TD 3 – CORBA

Kenneth VANHOEY
https ://dpt-info.u-strasbg.fr/~kvanhoey

1 Rappels

CORBA (Common Object Request Broker Architecture) est une norme qui permet de définir une interface de services disponibles à distance et indépendante d'un langage particulier. Pour cela, le langage déscriptif IDL (Interface Definition Langage) sert de support et permet une projection (traduction) dans les principaux langages objets. Cette norme permet l'interopérabilité entre programmes écrits dans différents langages.

1.1 Fonctionnement

Une interface définie en IDL est projetée dans un langage, c'est-à-dire que son équivalent est défini dans ce dernier. Le déroulement du développement et de l'exécution est alors similaire à celui de RMI, notamment le principe de transparence de l'appel pour le client et l'utilisation d'un stub et d'un squelette qui s'occupent de la sérialisation et de la transmission/réception des communications.

En CORBA, on appelle *servant* l'objet implantant une interface IDL. Un servant est *logé* sur une application *serveur*.

1.2 Développement & compilation

Le développement et la compilation d'un service CORBA se fait en plusieurs étapes. Tout le code qui suit forme un ensemble minimal. Toute classe importée, implémentée ou étendue l'est obligatoirement, toute exception gérée doit l'être.

1. **Développement** de l'interface de l'objet en langage IDL.

   ```
   // fichier I<classname>.idl
   interface I<classname>
   {
     <return_type_idl> myfunction1 ([<in/out/inout> <type_idl> arg]*) ;
     ...
   } ;
   ```

2. **Projection** de l'interface IDL en un langage donné (Java dans ce cours, avec l'outil *idlj*) :

   ```
   idlj -fall -oldImplBase I<classname>.idl
   ```

   Ceci crée les fichiers *_I<classname>ImplBase.java* (skeleton), *_I<classname>Stub.java (proxy)*, *I<classname>Helper.java* (assistant de transtypage entre un objet CORBA et un objet Java), *I<classname>Operations.java*, *I<classname>.java* (interfaces projetées en Java) et *I<classname>Holder.java* (pour le passage de paramètre de type *out* ou *inout*, voir plus bas).

3. **Programmation** d'une implémentation des méthodes correspondant à cette interface. Pour cela, il faut regarder le fichier *I<classname>Operations.java*, qui spécifie l'interface à implémenter.

   ```
   // fichier I<classname>Impl.java
   import org.omg.CORBA.* ;

   public class I<classname>Impl
   ```

```
  extends _I<classname>ImplBase // étends le skeleton
{
  // membres
  ...
  public I<classname>Impl() // constructeur
  {
    super() ;
  } ;

  public <return type java> myfunction1 ([<type_java> arg]*)
  {
    // corps
  }
  ...
}
```

4. **Développement** d'un programme serveur qui mettra à disposition l'objet :

```
import org.omg.CORBA.* ;

public class Serveur
{
  public static void main(String[] args)
  {
    try
    {
      //initialiser le bus ORB
      ORB orb = ORB.init(args, null ) ;

      // Déclarer l'objet
      I<classname>Impl myobj = new I<classname>Impl() ;
      // Déduire l'identifiant unique IOR
      String ior = orb.object_to_string( myobj ) ;
      // Afficher l'IOR
      System.out.println( ior ) ;

      orb.run() ;
    }
    catch( org.omg.CORBA.SystemException ex ) { ex.printStackTrace() ; }
  }
}
```

   Si l'on n'utilise pas d'annuaire, comme dans cet exemple, la chaîne identifiant l'objet (IOR) devra être transmise aux clients.

5. **Développement** d'un client qui utilise l'objet en question via des appels à distance.

```
import java.io.* ;
import org.omg.CORBA.* ;

public class Client
{
  public static void main(String args[])
  {
    if( args.length < 2 )
    {
      System.out.println( "Usage: java Client <ior>" ) ;
      System.exit( 1 ) ;
    }
```

```
      try
      {
        // initialiser l'ORB.
        ORB orb = ORB.init( args, null ) ;

        // Lire l'IOR
        String ior = args[0] ;

        // Traduire IOR en objet CORBA
        org.omg.CORBA.Object obj = orb.string_to_object(args[0]) ;
        // Traduire l'objet CORBA en Objet I<classname> grace au Helper
        I<classname> service = I<classname>Helper.narrow(obj) ;

        // Utiliser l'objet service récupéré
        ...
      }
      catch( org.omg.CORBA.SystemException ex )
      {
        System.err.println( "Erreur !!" ) ;
        ex.printStackTrace() ;
      }
    }
  }
```

6. **Compilation** des programmes serveur et client dès que leur code est terminé. Le client doit avoir accès à l'interface (projetée en java) lors de sa compilation. Le serveur doit avoir accès à l'implémentation de l'interface lors de la compilation.
7. Pour l'**exécution**, il faut d'abord lancer le serveur, ce qui aura pour effet de générer un IOR (identifiant unique) :

   ```
   java Serveur
   ```

   Le serveur doit avoir accès à l'interface et au squelette lors de l'exécution.
   Le client peut alors être exécuté, à condition d'avoir le stub à disposition. Il devra spécifier l'IOR du service mis en place par le serveur.

   ```
   java Client <IOR>
   ```

1.3 Variante : ajout du POA

L'option $-oldImplBase$ d'*idlj* permet de générer un certain type de squelette qui ne fonctionne qu'avec les ORB de Sun, mais est facile d'utilisation. Afin d'assurer l'interopérabilité avec tous les langages, le POA (Portable Object Adapter) se place entre le squelette et l'ORB, donc sur le serveur. L'implémentation de l'interface étends alors $<classname>POA$ au lieu de $\_I<classname>ImplBase$ et le serveur doit initialiser et activer le POA afin de générer l'IOR :

```
// initialiser le POA
POA poa = POAHelper.narrow( orb.resolve_initial_references( "RootPOA" )) ;

// activer le POA
poa.the_POAManager().activate() ;

// créer la référence vers le servant
org.omg.CORBA.Object poaobj = poa.servant_to_reference( myobj ) ;

// Générer l'IOR à partir de la référence
String ior = orb.object_to_string( poaobj ) ;
```

Remarque : le serveur doit importer $org.omg.PortableServer.*$ et traiter l'exception $org.omg.CORBA.UserException$.

2 Exercice

Définissez pas à pas un code mettant à disposition une classe *Couleur* composé d'un triplet RGB, puis l'utilisant via un mécanisme CORBA client/serveur. Cette classe aura une méthode *setColorRGB* permettant de définir ses trois composantes, et une méthode *getLuminance* qui renvoie la luminance de la couleur en question. Pour information, la luminance d'une couleur définie en RGB vaut $0.299 * R + 0.587 * G + 0.114 * B$. Vous utiliserez l'implémentation la plus simple, sans POA.

1. Écrivez l'interface *ICouleur.idl*, sa projection en Java et son implémentation ;
2. Écrivez le code d'une application serveur et d'une application cliente ;
3. Écrivez les instructions à exécuter dans le bon ordre afin de faire fonctionner l'intéraction Corba.

3 Définition de l'interface IDL

Le langage IDL a ses propres structures de données et sa syntaxe.

Types de base : *void*, *short* (16 bits), *unsigned short*, *long* (32 bits), *unsigned long*, *long long* (64 bits), *unsigned long long*, *float* (32 bits, IEEE), *double* (64 bits, IEEE), *long double* (128 bits), *boolean*, *octet* (8 bits), *char* (8 bits, ISO Latin-1), *string*.

Déclarations En IDL, une valeur constante se définit en préfixant la déclaration de variable par le mot-clé *const*, un alias, une énumération et une structure s'écrivent comme en C grâce aux mots-clé respectifs *typedef*, *enum* et *struct*.

Tableaux et séquences se déclarent grâce au mot-clé *sequence*. Une séquence peut être bornée ou non :

séquence bornée sequence <type, borne>

séquence non bornée sequence <type>

Exceptions :

```
exception nom_exception
{
  <type> nom_membre1 ;
  <type> nom_membre2 ;
  ...
  <type> nom_membreN ;
} ;
```

Passage de paramètres de type *in*, *out* ou *inout*. Les cas *out* et *inout* doivent pouvoir fonctionner comme un passage par référence en C++.

3.1 Questions

1. Comment fonctionne le passage de paramètres en Java ?
2. Qu'est-ce qu'une classe *Holder*, générée par l'outil de projection *idlj* ? Comment opèrent-t'elles ?
3. Ajoutez à votre interface *ICouleur.idl* une fonction permettant de récupérer ses valeurs CMY.
   Donnez l'interface Java générée par idlj.
   Ré-écrivez le client afin de pouvoir récupérer ces valeurs.
4. Par quel mot-clé peut-on rendre une fonction IDL non-bloquante ?
   Quelles sont les conditions pour qu'une fonction puisse être définie comme étant non bloquante ?
   Quelles méthodes parmi celles de votre interface IDL pourraient être définies comme non bloquantes ?
5. Un mécanisme de callback permet au serveur d'interpeller un ou des client(s). Pour cela, le client passe en paramètre d'un appel à l'objet distant, un objet sur lequel le serveur pourra appeller des méthodes d'avertissement du client. Quels problèmes va-t'on rencontrer ici ? Quelles solutions peut-on mettre en place ?