Une initiation à CORBA

Common Object Request Broker Architecture

Clémentine Nebut et Abdelhak-Djamel Seriai

LIRMM / Université de Montpellier 2

Octobre 2015

Sommaire

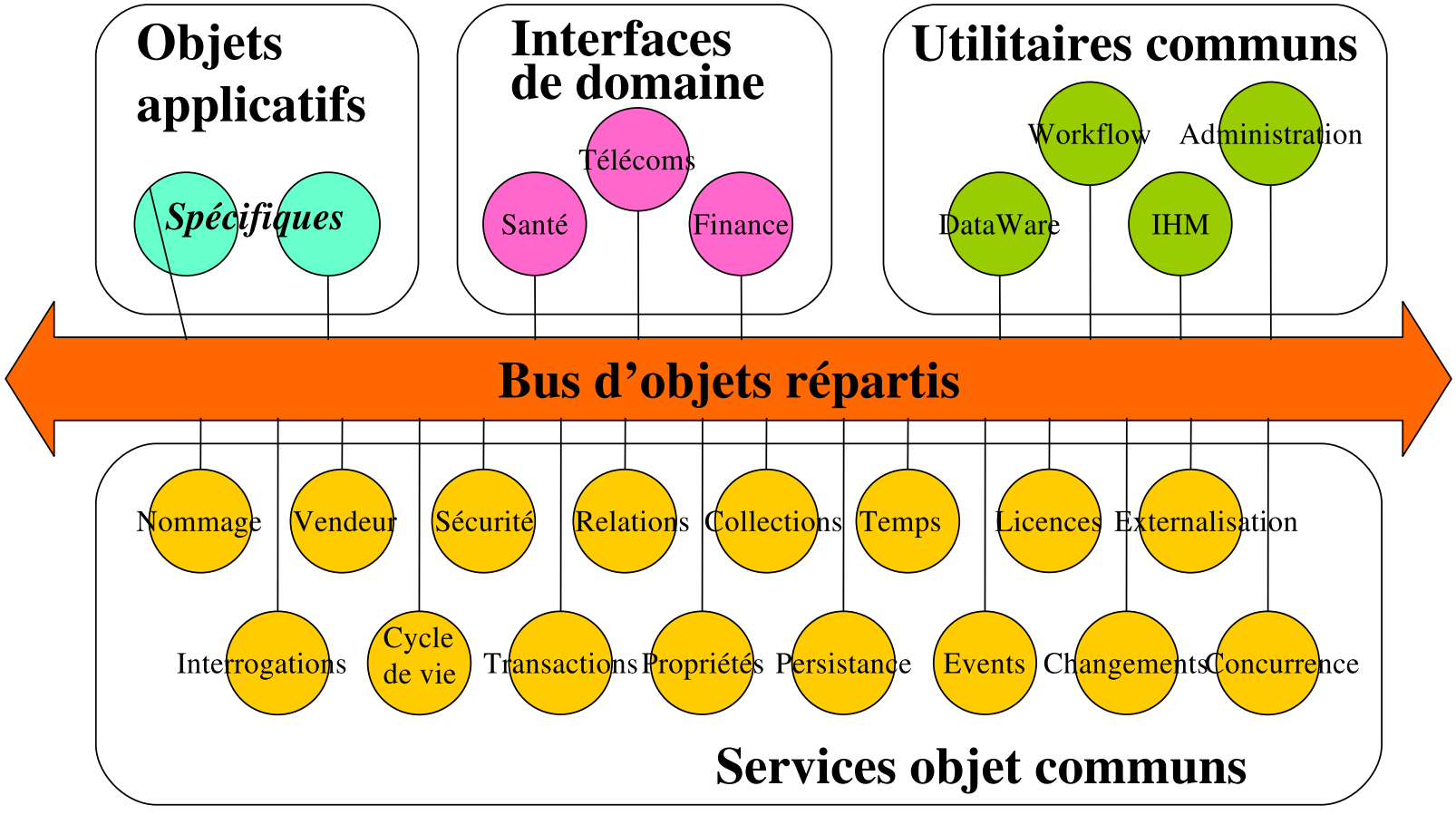
CORBA

- Common Object Request Broker Architecture
- Standard de l'OMG (Object Management Group)
- Répartition des objets dans des environnements hétérogènes
- Indépendant des langages
- Indépendant des OS

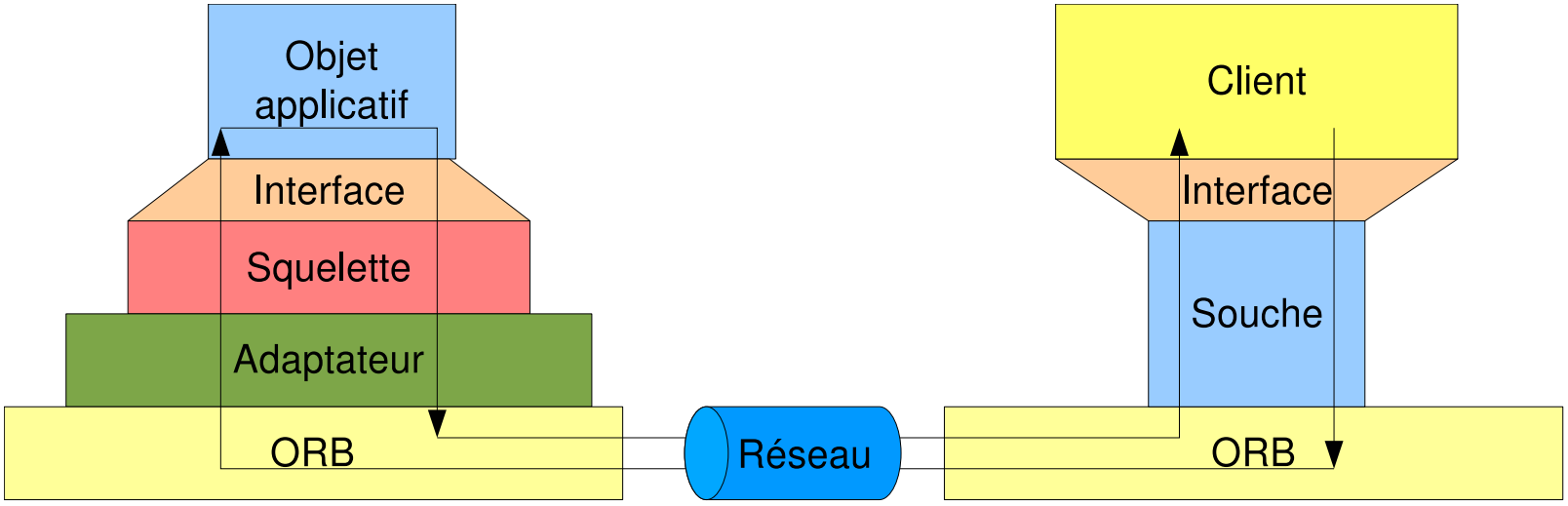
L'OMG

- Object Management Group
- Consortium industriel créé en 1989, environ 1000 membres
- Objectif : Faire émerger des standards industriels pour l'intégration d'applications distribuées hétérogènes et la réutilisation de composants logiciels
- Vision globale : Technologies Objets
  - CORBA comme intergiciel
  - OMA comme architecture globale (Object Management Architecture)
  - UML pour la modélisation
  - MOF pour la méta-modélisation

L'OMA


Objets applicatifs
Spécifiques
Interfaces de domaine
Télécoms
Santé
Finance
Utilitaires communs
Workflow
Administration
DataWare
IHM
Bus d'objets répartis
Nommage
Vendeur
Sécurité
Relations
Collections
Temps
Licences
Externalisation
Interrogations
Cycle de vie
Transactions
Propriétés
Persistance
Events
Changements
Concurrence
Services objet communs

La distribution avec CORBA



Entités

- Interface : écrite en IDL. Définit ce que le serveur met à disposition des clients, définit ce que le client peut utiliser.
- squelette / souche : générés à partir de l'IDL, dans le(s) langage(s) de programmation cible.
- adaptateur : Intermédiaire entre le bus et les possibles supports physiques d'implantation des objets
- ORB = Object Request Broker : bus à objets répartis, transporte les requêtes, active les objets

Référence CORBA

- Sert à désigner / localiser un objet CORBA
  - nom de l'interface OMG IDL
  - protocole de communication (ex. TCP/IP - IIOP)
  - processus serveur (ex. machine et port IP)
  - clé unique pour retrouver l'objet dans le serveur
- Représentation interne au bus
  - un objet langage instance d'une classe langage
  - désigné par une référence langage
- Représentation externe
  - chaîne : IOR :010000...345...2345
  - IOR : Interoperable Object Reference

Objet corba

- Décrit par une interface OMG IDL unique
- Implanté par un objet langage à un instant donné
- Désigné par des références CORBA
- Objet CORBA $\neq$ objet langage
  - car activation contrôlable
    - association objet langage - objet CORBA
  - l'objet d'implantation peut varier au cours du temps
  - une référence peut désigner un objet non présent en mémoire

Bus corba

- Le bus CORBA est omniprésent et réparti
- Chaque exécutable CORBA est lié à une bibliothèque pour utiliser le bus CORBA
- À l'exécution, un objet langage représente le bus dans chaque exécutable
  - c'est un objet local (pas d'IOR) donc non accessible à distance
  - assure le contrôle local du bus
  - fournit les objets « notoires »
    - Premiers objets utiles à l'exploitation du bus CORBA : référentiel d'interfaces et service de nommage

Adaptateur d'objet

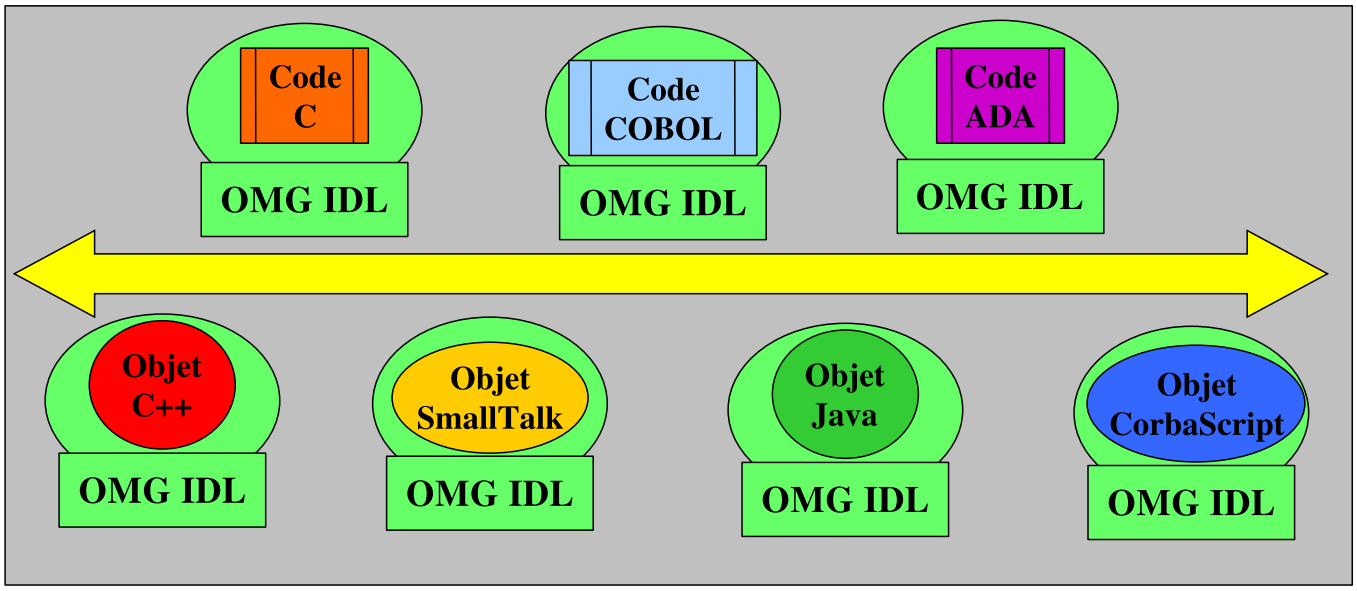
- La structure d'accueil (serveur) de l'implémentation d'un objet CORBA fournit un espace mémoire (état des objets, contexte d'exécution des opérations).
- L'adaptateur d'objet est l'abstraction permettant de :
  - savoir générer et interpréter des références d'objets,
  - connaître l'implantation courante associée à un objet
  - savoir déléguer les requêtes aux objets à leur implantation
  - savoir activer une implantation pour un objet s'il n'en existe pas encore.
- Intermédiaire entre le bus et les possibles supports physiques d'implantation des objets
- Un adaptateur d'objet peut utiliser un Référentiel des Implantations

IDL

Interface Description Language

- Un langage de spécification
  - pour exprimer dans un langage neutre les interfaces publiques des objets
  - indépendant de tout langage de programmation
    - $\neq$RMI ou remoting où interfaces écrites en java, C#, ...
    - parallèle : WSDL pour les services web
- Des constructions simples
  - Modules, interfaces, opérations, attributs
  - Types de base, types construits, méta-types
  - Exceptions
- IDL n'est pas un langage de programmation
  - Pas de constructions de programmation
  - parallèle : WSDL pour les services web

IDL et langages de programmation



- Projection de l'IDL vers différents langages de programmation
- Compilateurs IDL

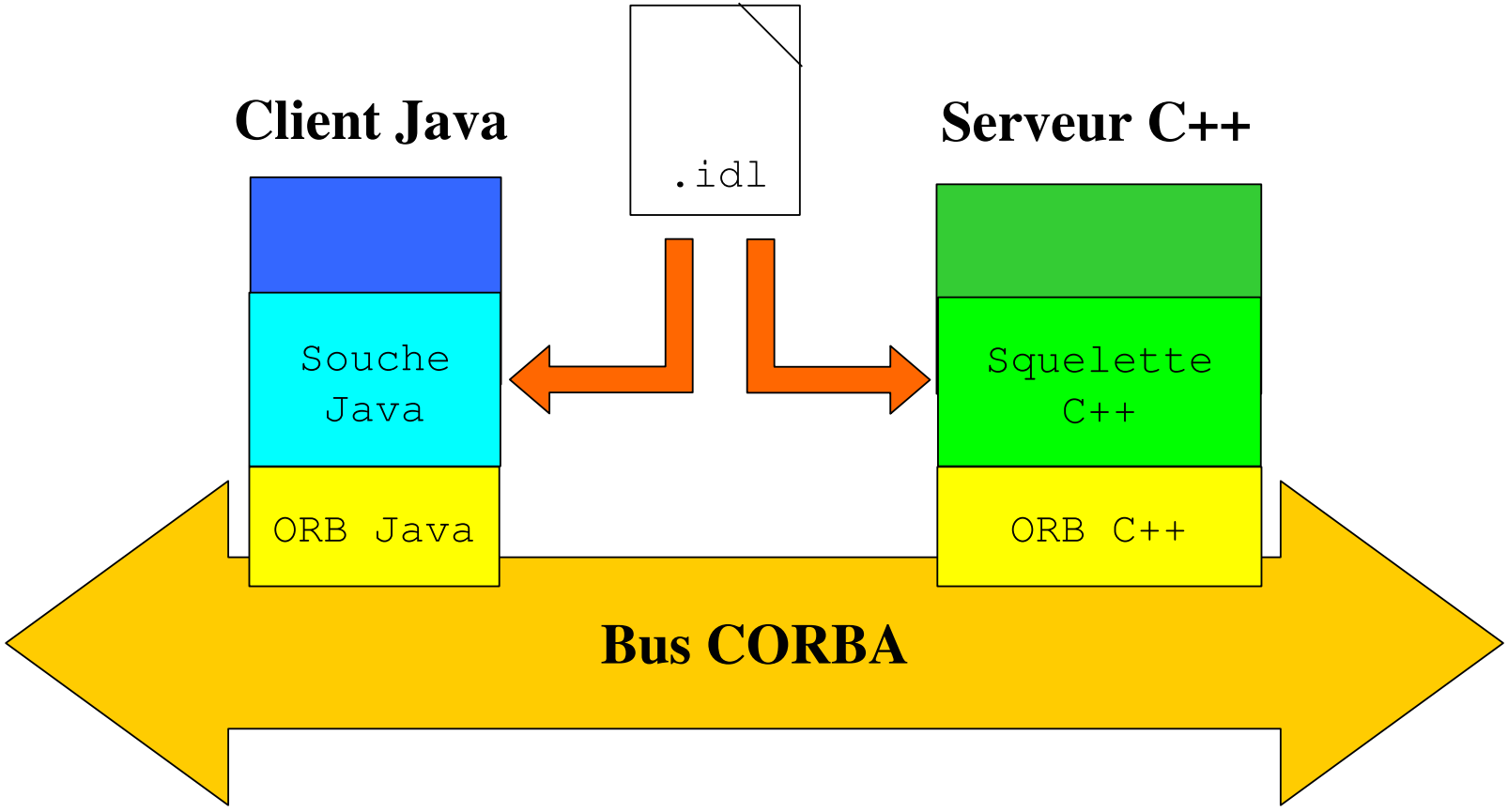
Interface IDL

- Une interface IDL =
  - l'abstraction d'un type d'objets CORBA
  - l'API à rendre publique
- Contient les opérations
  - exportées par l'objet
  - utilisées par les autres objets
- Pas forcément (rarement) toutes les méthodes implantées par la classe de l'objet

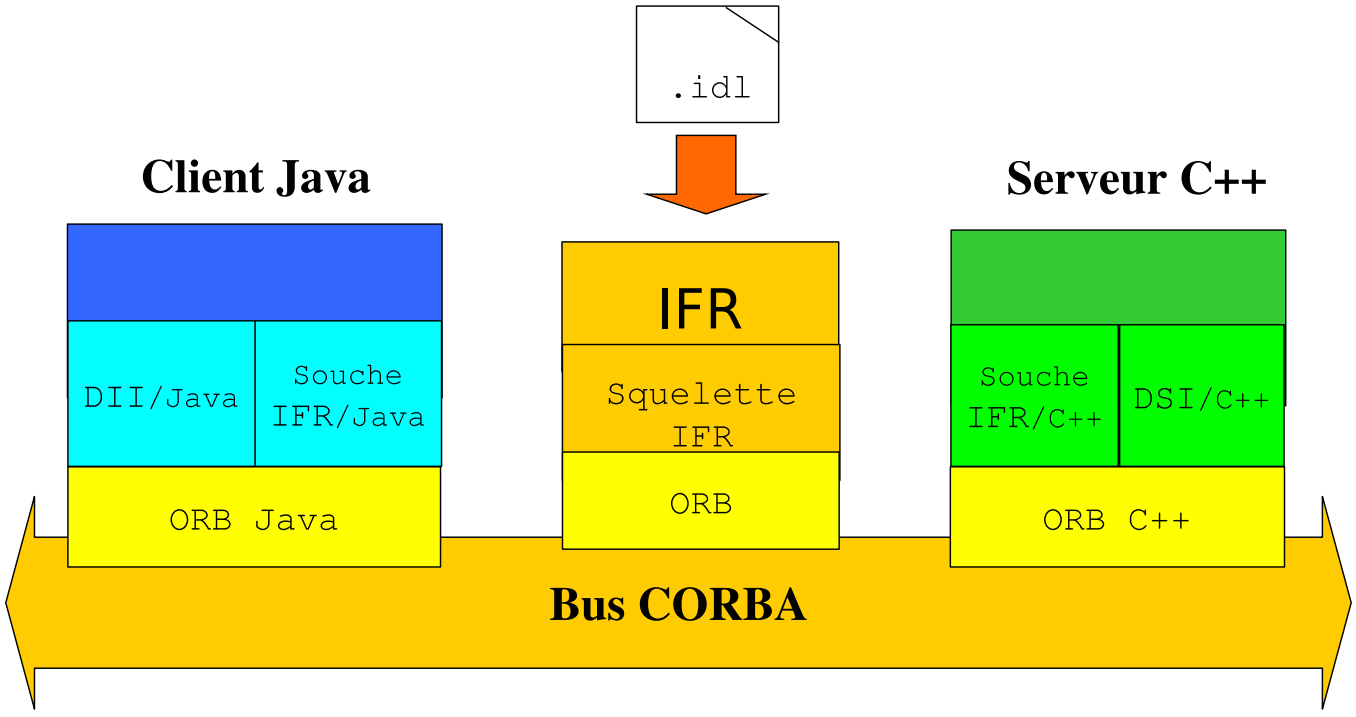
Projection statique et dynamique

- Mode « statique »
  - Projection des descriptions OMG IDL vers les langages d'implantation des clients et serveurs
  - Vérification du typage des invocations à la compilation
- Mode « dynamique »
  - Instanciation sous forme d'objets CORBA des descriptions OMG IDL dans un référentiel des interfaces commun
  - Vérification du typage des invocations à l'exécution

Mode statique

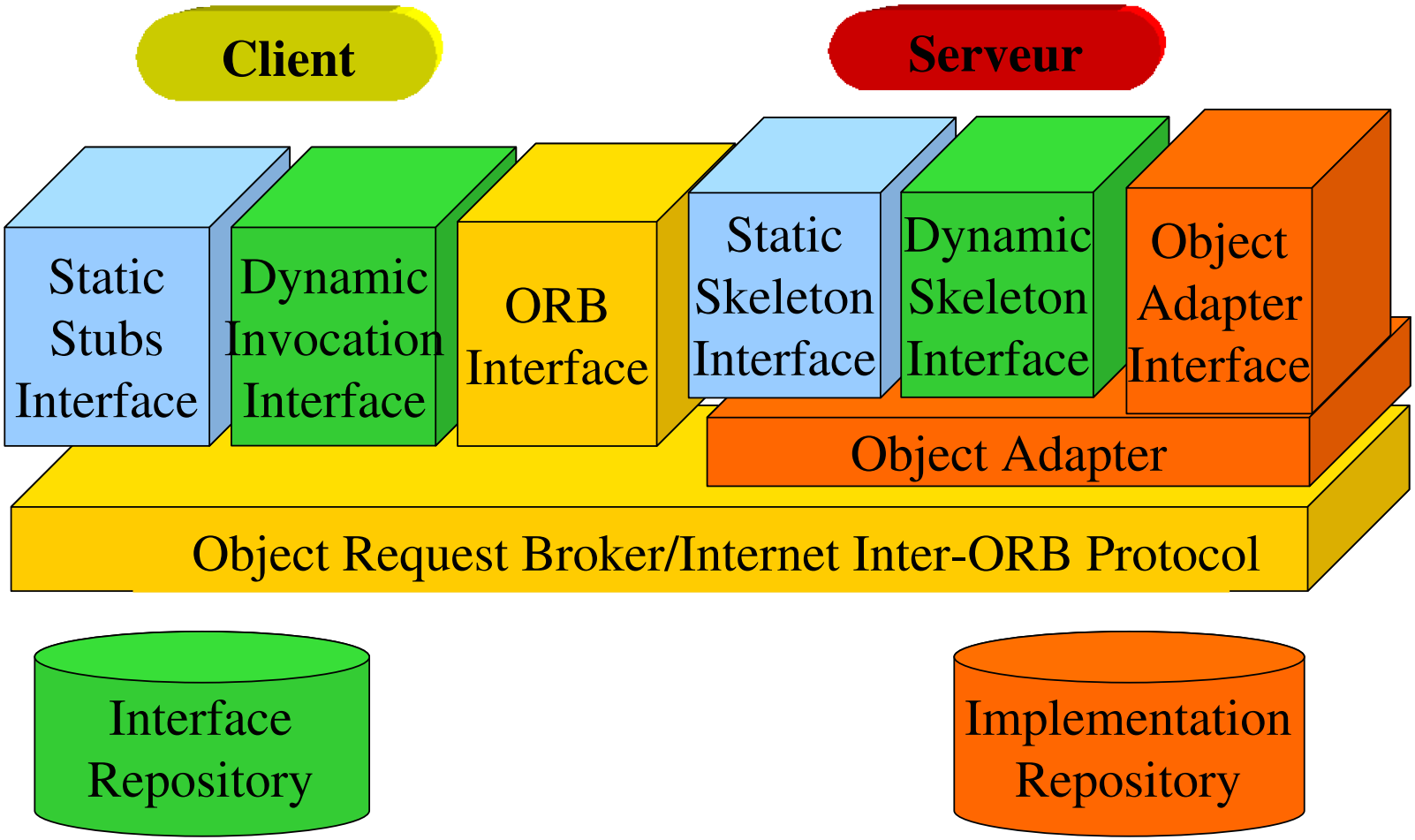


Mode dynamique



DII : Dynamic Invocation Interface – permet de construire des requêtes à l'exécution
IFR : InterFace Repository – Référentiel des interfaces
DSI : Dynamic Skeletton Interface – permet de décoder des requêtes à l'exécution

Les composantes du bus



Etape du développement avec CORBA

- Définir l'interface avec le langage IDL
- Générer les classes nécessaires à la distribution
- Définir le code fonctionnel de l'objet distribué : le servant
- Distribuer l'objet au travers de l'ORB
  - Initialiser l'ORB
  - Enregistrer le servant de l'objet distribué dans l'ORB
  - Rendre disponible une référence permettant de localiser l'objet distribué
  - Mettre l'ORB en attente de requêtes

L'ORB de SUN

- On utilise ici l'ORB fourni par SUN avec java
- Compilateur idl : idlj

Définition de l'interface

hello.idl

```
interface IHello {
void hello ();
};
```

Générer les classes java

lignes de commande

```
mkdir generated
idlj -td generated -emitAll -fall hello.idl
```

- -td : to directory
- -emitAll : génération pour l'idl et les idl inclus
- -fall : génération côté serveur et côté client, idem que -fclient et -fserver

Classes java obtenues

HelloHelper.java classe utilitaire, contient notamment une méthode narrow, servant à remplacer le cast java

HelloHolder.java classes gérant les paramètres out (non pris en charge par java)

Hello.java "cablage" CORBA

HelloOperations.java traduction java de l'idl

HelloPOA.java squelette (portable object adapter)

_HelloStub.java stub

Implémentation de l'objet distribué

Hello_impl.java

```
public class Hello_impl extends HelloPOA {
  public void hello () {
    System.err.println ("Hello World!");
  }
}
```

Distribuer l'objet

server.java

```
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
public class Server {
  public void run (String args[]) {
    try {
      // create and initialize the ORB
      ORB orb = ORB.init(args, null);
      // get reference to rootpoa & activate the POAManager
      POA rootpoa= POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();
      // create servant and register it with the ORB
      Hello_impl hello = new Hello_impl () ;
      // get object reference from the servant
      org.omg.CORBA.Object ref =rootpoa.servant_to_reference(hello);
      Hello href = HelloHelper.narrow(ref);
      System.out.println (orb.object_to_string (href)) ;
      System.err.println("JavaM2Server ready and waiting ...");
      // wait for invocations from clients
      orb.run();
   } catch (Exception e) {System.err.println("ERROR: " + e);}
 }
 public static void main (String args[]) {
  Server srv = new Server () ;
  srv.run (args) ;
 } }
```

Utiliser un objet distant

- Connaître l'interface et éventuellement générer les classes correspondantes
- Initialiser l'ORB
- Récupérer une référence de l'objet distant
- Obtenir la souche de l'objet distant de l'ORB
- Appeler la méthode à distance
- (ouf !)

Utiliser l'objet distant

client.java

```
import org.omg.CORBA.*;
public class Client {
    public void run (String args[]) {
        try {
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);
            // get object reference from the command line
            org.omg.CORBA.Object ref = orb.string_to_object (args[0]) ;
            Hello href = HelloHelper.narrow(ref);
            // perform the method call
            System.out.println ("Invoking object hello") ;
            href.hello () ;
        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
    public static void main (String args[]) {
        Client cli = new Client () ;
        cli.run (args) ;
    }
}
```

Remarque sur l'exécution de cet exemple

- Comment la référence de l'objet distant est échangée entre le serveur et le client ?
  - par affichage de l'IOR sur la sortie standard du serveur
  - et recopie de cet affichage pour le donner en entrée du client
- Bien sûr on peut (doit) faire autrement en réalité : utiliser un serveur de noms. Voir à la fin du cours.

Interface IDL

- Une interface OMG IDL =
  - l'abstraction d'un type d'objets CORBA
  - l'API à rendre publique
- Contient les opérations
  - exportées par l'objet
  - utilisées par les autres objets
- Pas forcément (rarement) toutes les méthodes implantées par la classe de l'objet

Opération IDL

- Une opération OMG IDL
  - l'abstraction d'un traitement réalisé par l'objet
- Est décrite par une signature
  - le nom de l'opération
  - les paramètres
    - leur nom formel
    - leur type
    - leur mode de passage (in, out, inout)
  - le type du résultat
  - les cas d'erreurs ou exceptions

Pourquoi définir les types de données des opérations IDL ?

- Les objets sont hétérogènes
  - différents langages de programmation
  - différents processeurs d'exécution
- Il faut définir précisément les types de données échangées lors des invocations entre objets
  - pour une gestion automatique de l'hétérogénéité
- Pour cela, le langage OMG IDL permet de décrire des types de données

Pourquoi des contrats IDL ?

- De nombreux types et interfaces IDL
- Possibilité de conflits de noms
  - un type Point peut avoir un sens différent et une représentation différente selon l'application
- Aspect génie logiciel
  - groupement de définitions communes
  - définition du contrat entre applications
- Pour cela, le langage IDL permet de définir des modules (contrats) réutilisables

Le langage IDL

- Un langage de spécification
  - modulaire (contrats applicatifs)
  - fortement typé
  - orienté objet
    - interfaces des objets CORBA
    - héritage multiple, polymorphisme
- Pas un langage de programmation
  - même si lexicalement proche du C / C++
  - avec macros et pré-processeur C++

IDL = esperanto ?

- Un « esperanto » entre les langages
  - indépendant des langages et compilateurs
  - des projections disponibles pour de nombreux langages / compilateurs
- Ni l'union de toutes les possibilités
  - pas de généricité / template C++
  - pas de surcharge ni de redéfinition des opérations
- Ni la stricte intersection
  - exceptions en C ?
  - passage « out » en Java ?

Exemple IDL

```
// Contrat OMG IDL d'une application d'annuaire.
#include <date.idl> // Réutilisation du service de dates.
module annuaire {
  typedef string Nom; // Nom d'une personne.
  typedef sequence<Nom> DesNoms; // Ensemble de noms.
  struct Personne { // Description d'une personne.
    Nom nom; // - son nom.
    string informations; // - données diverses.
    string telephone; // - numéro de téléphone.
    string email; // - son adresse Email.
    string url; // - son adresse WWW.
    ::date::Date date_naissance;
  };
  typedef sequence<Personne> DesPersonnes;
... // à suivre
```

Exemple d'interface IDL

```
... // la suite
  interface Repertoire {
    readonly attribute string libelle;
    exception ExisteDeja { Nom nom; };
    exception Inconnu { Nom nom; };
    void ajouterPersonne (in Personne personne) raises(ExisteDeja);
    void retirerPersonne (in Nom nom) raises(Inconnu);
    void modifierPersonne (in Nom nom, in Personne p) raises(Inconn
    DesNoms listerNoms ();
  };
}; // fin module
```

Le pré-processeur IDL

- Définitions OMG IDL dans des fichiers texte
- Vérification lexicale, syntaxique et sémantique réalisées par un compilateur
- Utilisation d'un pré-processeur ANSI C++
  - inclusion de fichiers : #include
  - définition de macros : #define, #ifdef, #endif ...

La projection

- Traduction des concepts CORBA et OMG IDL vers des constructions d'un langage de programmation
  - toutes les constructions OMG IDL
  - la notion de référence d'objet CORBA
  - l'invocation d'opérations
    - les modes de passage des paramètres
    - la gestion des exceptions
  - l'accès aux attributs
  - l'API du bus CORBA décrite en OMG IDL
- Définition des règles de programmation
  - portabilité du code des applications CORBA
  - indépendance par rapport
    - à l'implantation du bus CORBA utilisé
    - aux machines, systèmes d'exploitation et compilateurs
  - Actuellement définie / standardisée pour C, SmallTalk, C++, Ada, COBOL, Java, Python, Common Lisp, ...

Les constructions du langage IDL

- des types élémentaires de données
- des constantes
- des types construits
- des exceptions
- des interfaces
  - opérations, attributs et héritage multiple
- des modules
- des pragmas
- des valeurs
- des types de méta-données

Types de données élémentaires

Données au format binaire normalisé

void	rien
short	entier 16 bits
unsigned short	entier 16 bits non signé
long	entier 32 bits
unsigned long	entier 32 bits non signé
long long	entier 64 bits
unsigned long	long entier 64 bits non signé
float	réel 32 bits (IEEE) double réel 64 bits (IEEE)
long double	réel 128 bits (IEEE)
boolean	booléen
octet	opaque 8 bits
char	caractère 8 bits (ISO Latin)
wchar	caractère international

Projection Java des types élémentaires

Type IDL	Type Java	Exceptions
char	char	CORBA : :DATA_CONVERSION
wchar	char	
string	java.lang.String	CORBA : :MARSHAL CORBA : :DATA_CONVERSION
wstring	java.lang.String	CORBA : :MARSHAL
fixed	java.math.BigDecimal	CORBA : :DATA_CONVERSION

- Types Java plus « larges » que types IDL
  - vérification à l'exécution et exceptions

Autres types élémentaires IDL

- Les chaînes de caractères
  - non bornées : string ou wstring
  - bornées : string<80> ou wstring<256>
- Les nombres à précision fixe
  - fixed<9,2> : 9 chiffres dont 2 décimales
  - pour le commerce électronique
  - pas disponible dans tous les produits CORBA

Les constantes

- Définition d'une valeur nommée
  - d'un type élémentaire
    - entier, réel, booléen, caractère ou chaîne
  - une expression évaluable à la compilation
    - opérateurs « à la C » : (), +, -, *, /, %, <<, >>, ^, |, &, ~
- Exemple : const double PI = 3.14159 ;
- Projection Java –> constante java
  - public static final

Types construits : les alias

- Définition d'un nouveau type à partir de tout type OMG IDL déjà défini
  - équivalence des 2 types
- Exemple :
  - typedef fixed<6,2> Temperature ;
- Plus grande clarté des contrats
- Pas de projection Java (utilisation du type aliasé)

Types construits : les énumérations

- Définition d'un type discret
  - un ensemble de valeurs symboliques
- Exemple :
  - enum Mois { Janvier, Fevrier, Mars, Avril, Mai, Juin, Juillet, Aout, Septembre, Octobre, Novembre, Decembre } ;
- Projection Java –> Classe Java
  - non extensible (final), un constructeur privé, une méthode pour avoir la valeur entière, une méthode de conversion à partir d'un entier
  - A chaque label est associé une valeur entière unique (à partir de zéro)

```
public final class Mois {
public static final int _Janvier = 0;
public static final Mois Janvier = new Mois(_Janvier);
  . . .
public int value() { . . . }
public static Mois from_int(int value) { . . . }
private Mois(int v) { . . . }
```

Types construits : les structures

- Définition d'un type structuré
  - un ensemble de champs typés
- Exemple :
  - struct Date { Jour le_jour ; Mois le_mois ; Annee l_annee ; } ;
- Projection Java –> Classe Java
  - non extensible (final)
  - chaque champ est représenté par un attribut publique
  - un constructeur par défaut un constructeur initialisant tous les champs

```
public final class Date {
public short le_jour;
public Mois le_mois;
public short l_annee;
public Date() { }
public Date(short f1, Mois f2, short f3) { . . . }
```

Types construits : les unions

- Définition d'un regroupement
  - un discriminant scalaire + un ensemble de choix
- Exemple :
  - union DateMultiFormat switch(unsigned short) {
    case 0 : string chaine ;
    case 1 : Jour nombreDeJours ;
    default : Date date ; } ;
- Rarement utilisé (plutôt le type any) !
- Projection java –>classe Java
  - final, un constr. par défaut, accès au discriminant et à cq champ, affectation de cq champ (+ discriminant), affectation du champ default

```
public final class DateMultiFormat {
public DataMultiFormat() { . . . }
public short discriminator() { . . . }
public String chaine() { . . . }
public void chaine(String v) { . . . }
public short nombreDeJours() { . . . }
public void nombreDeJours(short v) { . . . }
public Date date() { . . . }
public void date(Date d) { . . . } }
```

Utilisation des unions

```
DateMultiFormat d = new DateMultiFormat();
d.chaine(« Hello World! »);
if (d.discriminator() != 0) { . . . bizarre . . . }
String s = d.chaine(); // OK
Date d1 = d.date(); // provoque CORBA::BAD_OPERATION
String id = DateMultiFormatHelper.id();
```

Types construits : les tableaux

- Définition d'un ensemble de données homogènes
  - borné à la spécification
  - multiple dimension
- Exemples :
  - typedef long[10] Vecteur10 ;
  - typedef double[10][10] Matrice10x10 ;
- Rarement utilisé (plutôt séquence)
- Projection Java –> tableaux java
  - Contrôle à l'emballage par CORBA
    - si taille différente de la taille de déclaration d'un tableau
    - si taille supérieure à la taille maximale d'une séquence
    - alors exception CORBA : :MARSHAL

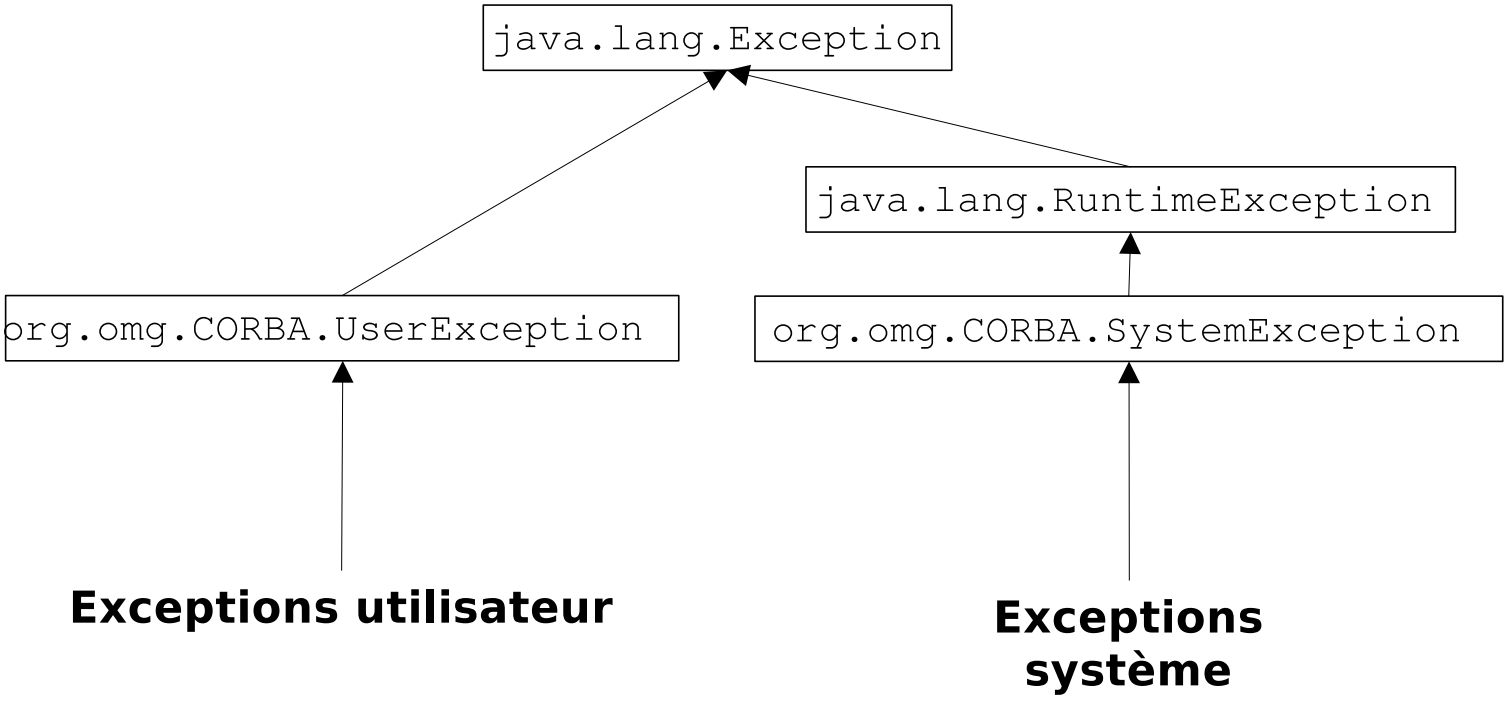
Types construits : les séquences

- Définition d'un ensemble de données homogènes non borné à la spécification
  - une seule dimension
  - avec une possible taille maximale
- Exemples :
  - typedef sequence<long> Vecteur ;
  - typedef sequence<Vecteur,1000> Matrice ;
  - typedef sequence<Date> DesDates ;
- Projection Java –> tableaux java (idem tableaux)

Les exceptions

- Définition d'exceptions
  - 0 ou N champs typés
  - similaire aux structures
- Exemples :
  - exception ErreurInterne {} ;
  - exception MauvaiseDate { DateMultiFormat date ; } ;
- Traduction Java -> Classe Java
  - classe Java non extensible (final), Holder, Helper
  - chaque champ est représenté par un attribut publique
  - un constructeur par défaut un constructeur initialisant tous les champs
  - donc similaire à la projection d'une structure IDL
  - traitement comme des exceptions Java (try-catch)

La hiérarchie des classes d'exception



Interfaces d'objets

- Définition d'un type d'objets abstrait
  - la signature des opérations et des attributs
  - un espace de définition de types / exceptions
- Exemple :
  - interface Repertoire { . . . des opérations, attributs et types . . . } ;
- Une seule spécification pour plusieurs implantations
  - en fonction de la qualité du service désiré, du contexte d'exécution et de l'existant, ...
- Projection Java –> interface Java + Holder + Helper avec narrow
  - signature des opérations et des attributs (accès en lecture et écriture si non readonly)
  - Graphe d'héritage IDL -> graphe d'interfaces Java
    - héritage de l'interface org.omg.CORBA.Object
    - héritage des interfaces Java respectives

Les opérations IDL

- Définition d'une opération
  - un traitement propre à un type d'objet
  - une signature fortement typée
    - le type du résultat
    - les arguments (mode, type, nom formel)
    - les exceptions
- Exemples :
  - void modifierPersonne (in Nom nom, in Personne p) raises(Inconnu) ;
  - Personne obtenirPersonne (in Nom nom) raises(Inconnu) ;

Opérations synchrones ou asynchrones

- Par défaut, une opération est synchrone
  - envoi de la requête vers l'objet
  - attente bloquante de la réponse à cette requête
- Possibilité de définir des opérations asynchrones
  - communication par messages
  - oneway void envoyer_un_message (in long d) ;
  - pas de résultat, de out, de inout ni d'exceptions
  - l'appelant ne peut rien savoir sur l'exécution de l'opération !

Les attributs IDL

- Attribut = opération(s) liée à une propriété
- interface Exemple {
  attribute long unePropriete ;
  readonly attribute long uneAutre ; } ;
- Pas d'exceptions utilisateur ni de paramètres
- Projection Java -> couple de méthodes Java

L'héritage d'interface IDL

- Classification / spécialisation de types d'objets
  - héritage multiple et répété possible
  - surcharge et redéfinition interdites
- Exemple :
  - interface Peripherique { . . . } ;
    interface Ecran : Peripherique { . . . } ;
    interface HautParleur : Peripherique { . . . } ;
    interface EcranAvecHautParleur : Ecran, HautParleur {...} ;
- Attention seulement héritage de spécification !
  - héritage d'implantation dépend du langage utilisé

Modules IDL

- Regroupement de définitions IDL
  - éviter les conflits de noms entre définitions
  - modularité, visibilité et portée des définitions
- Exemple :
  - module date { struct Date { . . . } ; } ;
    module monApplication {
    interface MonService { date : :Date
    retourner_la_date_courante () ; }
    } ;
- Projection Java -> Package

Projection Java : les classes Holder

- En Java, pas de passage « out » et « inout »
- Solution : les classes « Holder »
  - un attribut contenant la valeur
  - un constructeur par défaut
  - un constructeur avec valeur initiale

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

- Classes générées pour tous les types utilisateurs

Classes Holder pour les types élémentaires

- Le package org.omg.CORBA fournit des classes « Holder » pour les types élémentaires

```
final public class BooleanHolder {
public boolean value;
public BooleanHolder() {}
public BooleanHolder(boolean v) { value = v; }
}
```

- idem pour ByteHolder, ShortHolder, IntHolder, LongHolder, FloatHolder, DoubleHolder, CharHolder, StringHolder, ObjectHolder, AnyHolder, TypeCodeHolder

Projection Java : les classes Helper

- Fournissent des fonctions utilitaires pour les types IDL définis par les utilisateurs :
  - insertion dans un Any
  - extraction depuis un Any
  - opération de narrow pour les interfaces
  - ...
- Nom de la classe = NomTypeIDL + Helper

La projection Java et les identificateurs

- Par défaut identificateur OMG IDL => identificateur Java
- Si conflit :
  - avec mots clés du langage Java abstract, class, native, new, ...
  - avec identificateurs réservés pour la projection *Helper, *Holder, *Package
- Alors l'identificateur est précédé par un _
- Donc éviter l'utilisation d'identificateurs réservés !

Le service de nommage

- Permet le partage de références entre applicatifs CORBA
- Offre un espace de désignation symbolique des références d'objets CORBA
  - association de noms symboliques à des IOR
  - graphes de répertoires et chemin d'accès
- Equivalent des pages blanches
- Ce service est décrit en OMG IDL

L'interface du service de nommage

- Le module CosNaming définit
  - association nom - référence d'objet
  - graphes de répertoires : NamingContext
  - chemin d'accès : Name
- Opérations principales du NamingContext
  - ajouter une association : bind, rebind, ...
  - résoudre une association : resolve
  - détruire une association : unbind
  - lister le contenu : list
  - détruire le contexte : destroy

Le chemin d'accès

```
#pragma prefix "omg.org"
module CosNaming {
    typedef string Istring;
    // Définition d'un composant
    struct NameComponent {
            Istring id;
            Istring kind;
    };
    // Définition d'une séquence de composants (chemin c
    typedef sequence<NameComponent> Name;          // . .
```

NamingContext

```
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
// . . .
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(...);
void destroy() raises(NotEmpty);
// Autres déclarations
```

Obtenir la référence du système de nommage

- Comment retrouver la référence de ce service ?
  - C'est un « objet notoire » du bus CORBA
    - de nom NameService
    - fixé par configuration du produit utilisé
- Quel que soit le langage, le scénario est
  1. opération CORBA : :ORB : :resolve_initial_references
  2. conversion en CosNaming : :NamingContext

Obtenir le service de nommage en Java

```
// Retrouver la référence de l'objet notoire « NameService »
import org.omg.CosNaming.*;
org.omg.CORBA.Object objRef = null;
try {
  objRef = orb.resolve_initial_references ("NameService");
} catch( org.omg.CORBA.ORBPackage.InvalidName e ) {
  outils.ARRET ("Le service initial NameService est inconnu"); }
// La convertir en une référence à un objet de type
// CosNaming::NamingContext
NamingContext nsRef = NamingContextHelper.narrow(objRef);
if ( nsRef == null ) {
  outils.ARRET ("Le service initial 'NameService' n'est pas un objet CosNa
}
```

Créer un nom/chemin en Java

```
import org.omg.CosNaming.*;
// Créer un chemin simple
NameComponent[] nsNom = new NameComponent [1];
nsNom[0] = new NameComponent( "banque", "");
// Créer un chemin composé
NameComponent[] nsNom = new NameComponent [2];
nsNom[0] = new NameComponent( "appli", "");
nsNom[1] = new NameComponent(  "banque", "");
```

Enregistrer un objet

- Opération pour publier un objet
  - en général, opération réalisée par le serveur
- Scénario type :
  1. créer un objet
  2. construire un chemin d'accès (Name)
  3. appeler l'opération bind ou rebind avec le chemin et la référence de l'objet
- void bind(in Name n, in Object obj) raises(NotFound, CannotProceed, InvalidName, AlreadyBound) ;
- void rebind(in Name n, in Object obj) raises(NotFound, CannotProceed, InvalidName) ;

Enregistrer un objet en Java

```
// Créer un chemin
NameComponent[] nsNom = new NameComponent [1];
nsNom[0] = new NameComponent("MONOBJET","");
// Enregistrer l'objet
try {
    nsRef.bind (nsNom, uneRefObjet);
} catch (org.omg.CosNaming.NamingContextPackage.NotFound enf) {   . . . }
catch(org.omg.CosNaming.NamingContextPackage.AlreadyBound eab){   . . . }
catch(org.omg.CosNaming.NamingContextPackage.CannotProceed ecp){   . . . }
catch(org.omg.CosNaming.NamingContextPackage.InvalidName ein) {   . . . }
```

Retrouver un objet

- Opération réalisée par un client ou un serveur
- Scénario type :
  1. construire un chemin d'accès (Name)
  2. appeler l'opération resolve avec le chemin
  3. convertir la référence obtenue dans le bon type
- Object resolve (in Name n) raises(NotFound, CannotProceed, InvalidName) ;

Retrouver un objet en Java

```
// Créer un chemin
NameComponent[] nsNom = new NameComponent [1];
nsNom[0] = new NameComponent("MONOBJET","");
// Retrouver l'objet
org.omg.CORBA.Object objRef = null;
try {
    objRef = nsRef.resolve (nsNom);
}
catch (org.omg.CosNaming.NamingContextPackage.NotFound enf) {     . . . }
catch(org.omg.CosNaming.NamingContextPackage.CannotProceed ecp){     . . .
catch (org.omg.CosNaming.NamingContextPackage.InvalidName ein) {     . . .
// Convertir la référence
Banque uneRefObjet = BanqueHelper.narrow(objRef);
```

Rendons à César ...

Transparents inspirés :

- des transparents officiels de l'OMG
- des transparents du LIFL (Geib, Marvie, ...)
- des transparents de Gilles Roussel