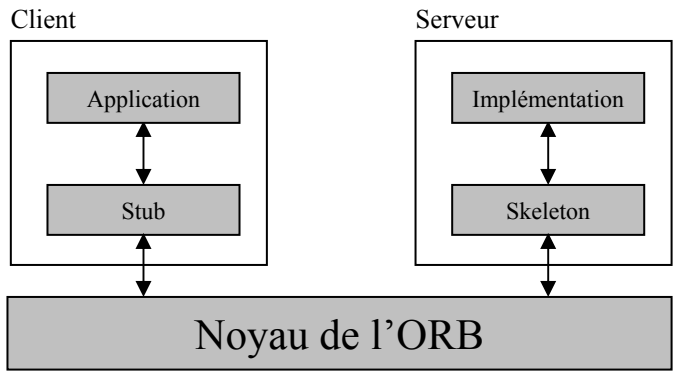
CORBA

Cycle de développement

Le cycle de développement (1/3)

❑ On s'intéresse au développement selon le mode statique,
c'est à dire en utilisant les talons (stub, skeleton, …) générés par le compilateur IDL

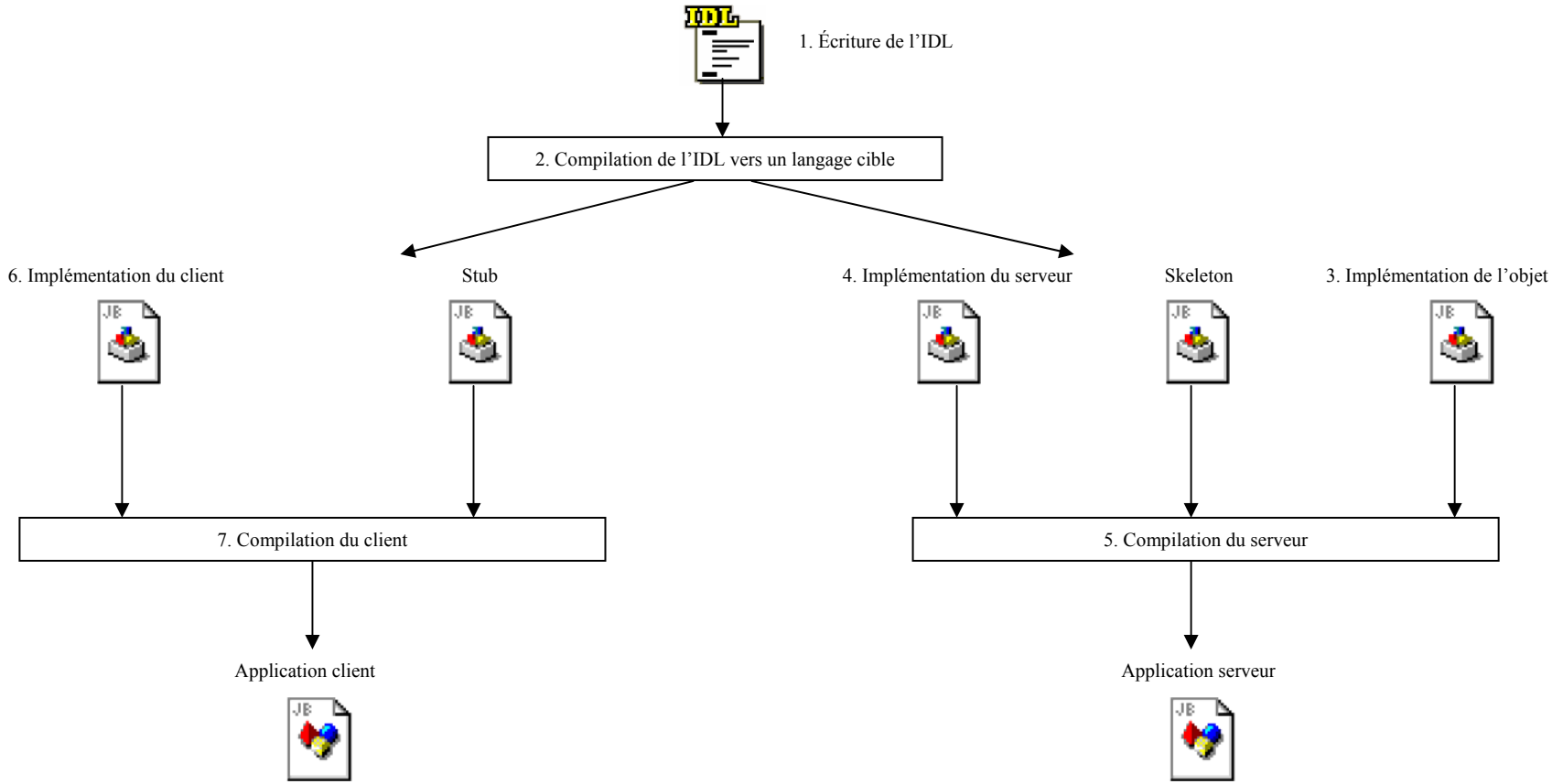


❑ Le cycle de développement se décompose en 7 étapes

1. Écriture de l'IDL
2. Compilation de l'IDL vers un langage cible
3. Implémentation de l'objet
4. Implémentation du serveur
5. Compilation du serveur
6. Implémentation du client
7. Compilation du client

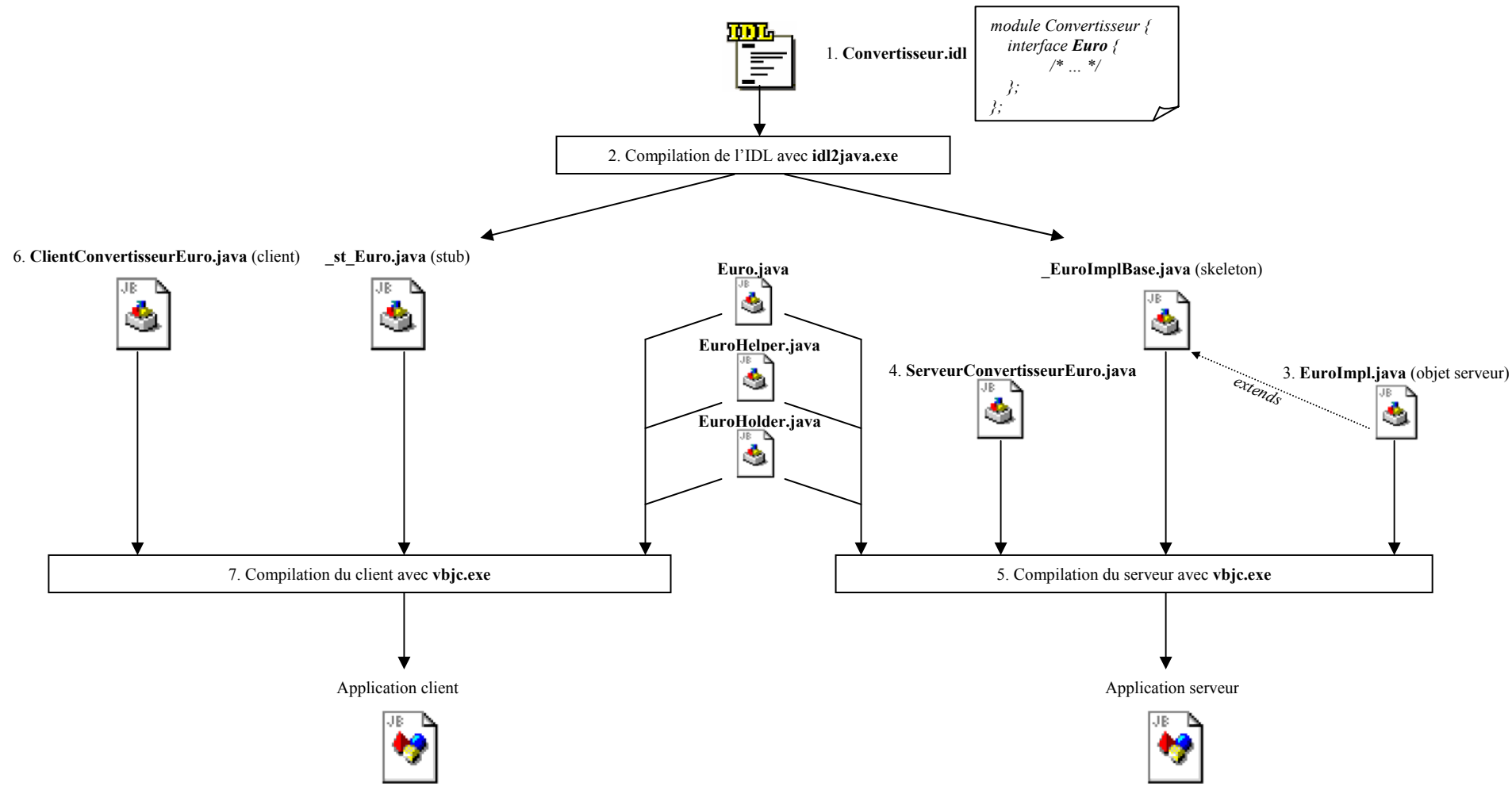
Le cycle de développement (2/3)

- On s'intéresse à la génération de code depuis des spécifications IDL



Le cycle de développement, d'IDL vers Java (3/3)

- La génération de code depuis l'IDL vers la langage cible Java se caractérise ainsi



Écriture de l'IDL (1/7)

- L'écriture permet de définir les différentes interfaces des composants

1. **Convertisseur.idl**

```
module Convertisseur {
   interface Euro {
      attribute double  taux;
      attribute string devise;
      void start();
      double toEuro(in double devise);
      double toDevise(in double euro);
   };
};
```

- Quelques règles de passage lors du passage de l'IDL vers Java

  - Un *module* IDL devient un *package* Java
  - Une *interface* IDL devient une *interface* Java
  - Un *attribute* IDL génére un attribut private et deux accesseurs Java
  - Une méthode IDL devient une méthode publique Java
  - …

Compilation de l'IDL vers Java (2/7)

- La compilation IDL vers Java permet de générer les différents talons (stub, skeleton, …)

```
C:\>idl2java Convertisseur.idl
```

- Les principaux fichiers générés sont les suivants

Fichier .java généré	Description du contenu
Euro.java	Interface Java définissant les services de l'objet Euro
EuroHelper.java	Classe qui dispose de méthodes permettant notamment de caster vers le type Euro
EuroHolder.java	Classe qui est utilisée lors du passage d'instances de l'objet Euro comme paramètre ou résultat d'une méthode
_EuroImplBase.java	Skeleton de l'objet, qui se charge des requètes en provenace du stub et les transmet à l'objet Euro
_st_Euro.java	Stub installé sur le client qui est appelée lors de l'invocation par le client de l'objet distant Euro

Implémentation de l'objet (3/7)

- On implémente l'objet en Java

3. **EuroImpl.java**

```
public class EuroImpl extends Convertisseur._EuroImplBase  {

  // construction des objets
  public EuroImpl(java.lang.String name) {
    super(name);
    _taux=6.55695;
    _devise="Francs";
  }
  public EuroImpl() {
    super();
    _taux=6.55695;
    _devise="Francs";
  }
  // taux de conversion
  private double _taux;

  public void taux(double taux) {
    System.out.println(taux);
      _taux=taux;
  }
  public double taux() {
    return _taux;
  }

// devise utilisée
  private String _devise;

  public void devise(java.lang.String devise) {
      _devise=devise;
  }
  public java.lang.String devise() {
      return _devise;
  }
  // implémentation des opérations de conversion
  public double toEuro(double devise) {
    return devise / _taux;
  }
  public double toDevise(double euro) {
    return euro * _taux;
  }
}
```

A noter que la classe d'implantation hérite du skeleton

- L'implémentation peut être faite par héritage, comme présenté ci-dessus, ou par délégation, qui permet à l'objet implémenté de conserver son héritage libre

Implémentation du serveur (4/7)

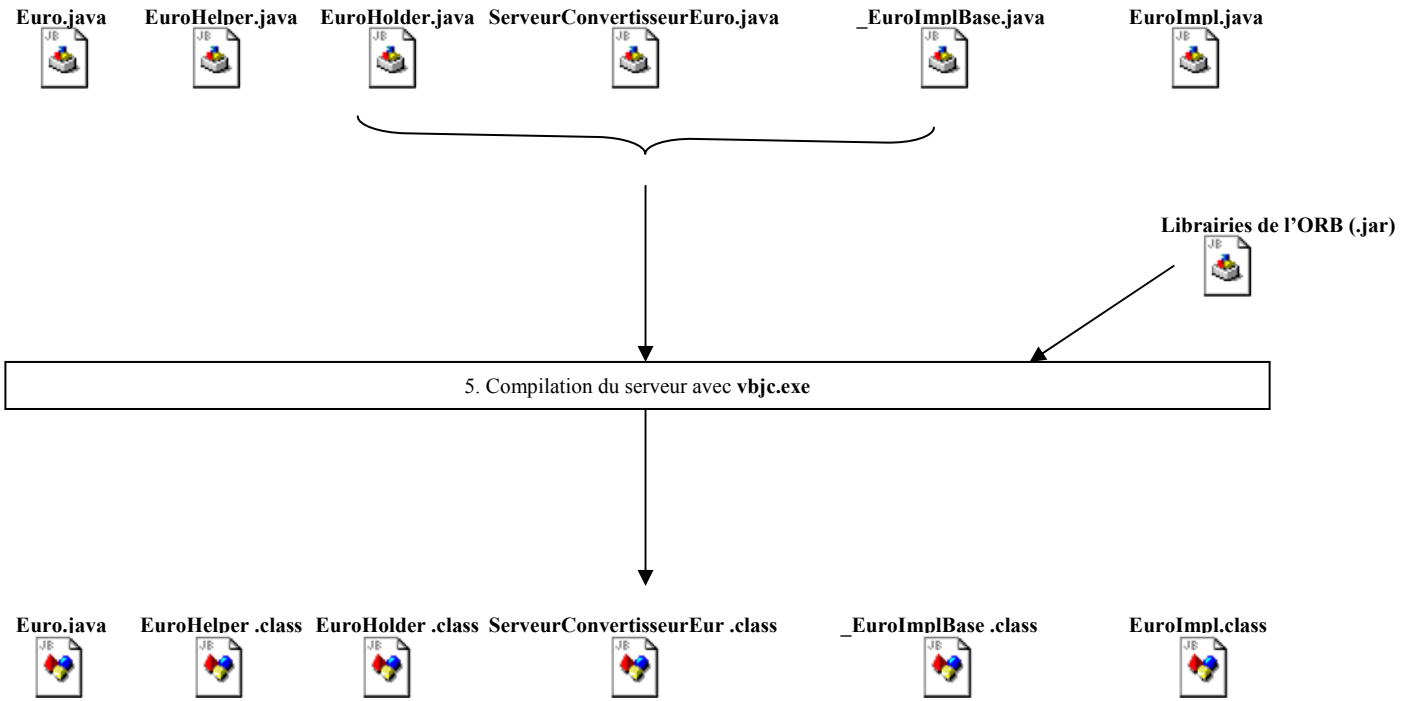
- On implémente en Java le serveur qui contiendra l'objet

4. **ServeurConvertisseurEuro.java**

```
import org.omg.CosNaming.*;
import com.visigenic.vbroker.orb.*;

public class ServeurConvertisseurEuro {

public static void main(String[] args) {
try {
      //intialisation de l'orb
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

      //initialiser le BOA
      org.omg.CORBA.BOA boa =  ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();

      //Création du servant
      EuroImpl monEuro = new EuroImpl("Convertisseur");

      //enregistrement de l'objet au niveau du BOA
      boa.obj_is_ready(monEuro);

      System.out.println(monEuro + " is ready.");

      // Mise en attente de requete
      boa.impl_is_ready();;

      //for(;;);
   }
   catch (Exception e) {e.printStackTrace();
   }
}
}
```

Compilation du serveur (5/7)

- On génère le code exécutable du serveur (.java vers .class)



Implémentation du client (6/7)

- On implémente en Java le client qui utilisera l'objet distant Euro

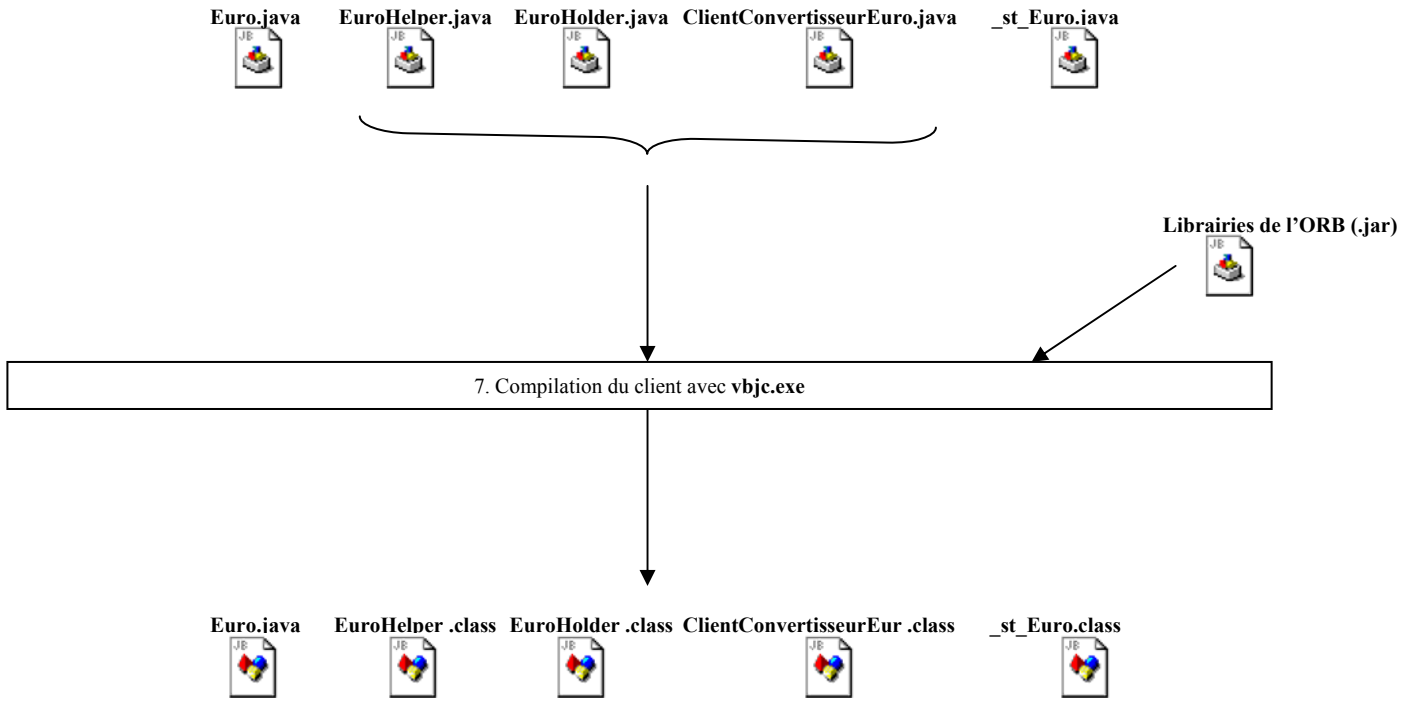
6. **ClientConvertisseurEuro.java**

```
//réaliser les import ...

class InterfaceFrame extends JFrame
  implements DocumentListener, ActionListener
{  public InterfaceFrame()
  { /* l'interface graphique en Java avec Swing et Awt ...*/
  }

public class ClientConvertisseurEuro {
public static Convertisseur.Euro monEuro;

public static void main(String args[]) {
org.omg.CosNaming.NamingContext root;
try {
      //intialisation de l'orb
      ORB orb = org.omg.CORBA.ORB.init(args,null);

      //connexion à l'objet
     monEuro =  Convertisseur.EuroHelper.bind(orb,"Convertisseur");

      //appel de l'interface graphique windows
      JFrame frame = new InterfaceFrame();
      frame.show();

     }
     catch (Exception e) {e.printStackTrace();}
}//fin du main

}fin du client
```

A noter que la connexion à l'objet se fait dans cet exemple avec la méthode propriétaire bind() c'est à dire que cette méthode dépend de l'ORB utilisé (VisiBroker par exemple). Il existe d'autres techniques de connexion plus génériques (service d'annuaire CORBA, appelé CosNaming)

Compilation du client (7/7)

- On génère le code exécutable du client (.java vers .class)



Attachement du client au serveur (1/1)

- L'attachement du client au serveur répond à la problématique de l'accès aux implémentations d'objets depuis les programmes clients
  1. la méthode bind()
  2. l'IOR (Interoperable Object Reference)
  3. le CosNaming (service de nommage)
  4. l'URLNaming

Méthode	Description	Avantage(s)	Inconvénient(s)
bind()	Permet au client de s'attacher à l'implémentation d'un objet, grâce à la méthode *NomClasseHelper.bind(« Nom »)*	La simplicité de codage	La méthode est propriétaire donc le client devient dépendant de l'ORB
IOR	On associe l'IOR (représentation binaire de la référence de l'objet) à une chaîne de caractère via les méthodes object_to_string() et string_to_object(). Plus précisément, le serveur stocke l'IOR dans un fichier, le client récupère l'IOR puis la convertit vers le type de la référence souhaitée	La méthode est générique	L'accès à un fichier sur un volume partagé est nécessaire depuis le client
CosNaming	La transparence de la localisation devient totale avec la mise en place d'un annuaire qui gère les associations entre les noms logiques d'instances et leurs IORs	Ce service standard est disponible dans la plupart des implémentations de CORBA. L'emploi des IORs et l'organisation des références d'objets sont simplifiées	L'écriture du code se complique
URLNaming	L'idée est d'associer une URL à une IOR	La distribution d'applications sur l'Internet devient possible	La notion d'IOR est toujours visible

Exécution (1/2)

- L'exécution d'une application répartie CORBA (avec l'ORB VisiBroker) requiert le lancement de services utilitaires, le Smart Agent (obligatoire) et le service de nommage (facultatif)

- Le Smart Agent est un service dynamique et distribué de répertoires qui permet aux clients lors du bind() d'accéder aux implémentations des objets distants. Il faut donc activer au moins un Smart Agent dans le réseau local sur lequel on veut exécuter une application CORBA avec VisiBroker

```
C:\>osagent
```

- Le service de nommage doit éventuellement être activé pour pouvoir utiliser les services du CosNaming. Le service de nommage, qui est lui-même un objet CORBA, nécessite donc qu'un Smart Agent ait été activé avant son propre démarrage.

```
C:\>start nameserv root_context_name
```

ou

```
C:\>vbj com.inprise.vbroker.naming.ExtFactory
```

Exécution (2/2)

- L'exécution de l'application répartie **ConvertisseurEuro** présentée dans le cycle de développement requiert donc l'activation d'un Smart Agent, le lancement du serveur puis le lancement du client