois ;
  Annee l_annee ;
} ;
```

- Unions : forme particulière de struct, avec un champ de type variable en fonction de la valeur d'un discriminant

```
union <nom> switch <type> {…} ;
```

exemple :
```
union DateMultiFormat switch (long) {
  case 1 : Date format_date;
  case 2 :
  case 3 : string format_chaîne ;
  default: Jour nombre_de_jours ;
} ;
```

ENSIAS

Les descriptions du langage IDL (4)

- Tableaux : type et taille explicites

  exemple :
  ```
  typedef long Vecteur[100] ;
  typedef short matrice[50][100] ;
  ```

- Séquences : tableau à une dimension, d'éléments de même type, borné ou non, utile pour l'échange des tableaux dynamiques

  exemple :
  ```
  typedef sequence<long> LongSeq ;
  typedef sequence<float,10> FloatSeq ;
  ```

- Enumérations : énumération d'id. regroupés sous un même type

  exemple :
  ```
  enum Couleur {Rouge, Jaune, Bleu, Vert} ;
  ```

Déclaration modules et interfaces (1)

- Modules
  - correspondent à un espace de nommage (cf. package Java) qui permet un regroupement de définitions IDL
  - évitent les conflits de noms entre définitions
  - déclaration :

```
module <nom> { <ensemble de déclarations> } ;
```

Exemple :

```
module monApplication
{
 typedef long monLong;
};
```

Déclaration modules et interfaces (2)

- Interfaces
  - représentent un point d'accès à un objet
  - décrivent l'ensemble des services (opérations et des attributs) offerts par un type d'objets
  - déclaration :

```
interface <nom> { <ensemble de déclarations> } ;
```

Exemple

```
module gestionBancaire {
  struct FicheClient{ string nom, string adresse};
  interface Compte {
        // attributs et opérations
};
```

Déclaration modules et interfaces (3)

- Portée d'une déclaration
  - la portée d'une définition limitée à l'espace du module
  - accessibilité extérieure d'une définition en la préfixant par le nom du module séparé par **::**

```
module M {
        interface I {
        typedef char T;
        };
};
```

  - Le type T déclaré dans l'interface I du module M peut-être utilisé dans un autre module en utilisant **M : : I : : T**

Exemple

```
module date { struct Date {…} ; };
module monApplication {
 interface monService {
        date::Date retourner_date_courante();
 };
};
```

Déclaration des opérations (1)

- Opérations
  - représentent l'abstraction d'un traitement réalisé par un objet
  - syntaxe :

```
<mode d'invocation> <type> <nom> (<paramètres>)
[raises (<liste exceptions>)]
[context (<liste contextes>)] ;
```

  - mode d'invocation : par défaut appel synchrone
    - Opération uni-directionnelle :
    - asynchrone avec `oneway`, valeur de retour `void`, uniqt des paramètres `in`, pas de clause `raises`.

```
oneway void g(in long a);
```

Déclaration des opérations (2)

- liste paramètres : mode de passage pour tous les paramètres
  - in : le paramètre est transmis vers l'objet
  - out : le paramètre est transmis par l'objet
  - inout : échange dans les 2 sens

Exemple

```
long f(inout float c, in string c) raises
  (TropLourd);
```

Déclaration des opérations (3)

– liste exceptions :

- Exceptions pouvant être déclenchées par cette opération pour signaler une erreur
- CORBA définit **2 types d'exceptions** :
  - les exceptions systèmes (pbs liés à l'utilisation bus Corba, implicitement spécifiées dans la signature des opérations)
  - les exceptions utilisateur (pbs liés à mauvaise utilisation de l'objet, explicitement spécifiées dans la signature des opérations)
- Les définitions IDL ne font aucune référence aux exceptions systèmes

Exemple

```
interface Compte {
   exception WithDrawFailure {string raison;};

   void makeWithdrawal(in float somme,out float
   nouvelleBalance)
   raises (WithDrawFailure);
…
};
```

Déclaration des opérations (4)

– liste contextes : permet de transmettre l'équivalent de variables d'environnement depuis le client vers les objets (très peu utilisée)

Exemple

```
Long l ( in string nom ) context ("login")
```

Déclaration des attributs (1)

- Attributs
  - spécifications décrivant une propriété d'un objet
  - peuvent être accédés en lecture-écriture (mode par défaut) ou en lecture seulement (`readonly`)
  - déclaration :

```
<mode> attribute <type> <nom> ;
```

  - exemple :

```
interface Compte {
  // attributs
 readonly attribute float balance;
  };
```

Déclaration des attributs (2)

- un attribut correspond à 2 opérations d'accès à la propriété
  - consultation et modification
  - exemple :

```
interface exemple {
  attribute long unePropriete;
  readonly attribute long uneAutre;
  };
// équivalent à :
interface exemple {
  long consulter_unePropriete();
  void modifier_unePropriete(in long v)
  long consulter_uneAutre();
};
```

L'héritage d'interfaces (1)

- Classification/spécialisation de types d'objets
  - Trois formes d'héritage : simple, multiple et répété
    - héritage multiple à condition qu'il n'y ait pas de conflit d'héritage sur les attributs ou les opérations
  - Redéfinitions et/ou spécialisations des opérations héritées sont interdites
  - Déclaration :

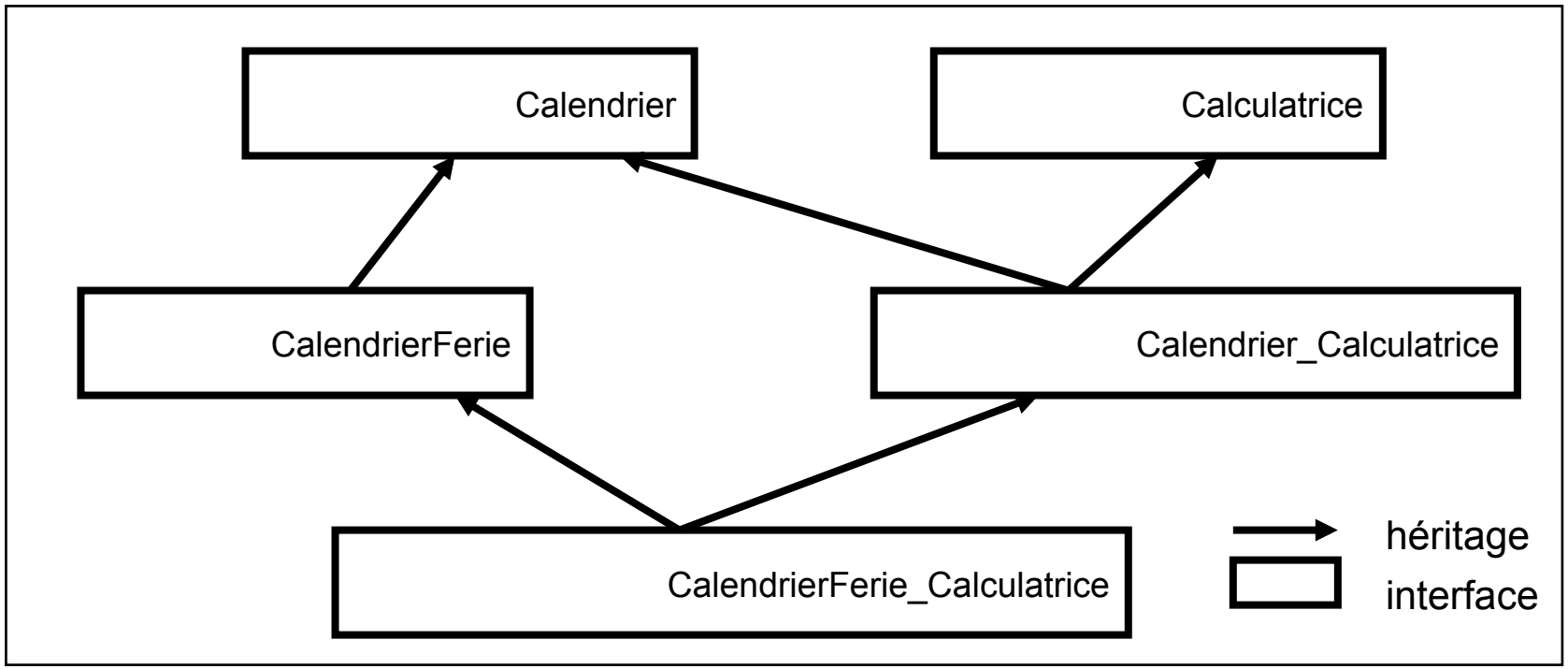
```
Interface <nom-interface-derivee> : <nom classe
  mère> { …} ;
```

  - Exemple :

```
interface CalendrierFerie : Calendrier {
  void les_jours_feries(in Annee de_l_annee, out
  desDates dates);
};
```

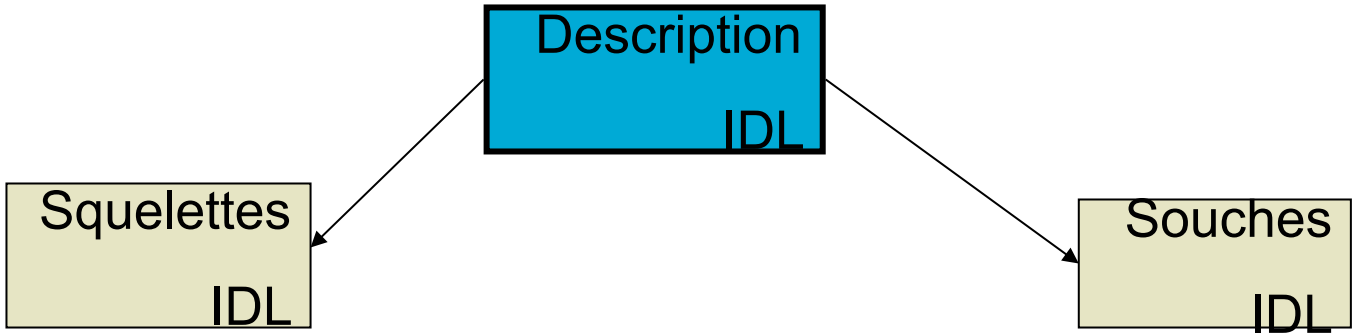
L'héritage d'interfaces (2)

```
interface Calculatrice {…};
  interface Calendrier_Calculatrice : Calendrier,
  Calculatrice {};
  interface CalendrierFerie_Calculatrice :
   CalendrierFerie, Calendrier_Calculatrice { };
```



Projection de l'IDL

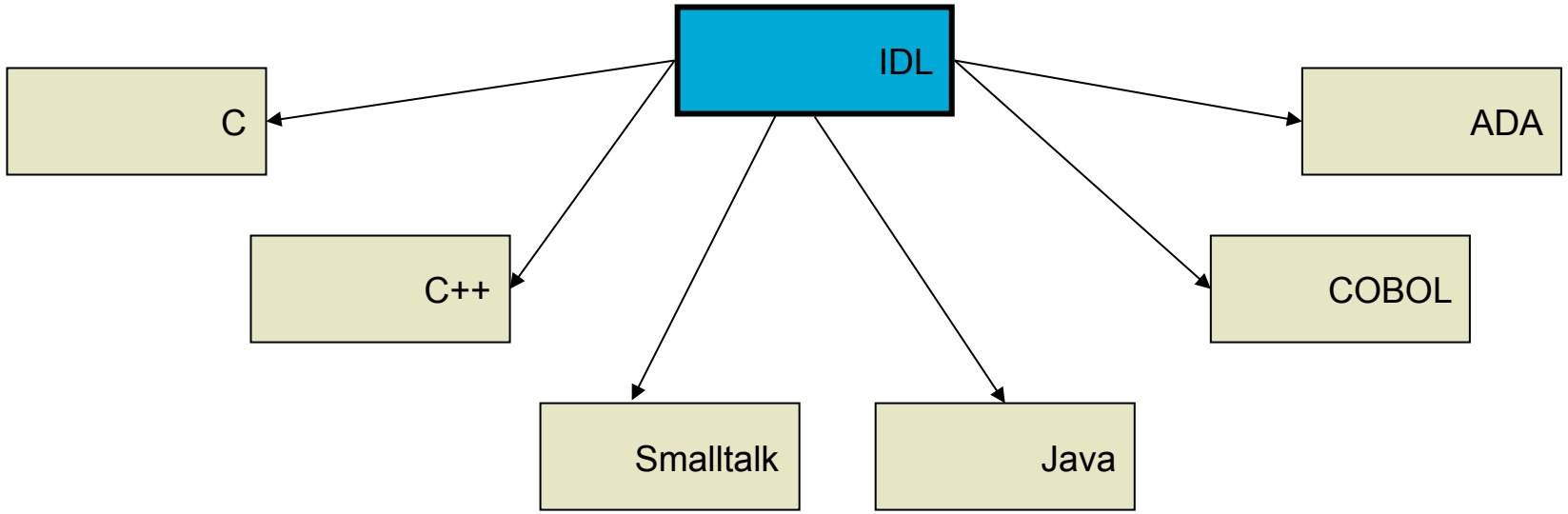
- Mécanisme de projection (mapping)
  - A partir d'une description IDL : 2 projections réalisées
    - celle pour implanter les objets (squelettes IDL)
    - celle pour les applications utilisant ces objets (souches IDL)



  - Pour une interface donnée : les projections peuvent être différentes
    - Serveur en C++ : squelettes en C++
    - Client en Java : souches en Java

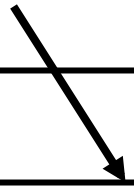
Projection de l'IDL

- Règles de projection :
  - des règles de transformation convertissent une interface en IDL dans les langages d'implantation des objets correspondants



Projection de l'IDL

```
module Mymodule
{
   interface MyInterface
   {
       attribute int lines;
       void printLine (in string toPrint);
   };
};
```



IDL vers Java

```
Public interface Mymodule.MyInterface extends ...{
  public int lines(); // get_lines
  public void lines(int lines); // set_lines
  public void printLine (java.lang.string toPrint);
  …
};
```

ENSIAS

Conclusion IDL

- OMG IDL : langage de déf. des interfaces des objets CORBA.
- Interface : principalement ensemble d'attributs et d'opérations.
- Principales différences avec des langages de programmation :

Pas d'instructions de contrôle

Pas de pointeur

Pas de constructeur

Pas de destructeur

Pas de surcharge d'opérations

Sens de passage des paramètres

Sémantique d'appel oneway

Attributs readonly

ENSIAS

VI. Le mapping Java

Projection IDL/Java

- Objectifs
  - utilisation d'objets CORBA depuis un programme Java
  - implanter des objets Java en CORBA
- Portabilité totale
  - langage Java unique
  - Sources Java 100% portables
- Compiler 1 fois, exécuter partout
  - tous les bus Corba Java

Projection vers Java

- Mission : traduire une description IDL vers un langage de programmation cible.
- Les règles de projection dépendent de la version de l'OMG IDL et des évolutions du langage Java.
- Les règles définissent les équivalents Java des types IDL ainsi que ceux de l'interface Object et d'autres standardisés (tels que ORB).

Projection côtés Client / Serveur

- Côté client
  - Souche : code généré auto. par compilateur IDL dans langage cible (utilisé par le client lors des invocations statiques)
- Côté serveur
  - Squelette : code généré auto. par compilateur IDL dans langage cible (utilisé par l'adaptateur d'objets lors des invocations statiques)

Passage de paramètres

- Paramètres IN :
  - Passage par valeur,
  - Valeur créée et initialisée par l'appelant.
- Paramètres OUT :
  - Utilisation des classes « Holder »,
  - Valeur créée et initialisée par l'appelé.
- Paramètres INOUT :
  - Utilisation des classes « Holder »,
  - Valeur « in » créée et initialisée par l'appelant,
  - Valeur « out » créée et initialisée par l'appelé.
- Valeur de retour :
  - Passage par valeur,
  - Valeur créée et initialisée par l'appelé.

Les classes « Holder »

- Java: pas de passage "out" et "inout" (passage par adresse)
- De façon à permettre l'utilisation des paramètres de type OUT et INOUT de CORBA, on introduit la notion de classe « Holder ».
- Solution : les classes Holder
  - gèrent les paramètres out/inout et simulent le passage par adresse
    - un attribut contenant la valeur
    - un constructeur par défaut (out)
    - un constructeur avec valeur initiale (inout)

Classes Holder et passage de paramètres (1)

- Chaque type de base de l'IDL dispose d'une classe « Holder » correspondante :

- De même, chaque type défini par l'utilisateur dispose d'une classe « Holder » générée par le compilateur d'IDL.

```
public final class TYPEHolder {
  public TYPE value ;
  public TYPEHolder () ;
  public TYPEHolder (TYPE v) {
         value = v;
  }
}
```

TYPE étant un type de base ou un type défini par l'utilisateur

- Le contenu de l'instance de classe « Holder » est modifié lors de l'invocation.

Classes Holder et passage de paramètres (2)

//Java

```
package org.omg.CORBA
public class IntHolder {
  public Int value ;
  public IntHolder () ;
  public IntHolder (int v) {
         value = v;
  }
}
```

//dans le fichier compteHolder généré par le compilateur IDL

```
public class compteHolder {
  public compte value ;
  public compteHolder () ;
  public compteHolder (compte v) {
         value = v;
  }
}
```

Classes Holder et passage de paramètres (3)

//IDL

```
interface Modes {
  long operation (in long inArg, out long outArg,
  inout long inoutArg};
```

//Java

```
interface Modes extends org.omg.CORBA.Object {
 int operation(int inArg
               org.omg.CORBA.IntHolder outArg,
               org.omg.CORBA.IntHolder inoutArg);
}
```

Classes Holder et passage de paramètres (4)

Import org.omg.CORBA.IntHolder;

//Obtenir une référence d'objet

```
Modes obj = ...
```

//Fixer la valeur des paramètres

```
int inArg = 20;
IntHolder outArg = new IntHolder();
IntHolder inoutArg = new IntHolder(80);
```

//Invoquer l'opération

```
int result = obj.operation(inArg,outArg,inoutArg);
```

//Utiliser les valeurs retournées

```
int v1 = outArg.value ;
int v2 = inoutArg.value ;
```

ENSIAS

Classes Helper (1)

- Le compilateur IDL génère des classes Helper pour toutes les interfaces définies par l'utilisateur
- Nom de la classe Helper : <type>Helper
- Ces classes contiennent des méthodes permettant de manipuler ces types IDL de différentes façons :
  - Insertion/ extraction vers le type Any
  - obtenir le repositoryID
  - lecture et écriture dans un stream CORBA
  - opération de narrow pour les interfaces (qui permet de convertir un type Object (Corba) vers l'objet de l'utilisateur).
- Chaque type défini par l'utilisateur dispose d'une classe supplémentaire appelée « Helper ».

Portée des déclarations

- Projection sur les règles liées aux « package » et « class » Java.
- Par exemple, l'IDL:

```
module M {
        interface I {
                typedef char T;
        }
}
```

- Le (ou les) type(s) correspondant(s) en Java aura (auront) une portée de définition incluse dans l'interface « I » du package « M »

Le mapping Java

- Les interfaces du bus CORBA sont décrites dans un module IDL représenté en Java par le package:

  org.omg.CORBA

  - Les classes abstraites Any et TypeCode
  - Les classes Holder et Helper de base
  - les classes pour les exceptions système
  - Les classes abstraites Object, ORB, BOA et POA
  - Des classes pour le DII et le DSI
  - …
- Identificateur IDL = Identificateur Java
  - si conflit alors identificateur précédé de ‘ _ ’
- Commentaires : /* sur plusieurs lignes */

  // sur une seule ligne

ENSIAS

Mapping des types simples de base

IDL	Java	Exceptions
void	void	
short, unsigned short	short	
long, unsigned long	int	
long long, insigned long long	long	
float	float	
double	double	
char, wchar	char	CORBA::DATA_CONVERSION
string, wstring	java.lang.string	CORBA::MARSHAL CORBA::DATA_CONVERSION
boolean	boolean	
octet	byte	

- Types Java « plus larges » que types IDL

Les types de base (suite)

Le type « any »

- Projection sur la classe org.omg.CORBA.Any.
- Cette classe dispose de toutes les méthodes nécessaires pour insérer ou extraire des instances de types prédéfinis.

Définition d'un type de données (1)

- Java ne dispose pas de mécanismes de définition de types tel que « typedef ».
- Pour les types IDL simples, le type original est utilisé à la place du type défini.
- Pour les types IDL complexes, le type défini est projeté sans lien direct avec les type original

Définition d'un type de données (2)

- Si un type IDL est défini à l'intérieur d'une interface : pb parce que Java ne permet pas d'imbriquer des classes à l'intérieur d'interfaces.
  - Or le type IDL possède au minimum une classe Java correspondante (la classe « Helper »).
  - Les classes Java générées doivent être déclarées dans un domaine de définition correspondant à un package ayant pour nom celui de l'interface suivi de « Package ».
  - voir exemple: la classe « THelper » est définie dans le package « M.IPackage ».

Mapping des modules

- Un module IDL :
  - un package Java du même nom
- Toutes les déclarations de types IDL à l'intérieur du module :
  - traduites en déclarations de classes et d'interfaces correspondantes à l'intérieur du package généré
- Exemple :

```
module IDLDemo {
        interface banque {
                struct Details {…}
                …
        };
};
```

La classe correspondante à la structure Details est :

IDLDemo.banquePackage.Details

ENSIAS

Mapping des interfaces (1)

- Une interface IDL = 2 interfaces Java (interface *signature* de même nom et une interface *opérations*) + un certain nombre d'autres constructions Java

Exemple :

```
public interface Compte extends CompteOperations,
                                org.omg.CORBA.Object,
                                org.omg.CORBA.portable.IDLEntity

public interface CompteOperations
```

Mapping des interfaces (2)

- Les constructions suivantes sont générées par le compilateur IDL pour une interface IDL<type>

Fichiers générés	Description	Côté
<type>.java	Interface Java	client
_<type>Stub.java	Classe Stub Java	client
_<type>POA.java	Classe Skel Java	serveur
_<type>Operations.java	Interface Java	client/serveur
_<type>Helper.java	Classe Helper Java	client/serveur
<type>Holder.java	Classe Holder Java	client/serveur
<type>Package	Package Java	client/serveur

Projection des attributs

- Un attribut est directement converti en une ou deux fonctions Java selon que l'attribut est en lecture seule (readonly) ou non.

```
interface Compte{
  attribute float solde;
};
```

```
public interface CompteOperations {

    float solde();
    void    solde(float val);

}
```

Projection des opérations

- Une opération est traduite en une fonction. Ses paramètres sont des Holders si ceux-ci sont de types OUT et INOUT.

```
// IDL
interface Exemple {
   long operation ( in long a, out long b, inout
                    long c );
};

// JAVA
public interface ExempleOperations {
   int operation ( int a, IntHolder b,
                   IntHolder c );
}
```

Héritage d'interfaces

- Indépendance entre l'héritage des implantations et l'héritage des interfaces.
- Classe de base commune
  - Côté client : `org.omg.CORBA.Object`,
  - Côté serveur : `org.omg.PortableServer.Servant`.

Mapping IDL/Java (1)

- Alias
  - pas d'alias en Java : utilisation du type réel
- Constantes IDL = constantes Java
- Structure IDL = classe Java
  - non extensible : *final*
  - chaque champ est représenté par un attribut *public*
  - un constructeur par défaut et un constructeur initialisant tous les champs
  - La traduction d'une structure entraîne la génération d'un Holder et d'un Helper

Mapping IDL /Java : exemple

```
//IDL
// struct Personne {
    string nom;
    string prenom;
}; //
//Java généré par le compilateur JavaIDL
public final class Personne {
  public String nom;
  public String prenom;

  public Personne() {}
  public Personne(String n, String p) {
   nom = n;
   prenom = p;
  }
}// class Personne
```

ENSIAS

Mapping IDL /Java : exemple

```
// utilisation de la structure

Personne ma_personne = new Personne();
ma_personne.nom= "Duval";
ma_personne.prenom= "Jean";
…
```

ENSIAS

Mapping IDL/Java (2)

- Enumération IDL :
  - Java ne supporte pas le type enum génération d'une classe Java (*final class*)
  - A chaque membre de l'énumération : 2 éléments de type *public static*
    - Champ (désigné par l'id du membre précédé par _)
    - Fonction qui porte l'id du membre et qui renvoie sa valeur
  - Une énumération dispose aussi de 2 méthodes :
    - Value : retourne la valeur interne du membre sous forme d'entier
    - From_int : retourne un membre à partir de la valeur interne donnée

Mapping IDL /Java : exemple 1

//IDL

enum Couleur {Rouge, Vert, Bleu} ;

//Java

```
Couleur ma_couleur;
ma_couleur = Couleur.from_int(0); // rouge
…
switch (ma_couleur.value() ) {
  case Couleur._Vert :
         System.out.println("vert");
         break;
         …
```

Mapping IDL /Java : exemple 2

//IDL

// enum Mois {Jan, Fev, … Dec}

//Java généré par le compilateur JavaIDL

```
public final class Mois {
  private int _value ;
  private static int _size=12;
  private static Mois[] _array = new Mois[_size];

  public static final int _jan=0;
  public static final Mois Jan = new Mois(_Jan);
  …

  public static final int _dec=11;
  public static final Mois Dec = new Mois(_Dec);
```

Mapping IDL /Java : exemple 2

// suite enum Mois {Jan, Fev, … Dec}

```
  public int value() { return _value ; }
  public static Mois from_int (int value)
  {
   if (value >= 0 && value <_size)
         return _array[value];
   else
         throw new org.omg.CORBA.BAD_PARAM () ;
  }
  private Mois(int value)
  { _value = value ;
   _array[_value] = this;
  }
}//class Mois
```

Des classes MoisHolder et MoisHelper sont aussi générées par le compilateur IDL

Mapping IDL /Java : exemple

// utilisation de l'énumération

```
Mois m=Mois.Janvier;
switch (m.value()) {
  case Mois._Jan: … ;
   …
  case Mois._Dec: … ;
};

m = Mois.from_int(7);
if (m==Mois.Aout) … ;
```

ENSIAS

Mapping IDL /Java : exemple

//IDL

// struct Date {Jour jour; Mois mois; Annee annee}

//Java généré par le compilateur JavaIDL

```
public final class Date {
  public short jour;
  public Mois mois;
  public short annee;

  public Date() {}
  public Date(short j, short m, short a) {
   jour = j;
   mois = m;
   annee = a;
  }
}// class Date
```

ENSIAS

Mapping IDL /Java : exemple

// utilisation de la structure

```
Date d = new Date();
d.jour = (short) 31 ;
d.mois = Mois.Dec ;
d.annee = (short) 2000;

d = new Date((short) 1, Mois.Jan, (short) 2001);
```

ENSIAS

Mapping IDL/Java (3)

- Tableaux et séquence IDL : tableaux Java
  - contrôle d'accès par Java
  - contrôle à l'emballage par CORBA
    - taille différente de la taille déclarée ou supérieure au maximum
    - exception : Corba::Marshal
  - la séquence peut être bornée ou non : on peut omettre la taille du tableau à sa déclaration
  - le mapping java d'une séquence : similaire à un tableau
  - des classes Holder et Helper sont également générées
- Exceptions : classes Java
  - similaire à la projection d'une structure IDL
  - Héritage
    - Classes exceptions utilisateur héritent de org.omg.CORBA.UserException
    - Classes exceptions système héritent de java.lang.RuntimeExceptions

Mapping IDL /Java : exemple

// IDL

// Tableau

```
typedef short CodeBanque[3];
interface agence {
  attribute string adresse;
  attribute CodeBanque code;
};
```

// Java

```
public interface agenceOperations {
  String adresse();
  void adresse(String newAdresse);
  short[] code();
  void code(short[] newCode);
}
```

Les classes AgenceHolder et AgenceHelper sont aussi générées.

ENSIAS

Les constantes

- La traduction d'une constante diffère selon la place qu'elle occupe dans une définition IDL :
  - Dans une interface, on traduit une constante en un type «public static final»

```
interface Exemple {
  const float Pi = 3.14;
};
```

```
public interface Exemple {
public static final float Pi = (
                    float )(3.14);
 }
```

  - En dehors d'une interface, la constante est traduite en une interface qui dispose d'un champ « value » de valeur celle de la constante.

```
module Exemple {
 const float Pi = 3.14;
};
```

```
package Exemple;
public interface Pi {
  public static final float
      value = (float)(3.14);
}
```

Synthèse
Mapping des types IDL

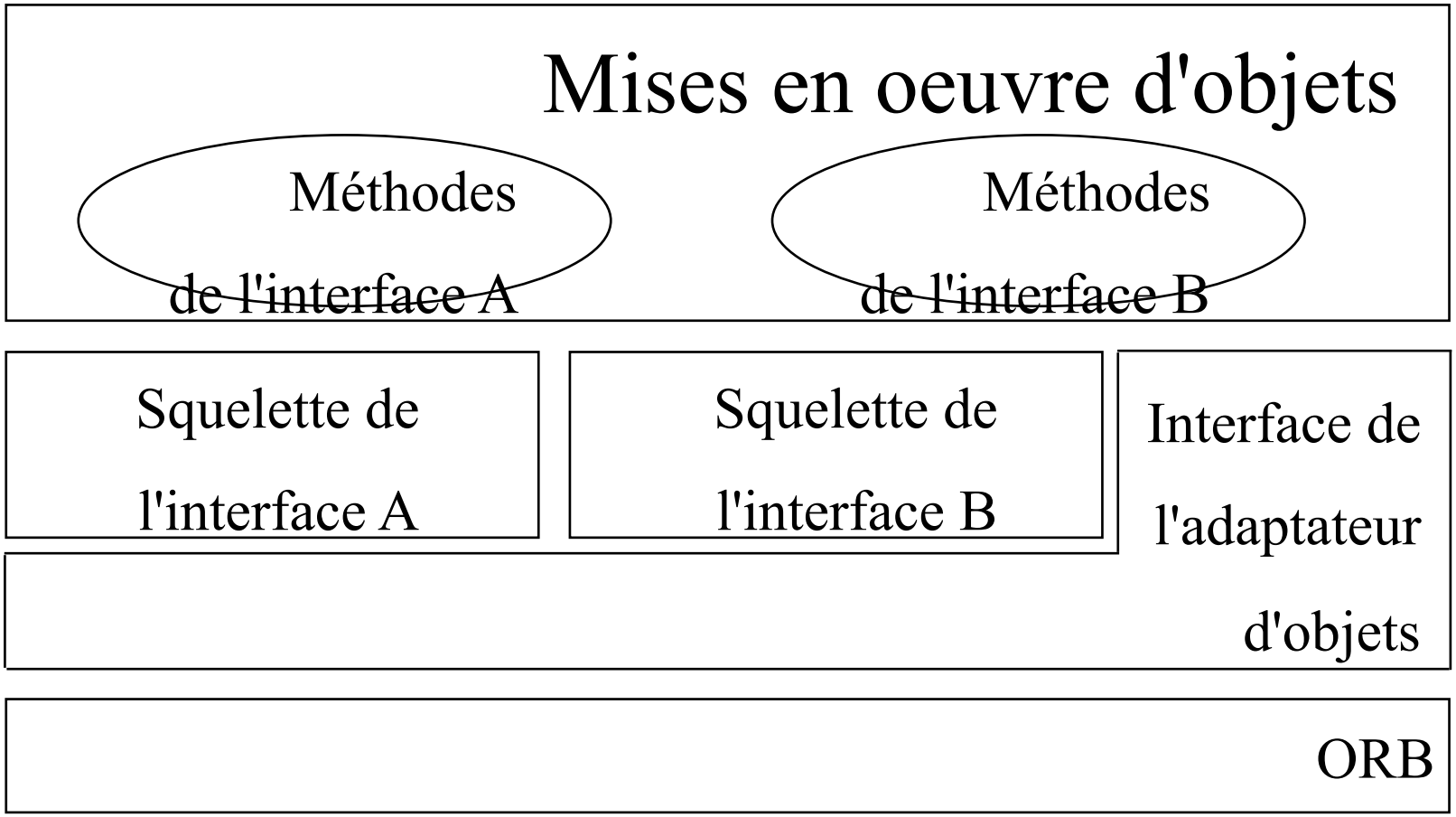
IDL Type	Java	
module	package	
interface	interface	
const	static final	
enum, struct, union	Classes Java ad hoc	
sequence	tableaux Java	
tableaux IDL	tableaux Java	
typedef	class	
exception	sous-type de java.lang.exception	
[readonly] attribute	méthode Java de lecture / MAJ	
opération d'une interface	méthode Java	

Conclusion mapping Java

- Règles de projection de l'OMG IDL vers Java variables selon :
  - type à longueur variable ou à longueur fixe,
  - sens de passage de paramètres,
  - côté client ou serveur.
- Règles additionnelles pour les autres types d'interface (locale, abstraite) et objets transmissibles, la gestion des exceptions, la gestion du contexte d'exécution, le mode d'invocation dynamique et le POA.
- Règles inverses de Java vers l'OMG IDL.

VII. L'adaptateur d'objets POA

Vue de l'adaptateur d'objets

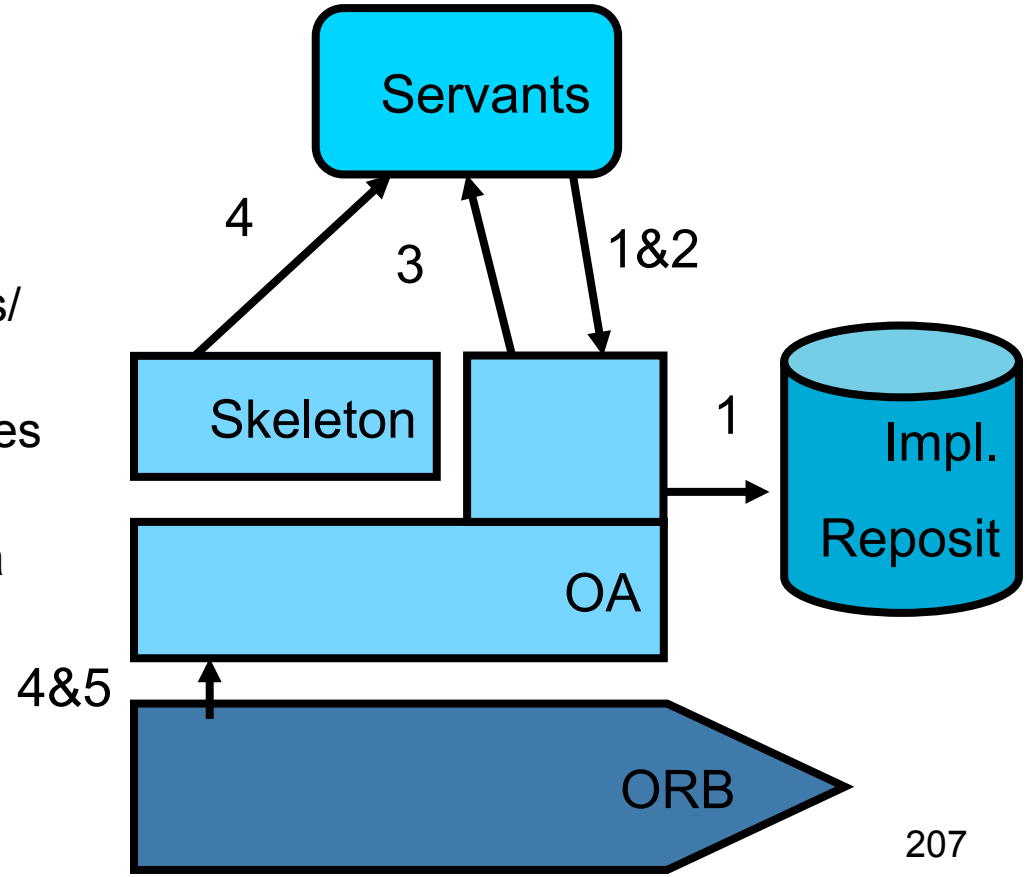


Adaptateur d'objets (AO)

- AO est un moyen pour l'objet d'implantation d'accéder aux services de l'ORB.

Fonctions :

1. **Enregistrement** des obj. d'implantation (servants)
2. **Gestion des références d'objet** : génération et correspondance réf. d'objets/ servants
3. **Activation** / désactivation des servants
4. **Invocation** de méthodes via skeleton
5. **Authentification** client et contrôle d'accès



BOA et POA

- Différents types de AO : BOA Basic Object Adapter, POA Portable Object Adapter, LOA Library Object Adapter , OODA Object-Oriented Database Adapter
- BOA : fournit les fonctions de base : génération et interprétation de références d'objets, invocation de méthodes, activation et désactivation des mises en oeuvre (CORBA < 2.2)
- POA
  - Offre la portabilité des applications.
  - Autorise plusieurs instances de POA à coexister dans un même serveur.
  - Fournit la possibilité d'une activation implicite des objets.
  - Offre une meilleure maîtrise du processus de traitement d'une requête au développeur.

POA : quelques définitions...

- Objet Corba : possède une identité (réf. Corba)
- Client : entité qui soumet une requête à un objet Corba via sa référence
- Serveur : entité dans laquelle un objet Corba est mis en œuvre.
- Servant : entité qui implante un objet corba et évolue dans le contexte du serveur.
  - S'il est persistant, un objet Corba peut être servi successivement par des servants différents ayant des durées de vie limitées

Architecture : l'arborescence de POAs

- Un POA peut posséder des POAs fils. L'ensemble donne une arborescence dont la racine est appelée « RootPOA ».

- Le RootPOA est accessible via l'opération :

```
resolve_initial_reference("RootPOA");
```

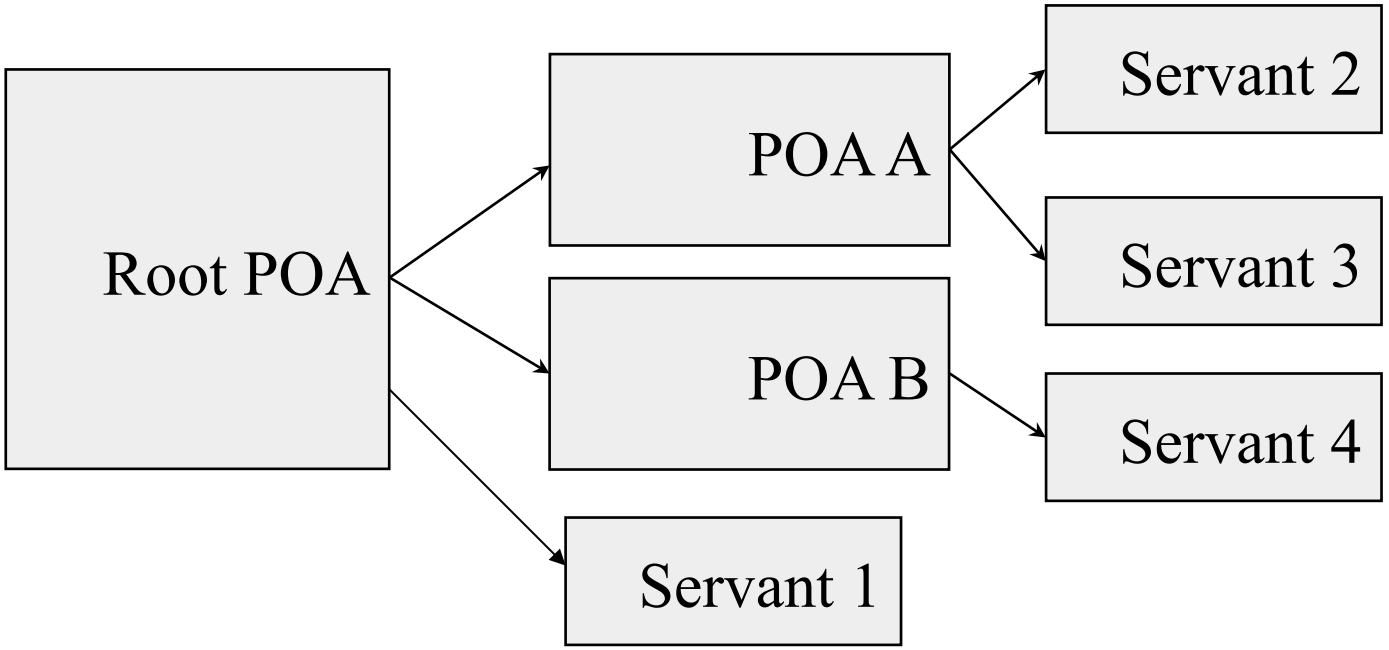
- Les autres POAs sont créés à partir de leur père via l'opération :

```
create_POA(nom, gérant, politiques);
org.omg.PortableServer.POA  filsPOA =
rootPOA.create_POA("persistant",gerant, politiques);
```

ENSIAS

Rôle d'un POA

- Chaque POA gère un ensemble de servants fournis par le développeur.



- Les règles de gestion propres à un POA sont définies par la valeur de ses paramètres (ses politiques).

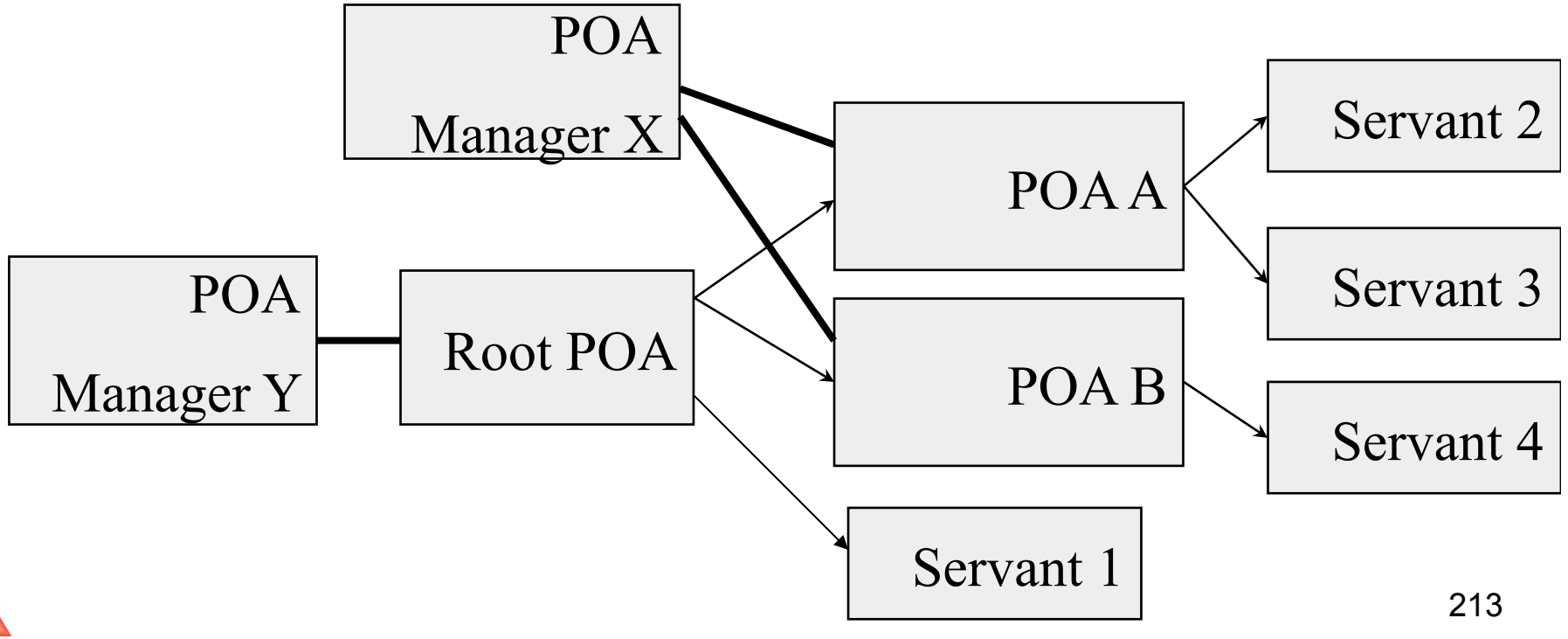
Co-existence de plusieurs POAs

- La création d'un POA spécifique à chacune des interfaces supportées par un même serveur permet d'adapter un ensemble de politiques à chaque interface considérée.

- La hiérarchie des POAs détermine l'ordre dans lequel les POAs sont détruits quand le serveur est arrêté (à partir des feuilles jusqu'à la racine).

Gérant de POA (1)

- Chaque POA est associé à un gérant de POA (POA Manager) qui peut être partagé entre plusieurs POAs.
- Le gérant de POA correspond à un point d'accès réseau et permet de gérer le flux des requêtes provenant des clients.

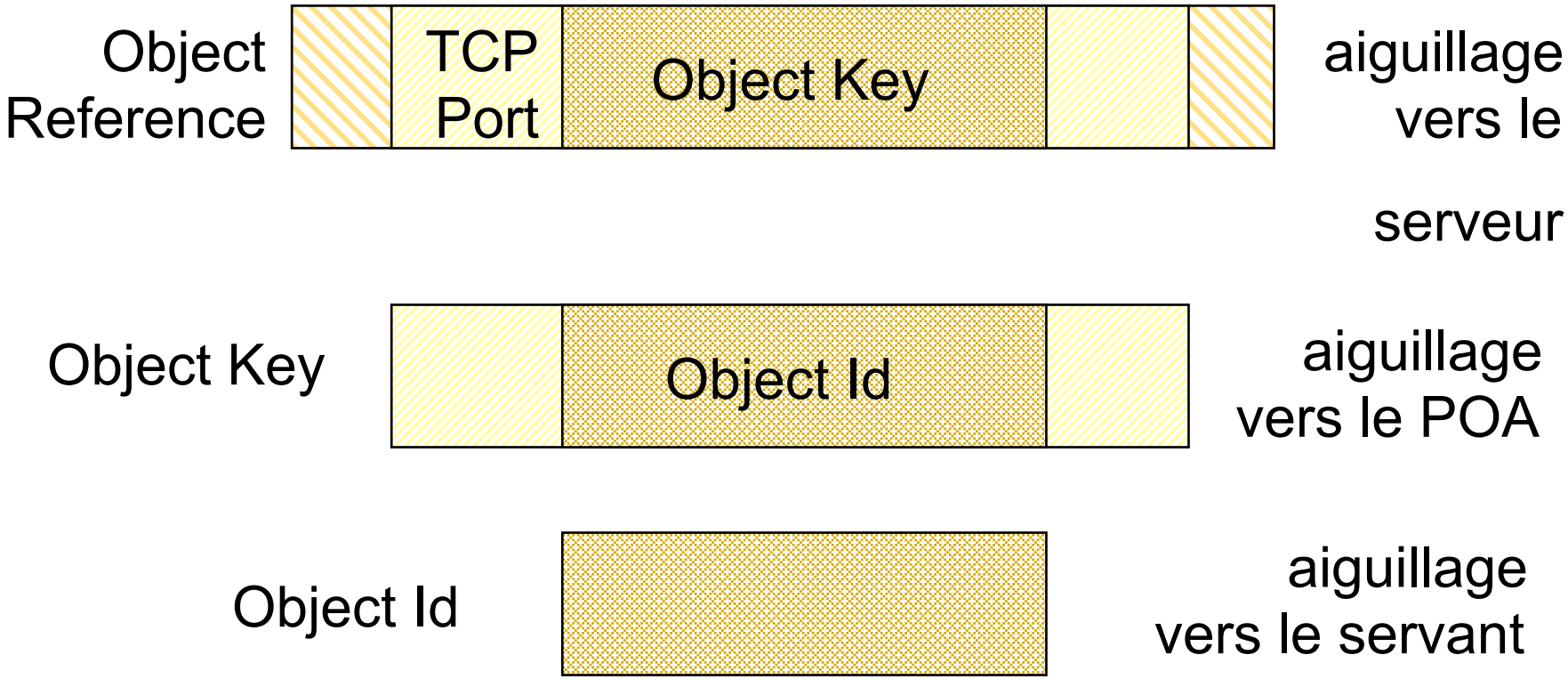


Gérant de POA (2)

- Un gérant de POA :
  - Est créé qd un POA est créé et est auto. associé à ce POA.
  - Est détruit qd tous ses POAs associés le sont.
- Il peut être dans l'un des états suivants :
  - holding (état initial), active, discarding ou inactive (état final).
  - Selon son état, le gérant de POA :
    - soit transmet les requêtes directement au POA intéressé (active),
    - soit les détruit (discarding),
    - soit les met en file d'attente du POA cible (holding)
    - soit ignore la req. en déclarant l'objet Corba inaccessible (inactive).

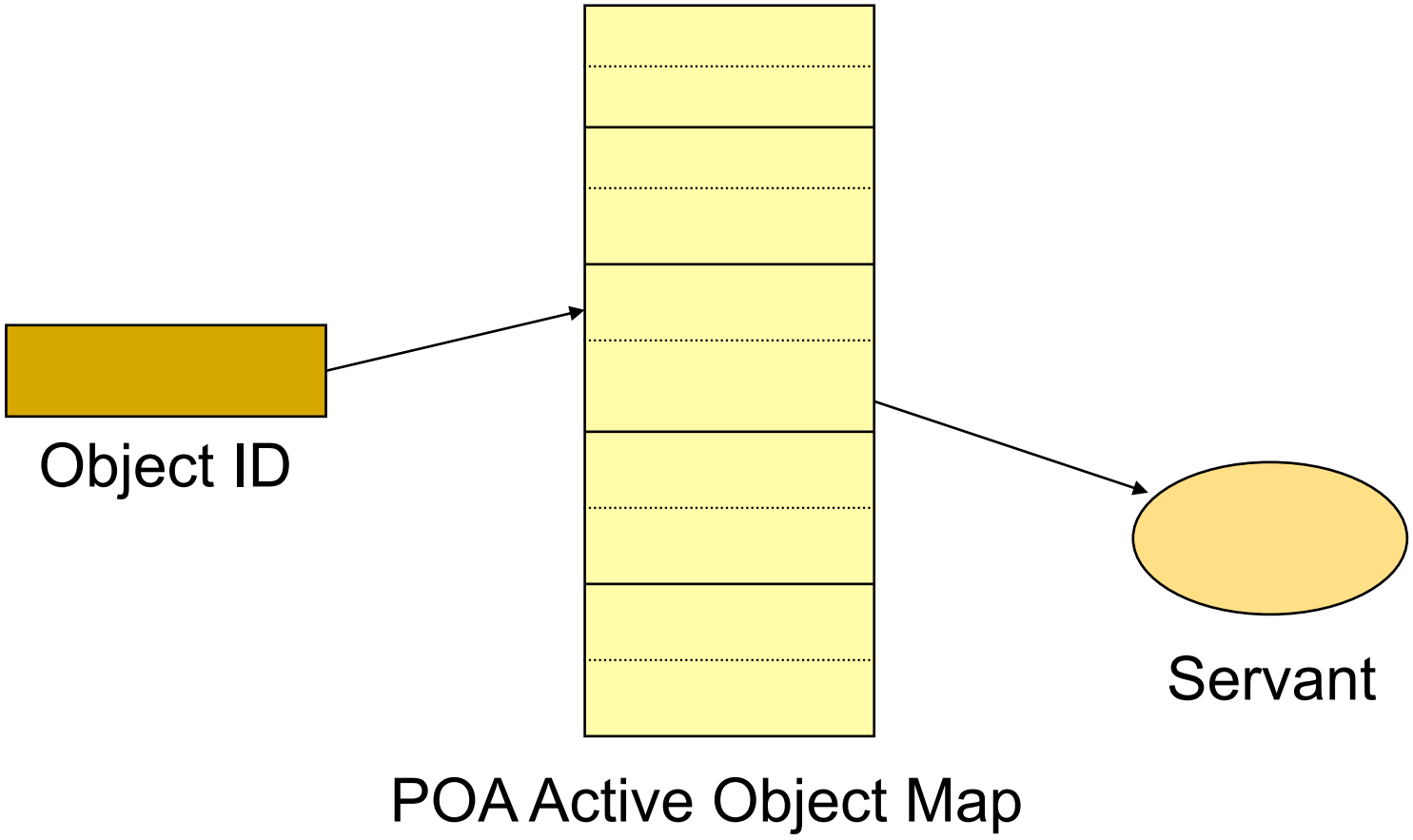
Routage des requêtes (1)

- Une fois transmise au POA concerné, la requête est aiguillée vers le servant capable de la traiter



Routage des requêtes (2)

- Object Id identifie de manière unique l'objet Corba au sein d'un POA. L'association se fait via une table active-object-map.



Co-existence de plusieurs gérants de POA au sein d'un même serveur ?

- Un gérant de POA peut être affecté à un groupe d'objets dont les règles d'accessibilité sont les mêmes.

- L'utilisation simultanée de plusieurs gérants de POA permet, par exemple, de suspendre le traitement de requêtes pour un groupe d'objets sans affecter les autres objets du serveur.

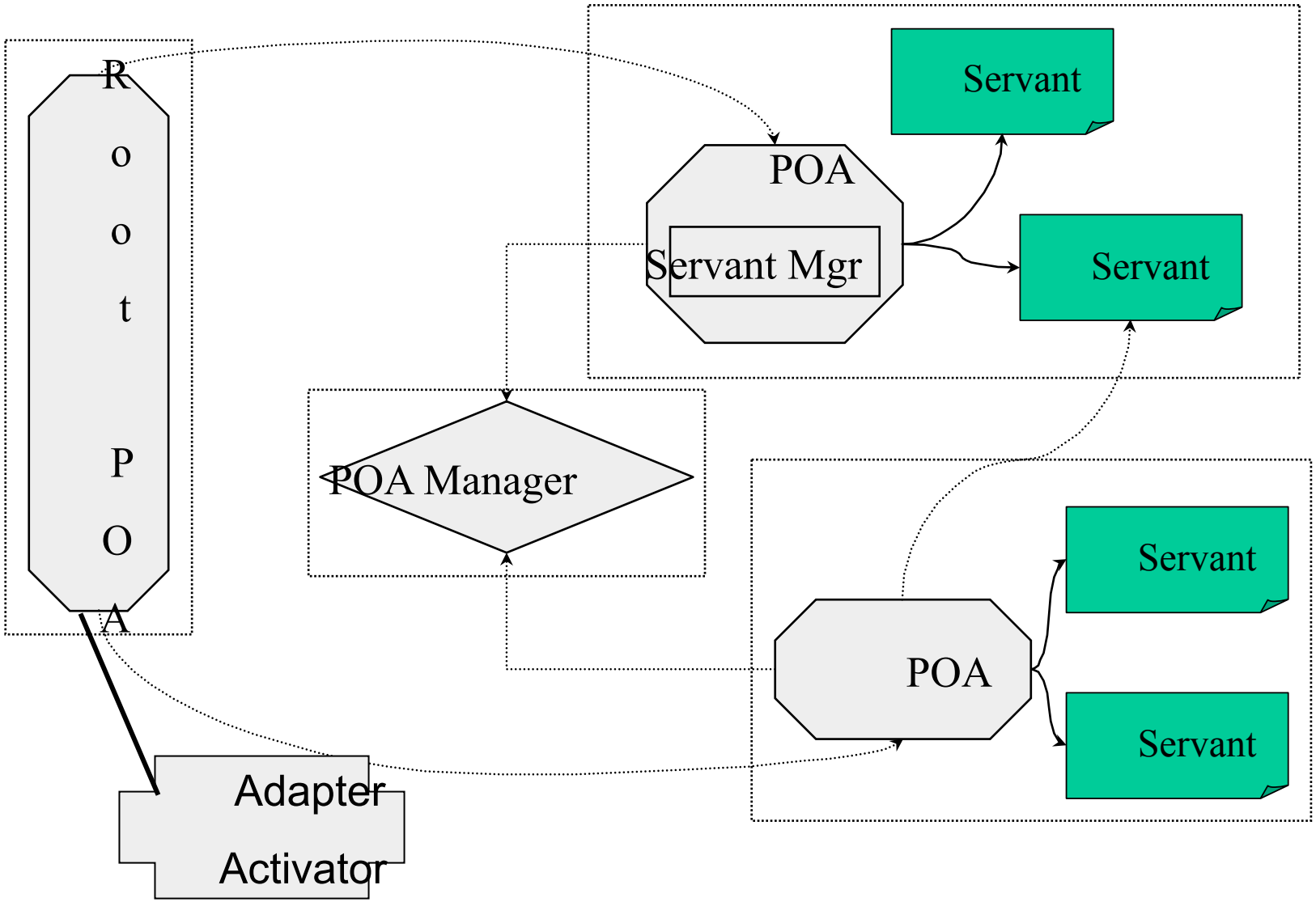
Gestionnaire de Servant

- Des gestionnaires de servant (ServantManager) peuvent être associés aux POAs.
- Un gestionnaire de servant est invoqué par un POA pour activer un servant à la demande.
- Une application qui activerait tous ses servants à l'initialisation ne nécessite pas de gestionnaire de servants.

Activateur d'adaptateur

- Des activateurs d'adaptateur (AdapterActivator) peuvent être associés aux POAs.
- Un activateur d'adaptateur permet la création de POA fils à la demande.
- Une application qui fournit tous les POAs nécessaires dès son initialisation ne nécessite pas d'activateur d'adaptateur.

Architecture globale du POA



Politiques d'un POA

- Le comportement d'un POA est régi par le biais de diverses politiques (policy).
- On dénombre pas moins de sept politiques avec chacune deux ou trois valeurs possibles :
  - Lifespan,
  - Id Assignment,
  - Id Uniqueness,
  - Implicit Activation,
  - Request Processing,
  - Servant Retention,
  - Thread.

Durée de vie des objets

- Description :
  - Détermine si un objet CORBA peut survivre au delà du processus qui l'a créé (plus précisément si sa référence d'objet est persistante ou temporaire).
- Type : Lifespan Policy,
- Valeurs :
  - (par défaut) TRANSIENT,
  - PERSISTENT.

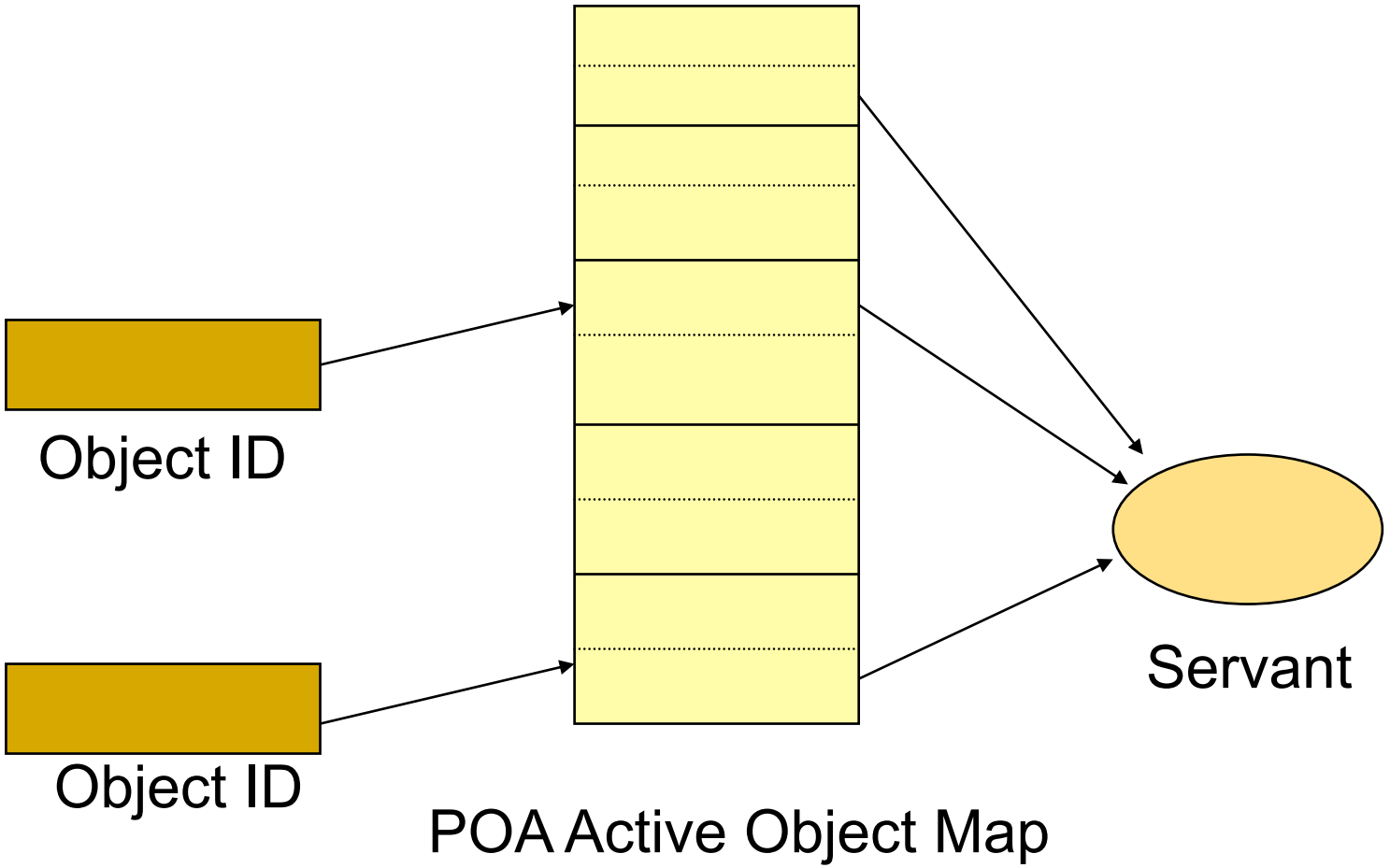
Identification des objets

- Description :
  - Détermine si c'est l'application ou le POA qui crée la partie Object_ID des références d'objet.
- Type : IdAssignment Policy,
- Valeurs :
  - (par défaut) SYSTEM_ID,
  - USER_ID.

Correspondance entre objets et servants

- Description :
  - Détermine si un même servant est dédié à un objet CORBA ou s'il peut servir pour plusieurs objets (dans ce dernier cas, la table active ObjectMap possède plusieurs entrées distinctes qui pointent vers une même sortie correspondant au servant partagé)
- Type : IdUniqueness Policy,
- Valeurs :
  - (par défaut) UNIQUE_ID,
  - MULTIPLE_ID.

Partage d'un même servant



Activation implicite

- Description :
  - Détermine si le servant doit être activé explicitement par l'application ou s'il l'est automatiquement dès que l'application crée une référence sur ce servant.
- Type : ImplicitActivation Policy,
- Valeurs :
  - (par défaut) NO_IMPLICIT_ACTIVATION,
  - IMPLICIT_ACTIVATION.

Correspondance entre requêtes et servants

- Description :
  - Détermine si les requêtes sont dirigées vers les servants par le POA (basé sur une table des objets actifs) ou par l'application (via un servant par défaut ou via un gestionnaire de servants).
- Type : RequestProcessing Policy,
- Valeurs :
  - (par défaut) USE_ACTIVE_OBJECT_MAP_ONLY,
  - USE_DEFAULT_SERVANT,
  - USE_SERVANT_MANAGER.

Maintien des servants en mémoire

- Description :
  - Détermine si un servant doit être conservé en mémoire à tout moment (avec une entrée dans la table des objets actifs - Retain) ou s'il peut être instancié à la demande lors de l'arrivée d'une requête le concernant.

- Type : ServantRetention Policy,

- Valeurs :
  - (par défaut) RETAIN,
  - NON-RETAIN.

Allocation de requêtes aux threads

- Description :
  - Détermine si l'allocation des requêtes aux threads est laissée à la charge de l'ORB (ce qui suppose que les méthodes de l'application sont réentrantes et exécutables de façon concurrente) ou séquentialisée par le POA (un seul thread - distingué ou quelconque - aiguille les requêtes).
- Type : Thread Policy,
- Valeurs :
  - (par défaut) ORB_CTRL_MODEL,
  - SINGLE_THREAD_MODEL,
  - MAIN_THREAD_MODEL.

Politiques du Root POA

- Lifespan TRANSIENT
- IdAssignment SYSTEM_ID
- IdUniqueness UNIQUE_ID
- ImplicitActivation **IMPLICIT_ACTIVATION**
- RequestProcessing USE_ACTIVE_OBJECT_MAP_ONLY
- ServantRetention RETAIN
- Thread ORB_CTRL_MODEL

ENSIAS

Autres usages

- 288 combinaisons théoriquement possibles.
- Mais :
  - certaines combinaisons sont interdites car il existe des incompatibilités,
  - certains combinaisons sont recommandées car elles évitent des problèmes potentiels.

Quelques interdictions

- Correspondance objets/servants/object ID :
  - L'usage combiné des valeurs « UNIQUE_ID » et « NON_RETAIN » peut conduire à utiliser le même servant pour des requêtes concurrentes sur des objets différents.
- Correspondance objets/servants/requêtes :
  - La valeur « NON_RETAIN » impose l'usage de « USE_DEFAULT_SERVANT » ou de « USE_SERVANT_MANAGER ».
  - La valeur « USE_ACTIVE_OBJECT_MAP_ONLY » ou de « IMPLICIT_ACTIVATION » impose l'usage de « RETAIN ».
  - La valeur « USE_DEFAULT_SERVANT » impose l'usage de « MULTIPLE_ID».

Quelques scénarios d'usage

- Objets persistants
- Activation de servants à la demande :
  - ServantActivator.
- Partage d'un même servant pour plusieurs objets :
  - ayant la même interface IDL,
  - ayant des interfaces IDL différentes (DSI).

Interfaces du POA

- POA
- POA Manager
- Servant
- POA Current
- AdapterActivator
- …

⇒ sont regroupées dans un module « PortableServer »

Obtenir le POA racine et son gérant

```
    org.omg.CORBA.ORB orb = null;
    // Specifique ORBACUS 4.0.4
    java.util.Properties proprietes = System.getProperties();
    proprietes.put("ooc.orb.oa.port", "5555");
    // Initialisation de l'ORB
    orb = ORB.init(args, proprietes);
    // Initialisation du POA
    org.omg.PortableServer.POA rootPOA =
        org.omg.PortableServer.POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));
    // Initialisation du gerant de POA
    org.omg.PortableServer.POAManager gerant = rootPOA.the_POAManager();
…
```

Créer un POA avec des politiques

```
…
 org.omg.CORBA.Policy[] politiques;
        politiques = new org.omg.CORBA.Policy[4];
        politiques[0] = rootPOA.create_lifespan_policy(
            org.omg.PortableServer.LifespanPolicyValue.PERSISTENT);
        politiques[1] = rootPOA.create_id_assignment_policy(
            org.omg.PortableServer.IdAssignmentPolicyValue.USER_ID);
        politiques[2] = rootPOA.create_thread_policy(
            org.omg.PortableServer.ThreadPolicyValue.SINGLE_THREAD_MODEL);
        politiques[3] = rootPOA.create_implicit_activation_policy(
            org.omg.PortableServer.ImplicitActivationPolicyValue.NO_IMPLICIT_ACTIVATION);
  org.omg.PortableServer.POA filsPOA =
            rootPOA.create_POA("persistant", gerant, politiques);
…
```

Activer un servant explicitement

```
…
  // Creation de l'objet d'implantation
     String nom_compte = "Compte";
      impl_compte_heritage compteImpl = new impl_compte_heritage(nom_compte);
  // Creation de la reference CORBA
     org.omg.PortableServer.Servant unCompte = compteImpl;
     byte[] oidCompte = nom_compte.getBytes();
     filsPOA.activate_object_with_id(oidCompte, unCompte);
     org.omg.CORBA.Object objCompte = filsPOA.id_to_reference(oidCompte);
…
```

Vocabulaire Corba

- Client
  - application invoquant des objets à travers le bus CORBA
- Référence d'objet
  - structure désignant l'objet CORBA contenant l'information nécessaire pour le localiser sur le bus
- Requête
  - mécanisme d 'invocation d'une opération ou d'accès à un attribut
- Bus CORBA
  - achemine les requêtes du client vers l'objet en masquant les problèmes d'hétérogénéité (lang., systèmes, matériels, etc.)

Vocabulaire Corba

- Objet CORBA
  - composant logiciel cible, entité virtuelle gérée par le bus CORBA
- Activation
  - processus d'association d'un objet d'implantation à un objet CORBA
- Implantation de l'objet (servant)
  - entité codant l'objet CORBA à un instant donné et gérant un état temporaire de l'objet
  - au cours du temps, un objet CORBA peut être associé à différentes implantations

Vocabulaire Corba

- Code d'implantation
  - ensemble des traitements associés à l'implantation des opérations de l'objet CORBA
  - exemple : une classe C++ ou Java, ou un ensemble de fonctions C
- Serveur
  - structure d'accueil des objets d'implantation et des exécutions des opérations (ex: un processus Unix)

CORBA : Conclusion

- Solution non propriétaire,
- Ouverture vers d'autres mondes et vers l'existant,
- Libre choix des technologies d'implantation et donc plus de portabilité,
- Plusieurs implantations gratuites et/ou open source,
- Une vision efficace et évolutive pour le développement des applications distribuées hétérogènes,