CORBA : des concepts à la pratique

Jean-Marc Geib - Christophe Gransart - Philippe Merle
Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
Email : {geib, gransart, merle}@lifl.fr

L'objectif de ce document est de présenter concrètement les différentes étapes de la construction d'une application répartie au dessus du bus CORBA. Pour cela, ce document fait quelques rappels sur la vision globale de l'OMG sur la construction d'applications réparties et passe en revue les possibilités du langage OMG-IDL. Le cœur de ce document illustre progressivement les composantes du bus CORBA sur la construction d'une application d'annuaires avec les langages de programmation Java, C++ et CorbaScript. La dernière partie est consacrée aux mécanismes dynamiques de CORBA.

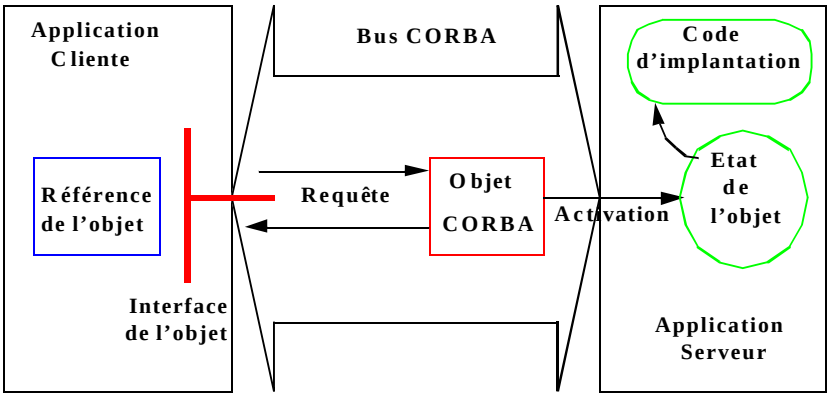
1. Une vision globale de la construction d'applications réparties

1.1. L'OMG

L'Object Management Group (OMG) est un consortium international créé en 1989 et regroupant actuellement plus de 850 acteurs du monde informatique : des constructeurs (IBM, Sun), des producteurs de logiciel (Netscape, Inprise ou ex-Borland/Visigenic, IONA Tech.), des utilisateurs (Boeing, Alcatel) et des institutionels et universités (NASA, INRIA, LIFL). L'objectif de ce groupe est de faire émerger des standards pour l'intégration d'applications distribuées hétérogènes à partir des technologies orientées objet. Ainsi les concepts-clés mis en avant sont la réutilisabilité, l'interopérabilité et la portabilité de composants logiciels. L'élément-clé de la vision de l'OMG est CORBA (Common Object Request Broker Architecture) : un « middleware » orienté objet. Ce bus d'objets répartis offre un support d'exécution masquant les couches techniques d'un système réparti (système d'exploitation, processeur et réseau) et il prend en charge les communications entre les composants logiciels formant les applications réparties hétérogènes.

1.2. Le modèle objet client/serveur

Le bus CORBA propose un modèle orienté objet client/serveur d'abstraction et de coopération entre les applications réparties. Chaque application peut exporter certaines de ses fonctionnalités (services) sous la forme d'objets CORBA : c'est la composante d'abstraction (structuration) de ce modèle. Les interactions entre les applications sont alors matérialisées par des invocations à distance des méthodes des objets : c'est la partie coopération. La notion client/serveur intervient uniquement lors de l'utilisation d'un objet : l'application implantant l'objet est le serveur, l'application utilisant l'objet est le client. Bien entendu, une application peut tout à fait être à la fois cliente et serveur.



La figure précédente présente les différentes notions intervenant dans ce modèle objet client/serveur :

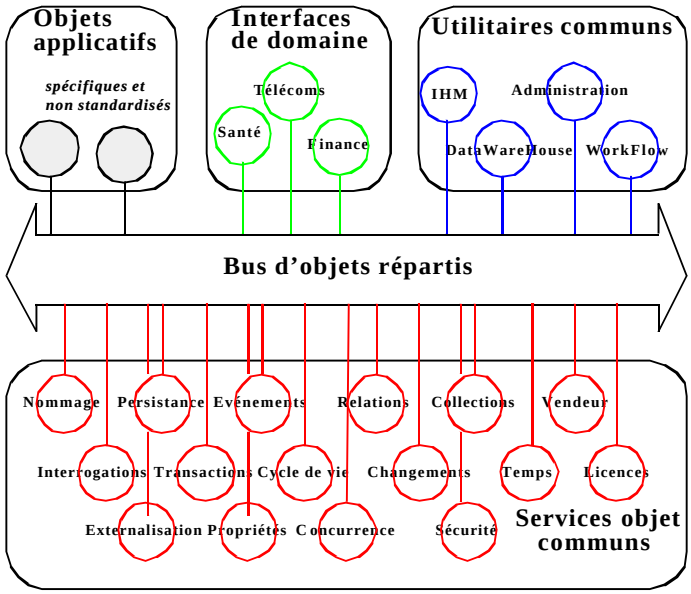
- L'**application cliente** est un programme qui invoque les méthodes des objets à travers le bus CORBA.

- La **référence d'objet** est une structure désignant l'objet CORBA et contenant l'information nécessaire pour le localiser sur le bus.
- L'**interface de l'objet** est le type abstrait de l'objet CORBA définissant ses opérations et attributs. Celle-ci se définit par l'intermédiaire du langage OMG-IDL (voir section 2).
- La **requête** est le mécanisme d'invocation d'une opération ou d'accès à un attribut de l'objet.
- Le **bus CORBA** achemine les requêtes de l'application cliente vers l'objet en masquant tous les problèmes d'hétérogénéité (langages, systèmes d'exploitation, matériels, réseaux).
- L'**objet CORBA** est le composant logiciel cible. C'est une entité virtuelle gérée par le bus CORBA.
- L'**activation** est le processus d'association d'un objet d'implantation à un objet CORBA.
- L'**implantation de l'objet** est l'entité codant l'objet CORBA à un instant donné et gérant un **état de l'objet** temporaire. Au cours du temps, un même objet CORBA peut se voir associer des implantations différentes.
- Le **code d'implantation** regroupe les traitements associés à l'implantation des opérations de l'objet CORBA. Cela peut être par exemple une classe Java aussi bien qu'un ensemble de fonctions C.
- L'**application serveur** est la structure d'accueil des objets d'implantation et des exécutions des opérations. Cela peut être par exemple un processus Unix.

Dans la suite de ce document, nous verrons que ces notions abstraites se traduisent par des composantes technologiques fournies par la norme CORBA.

1.3. L'architecture globale

L'OMG définit aussi une vision globale de la construction d'applications réparties : l'Object Management Architecture Guide [OMAG 95]. Cette architecture globale, appelée aussi l'OMA, vise à classifier les différents objets qui interviennent dans une application en fonction de leurs rôles :



- Le **bus d'objets répartis** est la clé de voûte de l'architecture globale de l'OMG. Il assure le transport des requêtes entre tous les objets CORBA. Il offre un environnement d'exécution aux objets masquant l'hétérogénéité liée aux langages de programmation, aux systèmes d'exploitation, aux processeurs et aux réseaux. Ce bus est plus amplement détaillé dans la suite.
- Les **services objet communs** (CORBAservices) fournissent sous forme d'objets CORBA, spécifiés grâce au langage OMG-IDL, les fonctions systèmes nécessaires à la plupart des applications réparties. Actuellement, l'OMG a défini des services pour les annuaires (Nommage et Vendeur), le cycle de vie des objets, les relations entre objets, les événements, les transactions, la sécurité, la persistance, etc. Au fur et à mesure des besoins, l'OMG ajoute de nouveaux services communs.
- Les **utilitaires communs** (CORBAfacilities) sont des canevas d'objets (« Frameworks ») qui répondent plus particulièrement aux besoins des utilisateurs. Ils standardisent l'interface utilisateur (e.g. Fresco, OpenDoc), la gestion de l'information, l'administration, le Workflow, etc.

- Les **interfaces de domaine** (Domain Interfaces) définissent des objets de métiers spécifiques à des secteurs d'activités comme la finance (e.g. monnaie électronique), la santé (e.g. dossier médical) et les télécoms (e.g. transport multimédia). Leur objectif est de pouvoir assurer l'interopérabilité sémantique entre les systèmes d'informations d'entreprises d'un même métier : les « Business Object Frameworks » (BOFs).
- Les **objets applicatifs** (Application Objects) sont ceux qui sont spécifiques à une application répartie et ne sont donc pas standardisés. Toutefois, dès que le rôle de ces objets apparaît dans plus d'une application ils peuvent alors rentrer dans une des catégories précédentes et donc être standardisés par l'OMG.

1.4. Le bus d'objets répartis CORBA

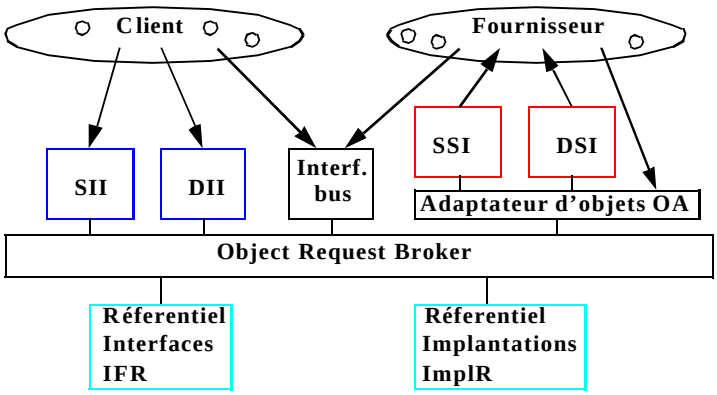
1.4.1. Les caractéristiques

Le bus CORBA est donc l'intermédiaire/négociateur à travers lequel les objets vont pouvoir dialoguer. Il fournit les caractéristiques suivantes :

- La **liaison avec « tous » les langages de programmation** : cependant, actuellement l'OMG a seulement défini officiellement cette liaison pour les langages C, C++, SmallTalk, Ada, COBOL et Java.
- La **transparence des invocations** : les requêtes aux objets semblent toujours être locales, le bus CORBA se chargeant de les acheminer en utilisant le canal de communication le plus approprié.
- L'**invocation statique et dynamique** : ces deux mécanismes complémentaires permettent de soumettre les requêtes aux objets. En statique, les invocations sont contrôlées à la compilation. En dynamique, les invocations doivent être contrôlées à l'exécution.
- Un **système auto-descriptif** : les interfaces des objets sont connues du bus et sont aussi accessibles par les programmes par l'intermédiaire du référentiel des interfaces. Nous reviendrons plus en détail sur cet aspect dans la section 4.
- L'**activation automatique et transparente des objets** : les objets sont en mémoire uniquement s'ils sont utilisés par des applications clientes.
- L'**interopérabilité entre bus** : à partir de la norme CORBA 2.0, un protocole générique de transport des requêtes (GIOP pour General Inter-ORB Protocol) a été défini permettant l'interconnexion de bus CORBA provenant de fournisseurs distincts, une de ses instanciations est l'Internet Inter-ORB Protocol (IIOP) fonctionnant au dessus de TCP/IP.

1.4.2. Les composantes

Le bus CORBA fournit les composantes suivantes :



- **ORB** (Object Request Broker) est le noyau de transport des requêtes aux objets. Il intègre au minimum les protocoles GIOP et IIOP. L'interface au bus fournit les primitives de base comme l'initialisation de l'ORB.
- **SII** (Static Invocation Interface) est l'interface d'invocations statiques permettant de soumettre des requêtes contrôlées à la compilation des programmes. Cette interface est générée à partir de définitions OMG-IDL.
- **DII** (Dynamic Invocation Interface) est l'interface d'invocations dynamiques permettant de construire dynamiquement des requêtes vers n'importe quel objet CORBA sans générer/utiliser une interface SII.
- **IFR** (Interface Repository) est le référentiel des interfaces contenant une représentation des interfaces OMG-IDL accessible par les applications durant l'exécution.

- **SSI** (Skeleton Static Interface) est l'interface de squelettes statiques qui permet à l'implantation des objets de recevoir les requêtes leur étant destinées. Cette interface est générée comme l'interface SII.
- **DSI** (Dynamic Skeleton Interface) est l'interface de squelettes dynamiques qui permet d'intercepter dynamiquement toute requête sans générer une interface SSI. C'est le pendant de DII pour un serveur.
- **OA** (Object Adapter) est l'adaptateur d'objets qui s'occupe de créer les objets CORBA, de maintenir les associations entre objets CORBA et implantations et de réaliser l'activation automatique si nécessaire.
- **ImplR** (Implementation Repository) est le référentiel des implantations qui contient l'information nécessaire à l'activation. Ce référentiel est spécifique à chaque produit CORBA.

Ces différentes composantes sont toutes décrites dans le langage OMG-IDL, ce qui les rend accessibles au travers du bus (e.g. le référentiel des interfaces). Cela permet aussi d'homogénéiser les différentes implantations possibles de la norme car différentes approches peuvent être envisagées pour construire un bus :

- **1 bus = 1 processus** : les objets sont dans le même espace mémoire que les clients. C'est le cas typique des applications embarquées. Ici, le langage OMG-IDL sert à spécifier les objets.
- **1 bus = 1 OS** : les clients et les fournisseurs sont des processus différents sur la même machine. Le bus CORBA peut alors être tout ou partie du système d'exploitation, e.g. Spring l'OS orienté objet de SUN [Spring 95].
- **1 bus réseau** : les processus sont sur des sites différents et les requêtes sont véhiculées à travers le réseau, c'est le cas sur Internet avec IIOP.
- **Fédération de bus CORBA** : plusieurs bus distincts peuvent être mis ensemble pour former une fédération. Les communications entre ces bus peuvent se faire soit par le protocole commun IIOP soit à travers des passerelles spécifiques ou génériques (DII-DSI).

1.4.3. Les protocoles réseaux

Tout bus à la norme CORBA 2.0 doit fournir les protocoles GIOP et IIOP. Le protocole GIOP définit une représentation commune des données (CDR ou Common Data Representation), un format de références d'objet interopérable (IOR ou Interoperable Object Reference) et un ensemble de messages de transport des requêtes aux objets (Request, Reply, …). Cependant, GIOP est seulement un protocole générique, IIOP fournit alors une implantation de GIOP au dessus de TCP/IP et donc d'Internet. Les IORs dans le contexte d'IIOP doivent contenir :

- le nom complet de l'interface OMG-IDL de l'objet ;
- l'adresse IP de la machine Internet où est localisé l'objet ;
- un port IP pour se connecter au serveur de l'objet ;
- une clé pour désigner l'objet dans le serveur. Son format est libre et il est donc différent pour chaque implantation du bus CORBA.

Toutefois, un bus CORBA peut contenir d'autres protocoles de transport des requêtes aux objets. Par exemple, nous travaillons actuellement sur un bus multi-protocoles permettant d'avoir simultanément des communications en IIOP, UDP-IOP et Multicast-IOP.

1.5. Les services objet communs

Les services objet communs ont pour objectif de standardiser les interfaces des fonctions système indispensables à la construction et l'exécution de la plupart des applications réparties. Cette section décrit sommairement ces services selon leur rôle dans une application répartie. Elle se base sur l'actuelle spécification CORBAservices [COS 98] et les travaux en cours à l'OMG.

1.5.1. La recherche d'objets

Cette catégorie de services offre les mécanismes pour rechercher/retrouver dynamiquement sur le bus les objets nécessaires aux applications. Ce sont les équivalents des annuaires téléphoniques :

- Le service **Nommage** (Naming Service) est l'équivalent des « pages blanches » : les objets sont désignés par des noms symboliques. Cet annuaire est matérialisé par un graphe de répertoires de désignation.
- Le service **Vendeur** (Trader Service) est l'équivalent des « pages jaunes » : les objets peuvent être recherchés en fonction de leurs caractéristiques.

1.5.2. La vie des objets

Cette catégorie regroupe les services prenant en charge les différentes étapes de la vie des objets CORBA.

- Le service **Cycle de Vie** (Life Cycle Service) décrit des interfaces pour la création, la copie, le déplacement et la destruction des objets sur le bus. Il définit pour cela la notion de fabriques d'objets («Object Factory»).
- Le service **Propriétés** (Property Service) permet aux utilisateurs d'associer dynamiquement des valeurs nommées à des objets. Ces propriétés ne modifient pas l'interface IDL, mais représentent des besoins spécifiques du client comme par exemple des annotations.
- Le service **Relations** (Relationship Service) sert à gérer des associations dynamiques (appartenance, inclusion, référence, auteur, emploi,...) reliant des objets sur le bus. Il permet aussi de manipuler des graphes d'objets.
- Le service **Externalisation** (Externalization Service) apporte un mécanisme standard pour fixer ou extraire des objets du bus. La migration, le passage par valeur, et la sauvegarde des objets doivent reposer sur ce service.
- Le service **Persistance** (Persistent Object Service) offre des interfaces communes à un mécanisme permettant de stocker des objets sur un support persistant. Quel que soit le support utilisé, ce service s'utilise de la même manière via un «Persistent Object Manager». Un objet persistant doit hériter de l'interface «Persistent Object» et d'un mécanisme d'externalisation.
- Le service **Interrogations** (Query Service) permet d'interroger les attributs des objets. Il repose sur les langages standards d'interrogation comme SQL3 ou OQL. L'interface *Query* permet de manipuler les requêtes comme des objets CORBA. Les objets résultats sont mis dans une collection. Le service peut fédérer des espaces d'objets hétérogènes.
- Le service **Collections** (Collection Service) permet de manipuler d'une manière uniforme des objets sous la forme de collections et d'itérateurs. Les structures de données classiques (listes, piles, tas,..) sont construites par sous-classement. Ce service est aussi conçu pour être utilisé avec le service d'interrogations pour stocker les résultats de requêtes.
- Le service **Changements** (Versionning Service) permet de gérer et de suivre l'évolution des différentes versions des objets. Ce service maintient des informations sur les évolutions des interfaces et des implantations. Cependant, il n'est pas encore spécifié officiellement.

1.5.3. La sûreté de fonctionnement

Cette catégorie de services fournit les fonctions système assurant la sûreté de fonctionnement nécessaire à des applications réparties.

- Le service **Sécurité** (Security Service) permet d'identifier et d'authentifier les clients, de chiffrer et de certifier les communications et de contrôler les autorisations d'accès. Ce service utilise les notions de serveurs d'authentification, de clients/rôles/droits (et délégation de droits), d'IIOP sécurisé (utilisant Kerberos ou SSL).
- Le service **Transactions** (Object Transaction Service) assure l'exécution de traitements transactionnels impliquant des objets distribués et des bases de données en fournissant les propriétés ACID.
- Le service **Concurrence** (Concurrency Service) fournit les mécanismes pour contrôler et ordonnancer les invocations concurrentes sur les objets. Le mécanisme proposé est le verrou. Ce service est conçu pour être utilisé conjointement avec le service Transactions.

1.5.4. Les communications asynchrones

Par défaut, la coopération des objets CORBA est réalisée selon un mode de communication client/serveur synchrone. Toutefois, il existe un ensemble de services assurant des communications asynchrones.

- Le service **Evénements** (Event Service) permet aux objets de produire des événements asynchrones à destination d'objets consommateurs à travers des canaux d'événements. Les canaux fournissent deux modes de fonctionnement. Dans le mode « push », le producteur a l'initiative de la production, le consommateur est notifié des événements. Dans le mode « pull », le consommateur demande explicitement les événements, le producteur est alors sollicité.
- Le service **Notification** (Notification Service) est une extension du service précédent. Les consommateurs sont uniquement notifiés des événements les intéressant. Pour cela, ils posent des filtres sur le canal réduisant ainsi le trafic réseau engendré par la propagation des événements inintéressants.

- Le service **Messagerie** (CORBA Messaging [OMG-CM]) définit un nouveau modèle de communication asynchrone permettant de gérer des requêtes persistantes lorsque l'objet appelant et l'objet appelé ne sont pas présents simultanément sur le bus. Cela permet d'avoir des fonctionnalités proches des MOMs (« Message Oriented Middleware »).

1.5.5. Autres fonctions

- Le service **Temps** (Time Service) fournit des interfaces permettant d'obtenir une horloge globale sur le bus (Universal Time Object), de mesurer le temps et de synchroniser les objets.
- Le service **Licences** (Licensing Service) permet de mesurer et de contrôler l'utilisation des objets, et cela en vue de facturer les clients et de rémunérer les fournisseurs.

1.6. Les interfaces de domaine

Parallèlement aux services, l'OMG porte ses efforts sur la conception de canevas d'objets dédiés à des secteurs d'activité (ou métiers) ou des besoins applicatifs transverses à des métiers. Ces efforts sont effectués au sein de groupes de travail nommés selon les cas : « Working Groups », « Domain Task Forces » ou « Special Interest Groups », etc. Les groupes les plus actifs sont actuellement :

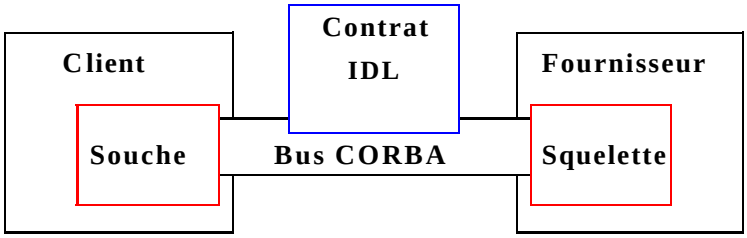
- **Business Objects DTF** standardise les méthodologies et les technologies pour la conception d'applications métier au dessus de CORBA.
- **Workflow Workgroup** travaille sur les outils CORBA nécessaires au fonctionnement des applications de gestion de flux de travail indispensables aux activités coopératives au sein des entreprises.
- **C4I DSIG** fait la promotion de CORBA au sein des organismes et administrations liés à la défense en jouant un rôle pédagogique.
- **End User SIG** travaille en relation avec les utilisateurs finaux à fin de déterminer leurs besoins, de connaître leurs expériences pour mieux orienter les futurs technologies CORBA.
- **CORBAmed** définit les standards OMG pour le domaine de la médecine. Ces standards permettront l'interopérabilité entre les acteurs de la santé en définissant par exemple la notion d'objets patient.
- **Life Sciences Research DTF** est un nouveau groupe dédié à l'utilisation de CORBA par les instituts de recherche dans les sciences de la vie : les universités et les laboratoires pharmaceutiques.
- **Electronic Commerce** s'intéresse à la définition de technologies CORBA pour les applications de commerce électronique.
- **Financial Services DTF** produit les spécifications OMG liées au domaine de la finance/banque comme, par exemple, la définition d'objets monnaie et d'objets convertisseur de monnaie.
- **Telecom DTF** fait la liaison entre le monde CORBA et le monde des télécommunications.
- **Internet Platform SIG** travaille sur les rapprochements des technologies CORBA avec les technologies d'Internet. Il étudie principalement les relations avec le World Wide Web.
- **Manufacturing DFT** tente de combler le fossé entre les technologies OMG et les besoins des industries.
- **Realtime SIG** réfléchit sur la prise en compte des aspects temps réel au sein du bus CORBA.
- **Distributed Simulation SIG** adresse les problèmes liés à la simulation distribuée et son exécution au dessus d'un bus CORBA.
- **Test SIG** doit définir les méthodologies, les outils, les protocoles pour automatiser les tests d'applications réparties CORBA.
- **Object Model Working Group** travaille sur l'unification des modèles orientés objet en vue de créer un standard plus généraliste que celui proposé actuellement par CORBA.
- **UML Revision Task Force** travaille sur le futur du langage UML. Ce langage, standardisé par l'OMG, est une notation graphique pour définir des modèles orientés objet.

Cette liste n'est pas exhaustive car les groupes de travail OMG sont lancés ou arrêtés au fur et à mesure du temps, des besoins et de l'activité de leurs participants. De plus, vu le nombre de groupes, il nous est assez difficile de suivre pleinement l'activité de chacun d'eux. Cependant, le point important à retenir est que l'OMG est un consortium de standardisation « de facto » ouvert qui est prêt à accueillir et/ou créer tout nouveau groupe de travail voulant réfléchir aux impacts de l'utilisation du bus CORBA dans un contexte applicatif particulier. Par exemple, le groupe « Sciences de la vie » a été créé tout récemment pour permettre d'étendre la vision globale de l'OMG vers ce nouveau domaine.

2. Le langage OMG-IDL

2.1 La notion de contrat IDL

Le langage OMG-IDL (Interface Definition Language) permet d'exprimer, sous la forme de contrats IDL [Geib 97], la coopération entre les fournisseurs et les utilisateurs de services, en séparant l'interface de l'implantation des objets et en masquant les divers problèmes liés à l'interopérabilité, l'hétérogénéité et la localisation de ceux-ci. Un contrat IDL spécifie les types manipulés par un ensemble d'applications réparties, c'est-à-dire les types d'objets (ou interfaces IDL) et les types de données échangés entre les objets. Le contrat IDL isole ainsi les clients et fournisseurs de l'infrastructure logicielle et matérielle les mettant en relation à travers le bus CORBA.



Les contrats IDL sont projetés en souches IDL (ou interface d'invocations statiques SII) dans l'environnement de programmation du client et en squelettes IDL (ou interface de squelettes statiques SSI) dans l'environnement de programmation du fournisseur. Le client invoque localement les souches pour accéder aux objets. Les souches IDL construisent des requêtes, qui vont être transportées par le bus, puis délivrées par celui-ci aux squelettes IDL qui les délégueront aux objets. Ainsi le langage OMG-IDL est la clé de voûte du bus d'objets répartis CORBA.

2.2. Un exemple

L'exemple ci-dessous illustre la spécification d'un service de gestion de dates. Les mots clés OMG-IDL sont mis en gras :

```
#pragma prefix "lifl.fr"            // Définit l'organisation spécifiant le contrat IDL.
module date {                       // Regroupement des définitions communes à ce service.
  typedef short Annee;              // Une année est codée par un entier 16 bits.
  typedef sequence<Annee> DesAnnees; // Ensemble non borné d'années.

  enum Mois {                       // Enumération des mois.
    Janvier, Fevrier, Mars, Avril, Mai, Juin, Juillet,
    Aout, Septembre, Octobre, Novembre, Decembre
  };
  typedef sequence<Mois> DesMois;   // Ensemble non borné de mois.

  enum JourDansLaSemaine {          // Enumération des jours de la semaine.
    Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche
  };
  typedef sequence<JourDansLaSemaine> DesJoursDansLaSemaine;

  typedef unsigned short Jour;      // Un jour est codé par un entier 16 bits non signé.
  typedef sequence<Jour> DesJours;  // Ensemble non borné de jours.

  struct Date {                     // Structure définissant la notion de date.
    Jour le_jour;
    Mois le_mois;
    Annee l_annee;
  };
  typedef sequence<Date> DesDates;  // Ensemble non borné de dates.

  union DateMultiFormat             // Regroupement de divers formats de dates.
  switch(unsigned short) {          // Le discriminant anonyme sert à choisir le format.
    case 0:    string chaine;       // Si =0 alors l'union contient une chaîne.
    case 1:    Jour nombreDeJours;  // Si =1 alors elle contient un nombre de jours.
    default:   Date date;           // Par défaut, cette union contient une date.
  };
  typedef sequence<DateMultiFormat> DesDateMultiFormats;
```

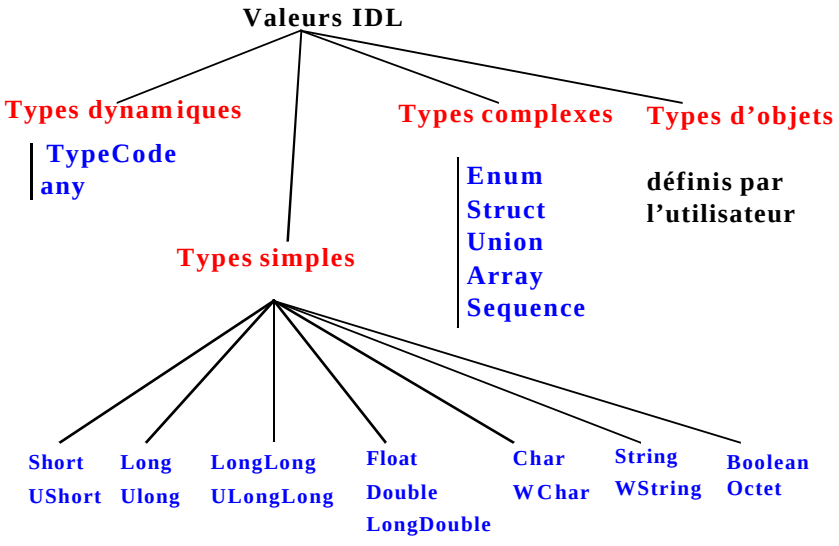
```
  exception ErreurInterne {};                       // Exception vide.
  exception MauvaiseDate { DateMultiFormat date; }; // Exception avec des champs.

  interface Traitement {            // Service de traitement sur les dates.
    boolean verifierDate (in Date d);
    JourDansLaSemaine calculerJourDansLaSemaine (in Date d)
                                        raises(ErreurInterne,MauvaiseDate);
    long nbJoursEntreDeuxDates (in Date d1, in Date d2) raises(MauvaiseDate);
    void dateSuivante (inout Date d, in Jour nombreJours) raises(MauvaiseDate);
  };

  interface Convertisseur {         // Service de conversion de dates.
    Jour convertirDateVersJourDansAnnee (in Date d) raises(MauvaiseDate);
    Date convertirChaineVersDate (in string chaine) raises(MauvaiseDate);
    string convertirDateVersChaine (in Date d) raises(MauvaiseDate);

    attribute Annee annee_courante;  // Fixer l'année courante pour l'opération suivante.
    Date convertirJourDansAnneeVersDate (in Jour jour);

    readonly attribute Annee base_annee ; // L'année de référence des opérations suivantes.
    Date convertirJourVersDate (in Jour jour);
    Jour convertirDateVersJour (in Date d) raises(MauvaiseDate);

    void obtenirDate (in DateMultiFormat dmf, out Date d) raises(MauvaiseDate);
  };

  interface ServiceDate : Traitement, Convertisseur { };  // Héritage de spécification.
};
```

Le langage OMG-IDL permet d'exprimer simplement et indépendamment de tout aspect d'implantation (langages, systèmes d'exploitation, machines et réseaux) et/ou de répartition les types de données et d'objets manipulés par un service CORBA. Bien sûr, le choix des identificateurs de ces types est déterminant pour une lecture aisée d'une spécification OMG-IDL comme c'est le cas dans l'exemple ci-dessus.

2.3. Les fonctionnalités

La liste suivante présente les constructions offertes par le langage OMG-IDL et leurs utilités :

- Un **pragma** permet de fixer les identifiants désignant les définitions OMG-IDL à l'intérieur du référentiel des interfaces. Ces identifiants assurent une désignation unique et cohérente des définitions OMG-IDL quel que soit le référentiel des interfaces consulté, i.e. ils sont utilisés pour fédérer des IFRs. La forme la plus utilisée est le pragma *prefix* qui fixe l'organisation définissant un contrat IDL en utilisant son nom de domaine Internet. Le pragma *version* permet de définir le numéro de version d'une spécification OMG-IDL.
- Un **module** sert à regrouper des définitions de types qui ont un intérêt commun (e.g. *date*). Cette structuration permet aussi de limiter les conflits de noms pouvant intervenir entre plusieurs spécifications, il est facilement imaginable que le nom *Convertisseur* soit utilisé dans une autre spécification. Les conflits de nom de module sont facilement réglés par le pragma *prefix*.



- Les **types de données de base** sont ceux couramment rencontrés en informatique : *void*, *short*, *unsigned short*, *long*, *unsigned long*, *long long* (64 bits), *unsigned long long*, *float*, *double*, *long double* (128 bits), *boolean*, *octet*, *char*, *string*, *wchar* et *wstring* (format de caractères international) et *fixed* pour les nombres à précision fixe. Le format binaire de ces types est défini par la norme afin de régler les problèmes d'échanges de données entre environnements hétérogènes.
- Les **types de méta-données** (*TypeCode* et *any*) sont une composante spécifique à l'OMG-IDL et nouvelle par rapport à d'autres langages IDL. Le type *TypeCode* permet de stocker la description de n'importe quel type IDL. Le type *any* permet de stocker une valeur IDL de n'importe quel type en conservant son *TypeCode*. Ces méta-types permettent de spécifier des contrats IDL génériques indépendants des types de données manipulés, e.g. une pile d'*any* stocke n'importe quelle valeur.
- Une **constante** se définit par un type simple, un nom et une valeur évaluable à la compilation (e.g. *const double PI = 3.1415 ;*).
- Un **alias** (ou *typedef*) permet de créer de nouveaux types en renommant des types déjà définis. Par exemple, il est plus clair de spécifier qu'une opération retourne un *Jour* plutôt qu'un entier 16 bits signé.
- Une **énumération** (ou *enum*) définit un type discret via un ensemble d'identificateurs. Par exemple, il est plus parlant d'énumérer les jours de la semaine que d'utiliser un entier pour coder un *JourDansLaSemaine*.
- Une **structure** définit une fiche regroupant des champs (e.g. *Date*). Cette construction est fortement employée car elle permet de transférer des structures de données composées entre objets CORBA.
- Une **union** juxtapose un ensemble de champs, le choix étant arbitré par un discriminant de type simple (entiers, caractère, booléen ou énumérations). Toutefois, cette construction est très rarement utilisée et on préférera plutôt utiliser le type *any*.
- Un **tableau** sert à transmettre un ensemble de taille fixe de données homogènes. Mais cette construction est rarement utilisée car on lui préfère la suivante.
- Une **séquence** permet de transférer un ensemble de données homogènes dont la taille sera fixée à l'exécution et non à la définition comme pour un tableau (e.g. *DesDates*).
- Une **exception** spécifie une structure de données permettant à une opération de signaler les cas d'erreurs ou de problèmes exceptionnels pouvant survenir lors de son invocation. Une exception se compose de zéro (e.g. *ErreurInterne*) ou plusieurs champs (e.g. *MauvaiseDate*).
- Une **interface** décrit les opérations fournies par un type d'objets CORBA (e.g. *Traitement* et *Convertisseur*). Une interface IDL peut hériter de plusieurs autres interfaces (e.g. *ServiceDate*). L'héritage multiple et répété est autorisé car ici on parle seulement d'héritage de spécifications. La seule contrainte imposée est qu'une interface ne peut pas hériter de deux interfaces qui définissent un même nom d'opérations. Ce problème doit être résolu manuellement en évitant/éliminant les conflits de noms. La surcharge est donc aussi interdite en OMG-IDL. Actuellement, un objet CORBA ne peut implanter qu'une seule interface IDL. CORBA 3.0 intégrera la notion d'interfaces multiples [OMG-MI] comme cela existe déjà en Java ou avec COM [Orfali 96].
- Une **opération** se définit par une signature qui comprend le type du résultat, le nom de l'opération, la liste des paramètres et la liste des exceptions éventuellement déclenchées lors de l'invocation. Un paramètre se caractérise par un mode de passage, un type et un nom formel. Les modes de passages autorisés sont *in*, *out* et *inout*. Le résultat et les paramètres peuvent être de n'importe quel type exprimable en IDL. Par défaut, l'invocation d'une opération est synchrone. Cependant, il est possible de spécifier qu'une opération est asynchrone (*oneway*), c'est-à-dire que le résultat est de type *void*, que tous les paramètres sont en mode *in* et qu'aucune exception ne peut être déclenchée. Malheureusement, CORBA ne spécifie pas la sémantique d'exécution d'une opération *oneway* : l'invocation peut échouer, être exécutée plusieurs fois sans que l'appelant ou l'appelé puissent en être avertis. Toutefois, dans la majorité des implantations CORBA, l'invocation d'une telle opération équivaut à un envoi fiabilisé de messages. De plus, l'OMG travaille sur les spécifications d'un service pour les communications asynchrones (CORBA Messaging).
- Un **attribut** est une manière raccourcie d'exprimer une paire d'opérations pour consulter et modifier une propriété d'un objet CORBA. Il se caractérise par un type et un nom. De plus, on peut spécifier si l'attribut est en lecture seule (*readonly*) ou consultation/modification (mode par défaut). Il faut tout de même noter que le terme IDL attribut est trompeur, l'implantation d'un attribut IDL peut être faite par un traitement quelconque, par exemple : l'implantation de l'attribut *temperature* d'un *Thermometre* consultera un capteur physique tandis que celle de l'attribut *annee_courante* du *Convertisseur* sera plutôt réalisée par une variable d'état de l'objet.

Cette description du langage OMG-IDL est toutefois incomplète. Les types *Object*, *any* et *TypeCode* seront détaillés par la suite. Pour plus d'informations sur la syntaxe du langage OMG-IDL, nous vous conseillons de vous reporter à [Geib 97] et bien sûr à [CORBA 98].

2.4. Les limites actuelles

Bien que le langage OMG-IDL soit simple et assez complet, il est actuellement limité sur certains points :

- Les types de données de base sont limités en nombre et l'utilisateur de CORBA ne peut pas en ajouter facilement. Cependant, l'OMG ne s'interdit pas d'introduire de nouveaux types au fur et à mesure des besoins, e.g. les entiers 64 bits, les réels 128 bits, les caractères internationaux et les nombres à précision fixe n'étaient pas présents dans les premières versions de la norme CORBA.
- L'impossibilité de spécifier des types intervalles qui seraient tout de même pratique pour exprimer des *Secondes*, des *Minutes* ou des *Heures*.
- L'impossibilité de sous-typer/d'étendre la définition d'une structure ou d'une union, cela peut être génant pour la réutilisation et la spécialisation de spécification IDL. Par exemple, il est impossible d'enrichir la structure *Date* dans le contexte d'un nouveau service de dates.
- L'interdiction de conflits de noms à l'intérieur d'un module ou d'une interface impliquant l'interdiction de surcharger des opérations, c'est-à-dire définir deux opérations ayant le même nom mais des signatures différentes.
- Le passage par valeur de structures de données et non d'objets. Par exemple, si l'on veut transférer un graphe d'objets, il faut l'aplatir dans une séquence de structures. Il serait préférable comme avec Java RMI de pouvoir passer l'objet graphe par valeur. L'OMG travaille actuellement sur une extension de l'OMG-IDL pour exprimer le passage d'objets par valeur, cela sera disponible dans CORBA 3.0 [OMG-OBV].

Toutefois, l'OMG est un lieu de standardisation très actif et donc si les limites soulevées ci-dessus deviennent à l'avenir des inconvénients majeurs au développement de CORBA alors on peut être sûr que de nouvelles spécifications viendront améliorer le langage OMG-IDL. D'autres limites peuvent encore être exprimées mais celles-ci nécessiteront de nouveaux langages :

- Le langage OMG-IDL ne sert pas à exprimer la sémantique des objets comme des contraintes, des pré et post conditions sur les opérations. L'expression de la sémantique permettrait de tester et valider automatiquement les objets.
- La description de la qualité de service, c'est-à-dire les caractéristiques d'une implantation, ne sont pas exprimables avec le langage OMG-IDL.
- La répartition et les liaisons entre les objets, c'est-à-dire l'architecture logicielle d'une application, ne s'expriment pas non plus *via* le langage OMG-IDL. Cependant, l'OMG a standardisé UML (Unified Modelling Language [UML 97]) qui offre une notation graphique universelle pour modéliser les applications. De plus, l'OMG travaille aussi sur un modèle de composants [OMG-MC] qui devrait palier ce problème.

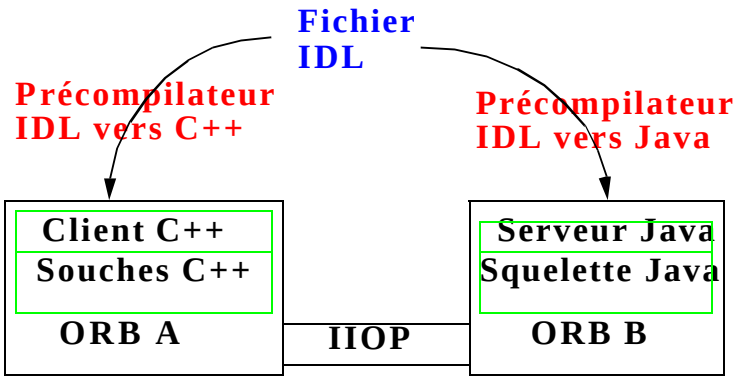
2.5. La projection vers un langage de programmation

Une projection est la traduction d'une spécification OMG-IDL dans un langage d'implantation. Pour permettre la portabilité des applications d'un bus vers un autre, les règles de projection sont normalisées et fixent précisément la traduction de chaque construction IDL en une ou des constructions du langage cible et les règles d'utilisation correcte de ces traductions. Actuellement, ces règles existent pour les langages C, C++, SmallTalk, Ada, Java et Cobol orienté objet. Nous ne détaillons pas ici ces règles, pour plus d'informations, reportez-vous à [CORBA 98]. Toutefois, voici quelques exemples de règles de projection :

Construction OMG-IDL	**Projection en C++**	**Projection en Java**
`module M { … };`	Au choix du produit CORBA : `namespace M { … }` ou `class M { … }` ou préfixe `M_`	`package M`
`interface J:I {…};`	`class J : public virtual I {…};`	`interface J extends I {…}`
`String`	`char *`	`java.lang.String`

La projection est réalisée par un pré-compilateur IDL dépendant du langage cible et de l'implantation du bus CORBA cible. Ainsi, chaque produit CORBA fournit un pré-compilateur IDL pour chacun des langages supportés. Le code des applications est alors portable d'un bus à un autre car les souches/squelettes générés s'utilisent toujours de la même manière quel que soit le produit CORBA. Par contre, le code des souches et des squelettes IDL n'est pas forcément portable car il dépend de l'implantation du bus pour lequel ils ont été généré.

La figure suivante illustre la pré-compilation d'un contrat IDL vers les langages cibles C++ et Java. Comme les deux applications vont dialoguer à travers IIOP, on peut très bien d'un côté utiliser un pré-compilateur IDL/C++ fourni par un bus A et de l'autre côté utiliser un pré-compilateur IDL/Java fourni par un bus B.



2.6. La mise en place d'une application CORBA

L'OMG n'impose pas de processus de conception et de développement d'applications distribuées : la norme CORBA laisse une entière liberté sur le choix des outils à mettre en œuvre. Néanmoins, la mise en place d'une application CORBA suit toujours à peu près le même scénario :

1. **La définition du contrat IDL** : à partir du cahier des charges, il faut définir les objets composant l'application à l'aide d'une méthodologie orientée objet (e.g. OMT). Cette modélisation est ensuite traduite sous la forme de contrats IDL composés des interfaces des objets et des types de données utiles aux échanges d'informations entre les objets.
2. **La pré-compilation du contrat IDL** : les interfaces des objets sont décrites dans des fichiers texte. Le pré-compilateur prend en entrée un tel fichier et opère un contrôle syntaxique et sémantique des définitions OMG-IDL contenues dans ce fichier. Le pré-compilateur peut aussi charger ces définitions dans le référentiel des interfaces. C'est la partie frontale commune à tous les pré-compilateurs IDL.
3. **La projection vers les langages de programmation** : le pré-compilateur IDL génère le code des souches qui sera utilisé par les applications clientes des interfaces décrites dans le fichier IDL, ainsi que le code des squelettes pour les programmes serveurs implantant ces types. Cette projection est spécifique à chaque langage de programmation : un environnement CORBA fournit un pré-compilateur IDL pour chacun des langages supportés.
4. **L'implantation des interfaces IDL** : en complétant et/ou en réutilisant le code généré pour les squelettes, le développeur implante les objets dans le langage de son choix ou dans le langage le mieux adapté à la réalisation de ses objets. Il doit tout de même se plier aux règles de projection vers ce langage.
5. **L'implantation des serveurs d'objets** : le développeur doit écrire les programmes serveurs qui incluent l'implantation des objets et les squelettes pré-générés. Ces programmes contiennent le code pour se connecter au bus, instancier les objets racines du serveur, rendre publiques les références sur ces objets à l'aide par exemple du service Nommage et se mettre en attente de requêtes pour ces objets. Le nombre de programmes serveurs est tout de même souvent limité.
6. **L'implantation des applications clientes des objets** : le développeur écrit un ensemble de programmes clients qui agissent sur les objets en les parcourant et en invoquant des opérations sur ceux-ci. Ces programmes incluent le code des souches, le code pour l'interface Homme-Machine et le code spécifique à l'application. Les clients obtiennent les références des objets serveurs en consultant le service Nommage. Il faut tout de même souvent développer une multitude de programmes clients des objets en fonction des rôles et des activités de chacun des utilisateurs de l'application répartie.
7. **L'installation et la configuration des serveurs** : cette phase consiste à installer dans le référentiel des implantations les serveurs pour automatiser leur activation lorsque des requêtes arrivent pour leurs objets.
8. **La diffusion et la configuration des clients** : une fois les programmes clients mis au point, il est nécessaire de diffuser les exécutables sur les sites de leur future utilisation et de configurer les sites clients pour qu'ils sachent où se trouvent les serveurs utilisés.
9. **L'exécution répartie de l'application** : enfin, l'exploitation de l'application peut commencer. Le bus d'objets répartis CORBA assure alors les communications entre les programmes clients et les objets via le protocole IIOP.

Quel que soit le langage employé avec CORBA et la nature de l'application, le développeur est toujours censé suivre le cycle de développement que nous venons de présenter. La section suivante présente dans les détails la mise en œuvre de ce scénario.

3. Une application d'annuaires

Après ce tour d'horizon des concepts proposés par l'OMG, nous allons passer à la mise en pratique du bus CORBA. Nous avons choisi pour cela de présenter la réalisation d'une application de gestion d'annuaires. Celle-ci est composée d'objets CORBA modélisant des répertoires d'adresses de personnes sur Internet. Ces objets sont gérés par des serveurs permettant de les créer, détruire et retrouver. Des exemples d'applications clientes illustrent l'utilisation de ces objets. Certains de ces clients sont prévenus automatiquement de tout changement dans les répertoires qui les intéressent (selon un principe de notification).

Cette application nous permet de voir concrètement l'utilisation du langage OMG-IDL et l'implantation d'objets CORBA, de serveurs et de clients avec les langages C++ , Java et CorbaScript. De plus, nous abordons l'utilisation du service Nommage, le cycle de vie des objets et la notification d'applications clientes.

Dans ce document, nous ne présentons que les fragments de code significatifs pour la compréhension. Toutefois, l'ensemble du code de cette application est disponible sur notre site WWW. Il fonctionne avec le bus CORBA ORBacus (connu aussi sous le nom d'OmniBroker [OOC 98]) mais doit pouvoir facilement être porté sur d'autres bus. Le choix de ce bus est justifié par le fait qu'il est gratuit pour des utilisations non commerciales, qu'il fournit une liaison avec les langages C++ et Java et surtout qu'il est de très bonne facture.

3.1. Le contrat IDL d'un répertoire d'adresses

L'ensemble du contrat IDL de notre application d'annuaires est introduit au fur et à mesure de sa construction. Le répertoire d'adresses est le type d'objets principalement manipulé par l'application, voici donc sa spécification OMG-IDL :

```
#include <date.idl>          // Réutilisation de la spécification du service de dates.
#pragma prefix "lifl.fr"     // C'est un contrat encore défini au LIFL :-)
module annuaire {            // Contrat IDL de l'application répartie.
  typedef string Nom;        // Définition du concept nom d'une personne.
  typedef sequence<Nom> DesNoms;// Ensemble de noms.

  struct Personne {                 // Informations associées à 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;    //     - date de naissance.
    // Réutilisation de la structure Date par préfixage du module date.
    // Le premier '::' n'est pas indispensable.
  };

  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 personne) raises(Inconnu);
    Personne obtenirPersonne (in Nom nom) raises(Inconnu);
    DesNoms listerNoms ();
  };

  typedef sequence<Personne> DesPersonnes; // Ensemble de personnes (utile plus tard en C++).
};
```

L'ensemble du contrat de l'application est regroupé dans le module *annuaire*. Les répertoires sont décrits par l'interface *Repertoire* qui fournit l'attribut *libelle* informant sur le contenu du répertoire et des opérations pour ajouter, retirer, modifier, obtenir et lister les personnes contenues par celui-ci. La structure *Personne* regroupe les

caractéristiques décrivant une personne : son nom, des informations diverses, son numéro de téléphone, sa boîte électronique, son adresse WWW et sa date de naissance (ici nous réutilisons la spécification du service de dates). L'alias *Nom* permet une meilleure lisibilité du concept de nom d'une personne. La séquence *DesNoms* est nécessaire pour pouvoir retourner l'ensemble des noms des personnes d'un répertoire. Les exceptions *ExisteDeja* et *Inconnu* permettent de signaler les cas exceptionnels dus à l'ajout d'une personne ayant un nom déjà recensé dans le répertoire ou à la recherche d'une personne inconnue.

3.2. La projection du contrat IDL

Le passage de ce contrat IDL dans des pré-compilateurs permet d'obtenir les souches et les squelettes pour les langages cibles. Le module *annuaire* est projeté vers l'espace de désignation C++ *annuaire::* ou *annuaire_* selon le produit CORBA utilisé car la projection C++ d'un module IDL peut se faire de deux manières différentes (ORBacus a choisi la seconde). Le C++ ne fournissant pas de ramasse-miettes par défaut, la projection IDL/C++ fournit des classes C++ pour gérer automatiquement la libération de la mémoire allouée par le bus CORBA (noms de type ayant comme extension *_var*). Les types générés avec l'extension *_ptr* sont juste des définitions de types pointeurs. À titre illustratif, voici une version simplifiée de la projection du contrat IDL avec le pré-compilateur IDL/C++ d'ORBacus (la partie implantation n'est pas présentée car elle dépend des spécificités techniques de ce bus) :

```
// Fichier : annuaire.h
#include <date.h>

typedef char* annuaire_Nom;                 // Projection du typedef annuaire::Nom.
typedef CORBA_String_var annuaire_Nom_var;

typedef OBStrSeq annuaire_DesNoms;          // Projection de la séquence annuaire::DesNoms.
typedef OBSeqVar< OBStrSeq > annuaire_DesNoms_var;

struct annuaire_Personne {                  // Projection de la structure annuaire::Personne.
  // constructeur, destructeur, opérateur de copie, ...
  annuaire_Nom_var nom;
  CORBA_String_var informations;
  CORBA_String_var telephone;
  CORBA_String_var email;
  CORBA_String_var url;
  date_Date date_naissance;
};
typedef OBVarVar< annuaire_Personne > annuaire_Personne_var;

class annuaire_Repertoire;                  // Projection de l'interface annuaire::Repertoire.
typedef annuaire_Repertoire* annuaire_Repertoire_ptr;
typedef OBObjVar< annuaire_Repertoire > annuaire_Repertoire_var;
class annuaire_Repertoire : virtual public CORBA_Object {
public:
  virtual char* libelle();
  struct ExisteDeja : public CORBA_UserException { ... };
  struct Inconnu : public CORBA_UserException { ... };
  virtual void ajouterPersonne(const annuaire_Personne& personne);
  virtual void retirerPersonne(const char* nom);
  virtual void modifierPersonne(const char* nom, const annuaire_Personne& personne);
  virtual annuaire_Personne* obtenirPersonne(const char* nom);
  virtual annuaire_DesNoms* listerNoms();

  static annuaire_Repertoire_ptr _narrow(CORBA_Object_ptr);
  ... fonctions spécifiques pour CORBA ou pour l'implantation ORBacus ...
};

typedef OBVarSeq< annuaire_Personne > annuaire_DesPersonnes;
typedef OBSeqVar< OBVarSeq< annuaire_Personne > > annuaire_DesPersonnes_var;
```

En Java, le contrat est projeté vers le package *fr.lifl.annuaire* car le langage Java offre la possibilité de définir l'organisation propriétaire d'un package. Le fragment suivant illustre la projection Java de la structure IDL *annuaire::Personne*.

```
package fr.lifl.annuaire;
final public class Personne
{
  public Personne() { }
  public Personne(String _ob_a0, String _ob_a1, String _ob_a2, String _ob_a3, String _ob_a4,
                  fr.lifl.date.Date _ob_a5)
```

```
  {
    nom = _ob_a0;
    informations = _ob_a1;
    telephone = _ob_a2;
    email = _ob_a3;
    url = _ob_a4;
    date_naissance = _ob_a5;
  }

  public String nom;
  public String informations;
  public String telephone;
  public String email;
  public String url;
  public fr.lifl.date.Date date_naissance;
}
```

Les deux sections suivantes illustrent l'utilisation de la projection IDL vers les langages C++ et Java.

3.3. L'implantation du répertoire

Comme on peut le constater, la description OMG-IDL du type d'objets *Repertoire* est totalement abstraite et complètement indépendante de tout aspect lié à son implantation. Il est facilement imaginable d'utiliser une base de données relationnelle (ou un fichier !) pour conserver l'état d'un répertoire de personnes. Dans cette section, nous allons voir comment facilement implanter ce répertoire en C++, Java et CorbaScript.

3.3.1. L'implantation en C++

Une implantation possible du répertoire est fournie par la classe C++ *RepertoireImpl* suivante. Celle-ci doit hériter du squelette généré par le pré-compilateur IDL/C++ (l'identificateur *annuaire_Repertoire_skel* est spécifique au bus ORBacus). Elle doit alors implanter l'attribut et les opérations de l'interface *Repertoire*. Pour cela, elle encapsule dans les deux variables d'instances *le_libelle* et *personnes* l'état du répertoire. Ici, la structure de données choisie est non persistante afin de simplifier l'exemple. De plus, la projection C++ de la séquence IDL *annuaire::DesPersonnes* nous fournit facilement une structure de stockage des personnes.

```
// Importation du squelette C++ de l'interface IDL annuaire::Repertoire.
#include <annuaire_skel.h>
//  * l'extension _skel est spécifique à la projection C++ d'ORBacus.
//  * ORBacus utilise la concaténation de noms pour désigner les
//    les définitions contenues dans un module IDL.
//    e.g.: en IDL, annuaire::Personne -> en C++, annuaire_Personne

class RepertoireImpl : public virtual annuaire_Repertoire_skel
{
  CORBA_String_var le_libelle;              // Une chaîne de caractères.
  annuaire_DesPersonnes personnes;          // L'ensemble des personnes.

  CORBA_Long rechercher (const char* nom)   // Recherche de la position d'une personne.
  {
    for(CORBA_ULong i=0; i<personnes.length(); i++)
      if (strcmp(personnes[i].nom, nom) == 0)
        return i;     // Si trouvé.
    return -1;        // Si pas trouvé.
  }

public:
  RepertoireImpl (const char* libelle)      // Constructeur.
    : annuaire_Repertoire_skel()     // Appel du constructeur du squelette.
    , le_libelle(libelle)            // Initialisation par copie implicite du libellé.
    , personnes()                    // Initialisation de la séquence de personnes.
  {}

  ~RepertoireImpl () {}                     // Destructeur.

  virtual char* libelle()
  { // En C++, la valeur retournée doit être une copie (c'est une règle de projection).
    return CORBA_string_dup(le_libelle);    // Duplique une chaîne de caractères.
  }

  virtual void ajouterPersonne(const annuaire_Personne& personne)
  {
```

```
    CORBA_Long pos = rechercher (personne.nom);
    if (pos != -1) throw annuaire_Repertoire::ExisteDeja (personne.nom);
    CORBA_ULong len = personnes.length();
    personnes.length(len+1);               // Ajoute une personne à la fin de l'ensemble.
    personnes[len] = personne;
  }

  virtual void retirerPersonne(const char* nom)
  {
    CORBA_Long pos = rechercher (nom);
    if (pos == -1) throw annuaire_Repertoire::Inconnu (nom);
    CORBA_ULong len = personnes.length();
    for (CORBA_ULong i=pos; i<len-1; i++)   // Retire la personne de l'ensemble.
      personnes[i] = personnes[i+1];
    personnes.length(len-1);
  }

  virtual void modifierPersonne(const char* nom, const annuaire_Personne& personne)
  {
    CORBA_Long pos = rechercher (nom);
    if (pos == -1) throw annuaire_Repertoire::Inconnu (nom);
    personnes[pos] = personne;
  }

  virtual annuaire_Personne* obtenirPersonne(const char* nom)
  {
    CORBA_Long pos = rechercher (nom);
    if (pos == -1) throw annuaire_Repertoire::Inconnu (nom);
    // En C++, la valeur retournée doit être une copie.
    return new annuaire_Personne(personnes[pos]);
  }

  virtual annuaire_DesNoms* listerNoms()
  {
    CORBA_ULong len = personnes.length();
    annuaire_DesNoms* resultat = new annuaire_DesNoms;
    resultat->length(len);
    for(CORBA_ULong i=0; i<len; i++)
      (*resultat)[i] = personnes[i].nom;
    return resultat;
  }
};
```

Comme on peut le constater, l'implantation C++ d'un objet CORBA doit suivre quelques règles imposées : la signature de l'implantation des opérations doit être respectée, le retour d'une opération (attribut) doit toujours se faire par copie. Le mécanisme C++ standard de déclenchement des exceptions (*throw*) permet de provoquer les exceptions IDL. Les séquences IDL sont représentées par des classes C++ offrant des opérations pour consulter/modifier leur taille (*length*) et des opérateurs pour consulter/modifier les éléments (*[]*). Toutefois, même si ces règles sont un peu contraignantes et compliquées au premier abord, elles permettent d'écrire relativement simplement du code portable pour un ORB/C++ (au renommage près de l'extension *_skel*).

3.3.2. L'implantation en Java

Notre répertoire peut être aussi simplement implanté par la classe Java *RepertoireImpl*. Le pré-compilateur IDL/Java génère le package Java *fr.lifl.annuaire* comme projection du contrat IDL. La classe *_RepertoireImplBase* fournit alors le squelette ou la base pour implanter le répertoire. Ainsi, la classe *RepertoireImpl* en hérite et fournit une implantation de l'attribut et des opérations IDL. Ici, nous utilisons une table de hachage pour stocker les personnes du répertoire. Une version persistante du répertoire aurait encapsulé l'utilisation des bibliothèques Java de sérialisation ou d'accès aux bases de données (JDBC).

```
public class RepertoireImpl extends fr.lifl.annuaire._RepertoireImplBase
{
  protected String le_libelle;
  protected java.util.Hashtable personnes;

  public RepertoireImpl (String le_libelle)
  {
    this.le_libelle = le_libelle;
    this.personnes = new java.util.Hashtable();
  }

  public String libelle()
```

```
  {
    return this.le_libelle;
  }

  public void ajouterPersonne(fr.lifl.annuaire.Personne personne)
        throws fr.lifl.annuaire.RepertoirePackage.ExisteDeja
  {
    if (this.personnes.containsKey(personne.nom))
      throw new fr.lifl.annuaire.RepertoirePackage.ExisteDeja(personne.nom);
    this.personnes.put(personne.nom,personne);
  }

  public void retirerPersonne(String nom)
        throws fr.lifl.annuaire.RepertoirePackage.Inconnu
  {
    if (this.personnes.remove(nom) == null)
      throw new fr.lifl.annuaire.RepertoirePackage.Inconnu(nom);
  }

  public void modifierPersonne(String nom, fr.lifl.annuaire.Personne personne)
        throws fr.lifl.annuaire.RepertoirePackage.Inconnu
  {
    if (this.personnes.remove(nom) == null)
      throw new fr.lifl.annuaire.RepertoirePackage.Inconnu(nom);
    this.personnes.put(personne.nom,personne);
  }

  public fr.lifl.annuaire.Personne obtenirPersonne(String nom)
        throws fr.lifl.annuaire.RepertoirePackage.Inconnu
  {
    fr.lifl.annuaire.Personne resultat = (fr.lifl.annuaire.Personne)this.personnes.get(nom);
    if (resultat == null)
      throw new fr.lifl.annuaire.RepertoirePackage.Inconnu(nom);
    return resultat;
  }

  public String[] listerNoms()
  {
    String[] resultat = new String[this.personnes.size()];
    int i = 0;
    for (java.util.Enumeration e = this.personnes.keys() ; e.hasMoreElements() ; i++) {
      resultat[i] = (String)e.nextElement();
    }
    return resultat;
  }
}
```

Comme on peut le constater, la structure de l'implantation Java est totalement similaire à celle en C++. Toutefois, la liaison IDL/Java est nettement plus simple à utiliser principalement parce que le langage Java est moins complexe que le langage C++. Notez que ce code aurait pu être plus concis en utilisant les clauses d'importation du langage Java (e.g. *import fr.lifl.annuaire.** et *fr.lifl.annuaire.RepertoirePackage.**). De plus, ce code est totalement portable sur d'autres implantations CORBA/Java aussi bien les sources que les binaires.

3.3.3. L'implantation en CorbaScript

CorbaScript est un langage de scripts dédié au bus CORBA. Ce langage est développé par notre équipe de recherche au LIFL [CS] et il est en cours de standardisation auprès de l'OMG [OMG-CSL]. L'idée maîtresse de ce nouveau langage est de pouvoir implanter et utiliser n'importe quel objet CORBA en profitant des bénéfices (et inconvénients) d'un environnement de scripts : interactivité, typage dynamique, facilité d'utilisation et d'apprentissage. Sa caractéristique majeure réside dans sa liaison transparente avec le système de typage du bus CORBA : toute définition OMG-IDL est directement accessible depuis les scripts, il n'y a pas de génération de souches ou de squelettes IDL, la projection IDL/CorbaScript est donc transparente pour les utilisateurs. Nous expliquerons dans la section 4 comment cela est techniquement réalisé grâce aux mécanismes dynamiques du bus CORBA.

Comme le lecteur n'est pas (encore :-) un adepte de CorbaScript, nous présentons plus en détail les fonctionnalités de ce nouveau langage. CorbaScript offre un typage dynamique : le type des variables n'est pas explicitement précisé, le contrôle du typage est réalisé dynamiquement durant l'exécution. De plus, c'est un langage orienté objet afin de mieux refléter les objets du monde CORBA. Ainsi, l'interface *annuaire::Repertoire* est implantée par la classe *RepertoireImpl*. La méthode *__RepertoireImpl__* initialise les instances créées à partir

de cette classe. Toute méthode (i.e. *proc*) doit avoir un premier paramètre explicite qui désigne l'instance réceptrice de l'invocation (e.g. *self*). Les attributs d'une instance sont déclarés à leur première affectation (e.g. *self.personnes* et *self.libelle*). Il est toujours obligatoire de préfixer l'accès à un attribut ou l'invocation d'une méthode par la référence de l'instance (e.g. « *self.* »). L'implantation des attributs OMG-IDL est réalisée par des méthodes de consultation (le préfixe *_get_* suivi du nom de l'attribut, e.g. *_get_libelle*) ou de modification (avec le préfixe *_set_* et un paramètre de plus pour la nouvelle valeur). L'implantation des opérations est réalisée par une méthode de même nom et ayant un paramètre référençant l'instance et autant de paramètres que dans la signature de l'opération OMG-IDL (e.g. *ajouterPersonne*).

```
class RepertoireImpl                        # Une classe CorbaScript.
{
  proc __RepertoireImpl__(self,libelle)     # La méthode d'initialisation des instances.
  {                                         # self référence l'instance réceptrice.
    self.libelle = libelle                  # L'affectation/déclaration d'un attribut.
    self.personnes = []                     # Ici, l'attribut personnes est un tableau vide.
  }

  proc chercherPosition (self,nom)   # Recherche la position dans le tableau des personnes.
  {                                  # Equivalent de la fonction C++ 'rechercher' précédente.
    i = 0
    for p in self.personnes {        # Itération sur les éléments du tableau.
      if (p.nom == nom) return i;
      i = i + 1
    }
    return -1
  }

  proc _get_libelle(self)            # Implantation de l'attribut OMG-IDL 'libelle'.
  {
    return self.libelle
  }

  proc ajouterPersonne(self,personne) # Implantation de l'opération OMG-IDL 'ajouterPersonne'.
  {
    p = self.chercherPosition(personne.nom)
    if (p != -1)                           # Accès direct aux exceptions CORBA.
      throw annuaire.Repertoire.ExisteDeja (personne.nom)
    self.personnes.append(personne)  # Ajouter à la fin du tableau.
  }

  proc retirerPersonne(self,nom)            # Pour l'opération OMG-IDL `retirerPersonne'.
  {
    p = self.chercherPosition(nom)
    if (p == -1)
      throw annuaire.Repertoire.Inconnu (nom)
    self.personnes.delete(p)         # Retirer l'élément en position p.
  }

  proc modifierPersonne(self,nom,personne)  # Pour l'opération OMG-IDL 'modifierPersonne'.
  {
    p = self.chercherPosition(nom)
    if (p == -1)
      throw annuaire.Repertoire.Inconnu (nom)
    self.personnes[p] = personne     # Modifier l'élément en position p.
  }

  proc obtenirPersonne(self,nom)            # Pour l'opération OMG-IDL 'obtenirPersonne'.
  {
    p = self.chercherPosition(nom)
    if (p == -1)
      throw annuaire.Repertoire.Inconnu (nom)
    return self.personnes[p]
  }

  proc listerNoms(self)                     # Pour l'opération OMG-IDL `listerNoms'.
  {
    resultat = []                           # tableau vide.
    for p in self.personnes                 # Parcours des personnes présentes.
      resultat.append(p.nom)                # Ajouter leur nom au tableau.
    return resultat                         # Conversion automatique et implicite
  }                                         # d'un tableau en séquence OMG-IDL.
}
```

Comme on peut le constater ci-dessus, l'implantation en CorbaScript de notre répertoire n'est pas foncièrement différente de celles réalisées en C++ et Java. Les différences sont cachées aux développeurs. La première est que l'accès aux définitions OMG-IDL depuis CorbaScript est directe et ne nécessite aucune génération de souches (e.g. *annuaire.Repertoire*, *annuaire.Repertoire.ExisteDeja* et *annuaire.Repertoire.Inconnu*). La seconde est que le script précédent peut être fourni interactivement à un interpréteur CorbaScript.

3.4. L'utilisation d'un répertoire

Maintenant, nous allons regarder l'utilisation de la souche IDL de l'interface *Repertoire*. Pour cela, nous présentons un fragment de traitement en C++, en Java et en CorbaScript qui invoque l'ensemble des opérations d'un objet répertoire quelconque. Ce traitement affiche le libellé du répertoire, ajoute une nouvelle personne dans le répertoire, modifie cet enregistrement, liste le contenu du répertoire et finalement détruit la personne enregistrée. Comme vous pourrez le constater, ce traitement est totalement indépendant de l'implantation et de la localisation du répertoire : on peut utiliser le traitement C++ sur l'implantation Java ou CorbaScript, ou réaliser toute autre combinaison.

3.4.1. L'invocation en C++

En C++, un objet CORBA est référencé par un pointeur sur la classe souche représentant son interface (*annuaire_Repertoire**). L'invocation d'opérations OMG-IDL s'exprime alors simplement par l'appel de méthodes C++ (notation ->). Les variables de type *_var* gèrent la libération automatique de la mémoire (*c.f.* section 3.2). L'initialisation d'un champ *string* doit obligatoirement se faire par copie soit explicite (*CORBA_string_dup*) soit implicite en passant une chaîne constante (*const char**). Le passage de paramètres en mode *in* se fait par valeur. Le mécanisme C++ standard de traitement des exceptions (*try*/*catch*) permet d'intercepter les exceptions IDL.

```
// Importation de la souche annuaire générée par le pré-compilateur IDL.
#include <annuaire.h>

void traitement (annuaire_Repertoire* repertoire)
{
  const char* MOIS[] = {"Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet",
    "Aout", "Septembre", "Octobre", "Novembre", "Décembre" };

  CORBA_String_var libelle = repertoire->libelle();
  cout << "Libelle du répertoire consulté : " << libelle << endl;

  annuaire_Personne personne;
  personne.nom = CORBA_string_dup("Merle Philippe");
  personne.informations = (const char*)"Enseignant/Chercheur";
  personne.telephone = (const char*)"03.20.43.47.21";
  personne.email = (const char*)"merle@lifl.fr";
  personne.url = (const char*)"http://www.lifl.fr/~merle";
  personne.date_naissance.le_jour = 4;
  personne.date_naissance.le_mois = date_Mars;
  personne.date_naissance.l_annee = 1969;

  try {
    repertoire->ajouterPersonne(personne);
  } catch (annuaire_Repertoire::ExisteDeja& ) {
    cout << "ATTENTION : Merle a déjà été ajouté" << endl;
  }

  repertoire->modifierPersonne ("Merle Philippe",personne);

  annuaire_DesNoms_var noms = repertoire->listerNoms();
  for (CORBA_ULong i=0; i<noms->length(); i++) {
    annuaire_Personne_var p = repertoire->obtenirPersonne(noms[i]);
    cout << "Nom : " << p->nom << endl;
    cout << "Informations : " << p->informations << endl;
    cout << "Téléphone : " << p->telephone << endl;
    cout << "Email : " << p->email << endl;
    cout << "URL : " << p->url << endl;
    const date_Date& d = p->date_naissance;
    cout << "Date de naissance : " << d.le_jour << " "
         << MOIS[d.le_mois] << " " << d.l_annee << endl;
  }

  // Attention, ici, la gestion des exceptions est omise.
```

```
  repertoire->retirerPersonne ("Merle Philippe");
}
```

3.4.2. L'invocation en Java

Notre traitement s'exprime aussi simplement en Java avec seulement quelques différences. Les interfaces IDL sont représentées par des interfaces Java (*fr.lifl.annuaire.Repertoire*). L'invocation d'opérations IDL utilise la notation pointée d'appel de méthodes Java. Les structures IDL sont représentées par des classes Java et doivent donc être allouées dynamiquement. Les énumérations IDL sont aussi représentées par des classes fournissant des champs statiques pour chaque élément de l'énumération et une opération de conversion vers un entier (*value*). La libération de la mémoire allouée par CORBA n'est pas ici un problème car le langage Java fournit en standard un ramasse-miettes. Les exceptions IDL sont gérées par l'intermédiaire du mécanisme standard de Java (*try*/*catch*/*finally*).

```
static void traitement (fr.lifl.annuaire.Repertoire repertoire) throws Exception
{
  String [] MOIS = { "Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet",
         "Aout", "Septembre", "Octobre", "Novembre", "Décembre" };

  String libelle = repertoire.libelle();
  System.out.println("Libelle du répertoire consulté : " + libelle);

  fr.lifl.annuaire.Personne personne = new fr.lifl.annuaire.Personne();
  personne.nom = "Merle Philippe";
  personne.informations = "Enseignant/Chercheur";
  personne.telephone = "03.20.43.47.21";
  personne.email = "merle@lifl.fr";
  personne.url = "http://www.lifl.fr/~merle";
  personne.date_naissance = new fr.lifl.date.Date((short)4,
                       fr.lifl.date.Mois.Mars,(short)1969);
  try {
    repertoire.ajouterPersonne(personne);
  } catch (fr.lifl.annuaire.RepertoirePackage.ExisteDeja ed) {
    System.out.println ("ATTENTION : Merle a déjà été ajouté");
  }
  repertoire.modifierPersonne ("Merle Philippe",personne);

  String[] noms = repertoire.listerNoms();
  for (int i=0; i<noms.length; i++) {
    personne = repertoire.obtenirPersonne(noms[i]);
    System.out.println("Nom : " + personne.nom);
    System.out.println("Informations : " + personne.informations);
    System.out.println("Téléphone : " + personne.telephone);
    System.out.println("Email : " + personne.email);
    System.out.println("URL : " + personne.url);
    fr.lifl.date.Date d = personne.date_naissance;
    System.out.println("Date de naissance : " + d.le_jour + " "
                     + MOIS[d.le_mois.value()] + " " + d.l_annee);
  }
  repertoire.retirerPersonne ("Merle Philippe");
}
```

3.4.3. L'invocation en CorbaScript

Le traitement sur le répertoire s'exprime très simplement en CorbaScript en invoquant les attributs et opérations de l'interface *annuaire::Repertoire*. Tous les types OMG-IDL sont directement et naturellement accessibles, e.g. *annuaire.Personne*, *date.Date*, *date.Mois* et *annuaire.Repertoire.ExisteDeja*.

```
proc traitement (repertoire)               # Une procédure CorbaScript.
{
  # Consultation de l'attribut OMG-IDL 'libelle'.
  println("Libellé du répertoire consulté : ", repertoire.libelle)

  # Création d'une structure Personne.
  personne = annuaire.Personne ( "Merle Philippe",
                                 "Enseignant/Chercheur",
                                 "03.20.43.47.21",
                                 "merle@lifl.fr",
                                 "http://www.lifl.fr/~merle",
                                 date.Date(4,date.Mois.Mars,1969)
                               )
  try {                                          # Gérant d'exceptions.
```

```
    repertoire.ajouterPersonne(personne)           # Invocation d'une opération OMG-IDL.
  } catch (annuaire.Repertoire.ExisteDeja ed) {
    println ("ATTENTION : Merle a déjà été ajouté")
  }

  repertoire.modifierPersonne ("Merle Philippe",personne)

  noms = repertoire.listerNoms()
  for i in range(0,noms.length-1) {         # Parcourir les éléments de la séquence DesNoms.
    personne = repertoire.obtenirPersonne(noms[i])
    println("Nom : ", personne.nom)
    println("Informations : ", personne.informations)
    println("Téléphone : ", personne.telephone)
    println("Email : ", personne.email)
    println("URL : ", personne.url)
    d = personne.date_naissance
    println("Date de naissance : ", d.le_jour, " ",
            d.le_mois, " ", d.l_annee)
  }
  repertoire.retirerPersonne ("Merle Philippe")
}
```

3.4.4. Une première conclusion

Comme on vient de le voir, implanter et utiliser un objet CORBA depuis un langage de programmation comme le C++ et Java n'a rien d'extraordinaire : cela paraît assez naturel. Cette simplicité est liée au fait que les règles de projection d'IDL tentent de perturber au minimum les habitudes de programmation du langage cible en utilisant au mieux les mécanismes/constructions qu'il offre. Cela est légèrement différent avec CorbaScript : ce langage de scripts a été spécialement conçu pour simplifier l'utilisation et l'implantation d'objets CORBA, la projection IDL/CorbaScript est totalement implicite et prise en charge par l'interpréteur de ce langage.

3.5. Retour sur le bus CORBA

Avant de passer à la construction des applications serveurs et clientes, nous allons revenir sur le bus CORBA pour présenter quelques unes de ces composantes à savoir le module *CORBA*, l'interface *Object*, l'interface *ORB* et l'adaptateur d'objets.

3.5.1 Le module CORBA

Les composantes du bus CORBA sont décrites elles aussi par l'intermédiaire du langage OMG-IDL. Celles-ci sont regroupées dans le module IDL *CORBA* défini par l'OMG, i.e. le pragma *prefix* « omg.org ». La projection de ce module constitue alors la bibliothèque de programmation du bus CORBA depuis un langage cible. Il contient en autres l'ensemble des exceptions systèmes que peut lever le bus comme, par exemple, un problème de communication (*COMM_FAILURE*).

```
#pragma prefix "omg.org"
module CORBA {
  // L'ensemble des composantes du bus CORBA.
  exception COMM_FAILURE { … };
  // Autres exceptions systèmes.
};
```

En C++, le contenu de ce module est accessible par le préfixe *CORBA::* ou *CORBA_* selon le choix de projection fait par le produit CORBA utilisé. En Java, ce module est représenté par le package *org.omg.CORBA*. En CorbaScript, ce module est reflété par l'objet *CORBA*. Ainsi l'exception système *COMM_FAILURE* est accessible par *CORBA::COMM_FAILURE* ou *CORBA_COMM_FAILURE* en C++, *org.omg.CORBA.COMM_FAILURE* en Java et *CORBA.COMM_FAILURE* en CorbaScript.

3.5.2. L'interface Object

L'interface *Object* contenue dans le module *CORBA* est implicitement héritée par toute interface IDL (les souches IDL en héritent). Cette interface définit le concept de référence d'objet CORBA et son implantation encapsule l'adresse de l'objet CORBA référencé (ou IOR). Elle fournit un ensemble d'opérations de manipulation de base applicables sur n'importe quelle référence d'objet CORBA. Voici un fragment de sa spécification IDL :

```
module CORBA {
  interface Object {
    Object _duplicate();                      // Duplique une référence d'objet CORBA.
    void _release();                          // Libère une référence d'objet.
    boolean _is_nil ();                       // Teste si une référence ne dénote aucun objet.
    boolean _non_existent();                  // Teste si un objet référencé n'existe plus.
    boolean _is_equivalent(in Object that);   // Teste si 2 références désignent la même IOR.
    long _hash(in long maximum);              // Calcule une clé de hachage.
    boolean _is_a(in string type_identifier); // Teste si un objet est d'un type donné.

    // D'autres opérations, voir la section sur les mécanismes dynamiques.
    InterfaceDef _get_interface();
    Request _request(in string s);
    Request _create_request(in Context ctx, in string operation,
                            in NVList arg_list, in NamedValue result);
    Request _create_request2(in Context ctx, in string operation,
                             in NVList arg_list, in NamedValue result,
                             in ExceptionList excepts, in ContextList contexts);
  };
};
```

3.5.3. L'interface ORB

L'interface *ORB* modélise le bus CORBA. Elle permet de contrôler le comportement du bus, de créer les autres objets représentant les composantes du bus, de convertir l'IOR d'une référence d'objet en chaîne de caractères et vice-versa (i.e. *object_to_string* et *string_to_object*) et d'obtenir les références des objets « notoires » (i.e. *list_initial_services* et *resolve_initial_references*). De plus, une opération contenue dans le module *CORBA* permet d'initialiser l'objet *ORB* (i.e. *ORB_init*). Voici une partie de sa spécification IDL :

```
module CORBA {
  typedef string ORBid;
  typedef sequence<string> arg_list;
  ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);  // Obtenir l'ORB.
  // ATTENTION, normalement une opération doit être dans une interface => ici, pseudo IDL

  interface ORB {
    string object_to_string (in Object obj);     // Une référence d'objet vers une chaîne IOR.
    Object string_to_object (in string str);     // Une chaîne IOR vers une référence d'objet.

    typedef string ObjectId;
    typedef sequence<ObjectId> ObjectIdList;
    ObjectIdList list_initial_services ();      // La liste des objets notoires.
    Exception InvalidName {};                   // Obtenir un objet notoire.
    Object resolve_initial_references (in ObjectId identifier) raises(InvalidName);

    // Autres opérations.
  };
};
```

Les objets « notoires » sont les premiers objets utiles à l'exploiration du bus CORBA, e.g. le référentiel des interfaces *InterfaceRepository* et le service Nommage *NameService*. Les références de ces objets sont maintenues en interne par le bus CORBA. Notez que la configuration de ce mécanisme est dépendante de chaque produit CORBA : cela peut être un fichier de configuration (ORBacus) ou un démon (Visibroker, Orbix) ou bien tout autre moyen.

3.5.4. L'adaptateur d'objets

La structure d'accueil, c'est-à-dire le serveur, de l'implantation d'un objet CORBA fournit un espace mémoire pour l'état des objets et un contexte d'exécution des opérations. L'adaptateur d'objets est l'abstraction CORBA permettant à une structure d'accueil 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.

De plus, l'adaptateur d'objets isole le bus des différentes technologies pouvant être employées pour l'implantation de la structure d'accueil. Pour cela, différents types d'adaptateurs sont envisageables :

- **BOA** ou Basic Object Adapter : les structures d'accueil sont matérialisées par des processus systèmes.
- **OODA** ou Object-Oriented Database Adapter : la structure d'accueil est une base de données orientée objet.
- **LOA** ou Library Object Adapter : le code d'implantation des objets est stocké dans des bibliothèques chargées dans l'espace des applications clientes.
- **COA** ou Card Object Adapter : l'implantation des objets est stockée dans une carte à microprocesseur. Cet adaptateur a été expérimenté conjointement par le LIFL et la société Gemplus.
- **POA** ou Portable Object Adapter : l'implantation des objets est réalisée par des objets fournis par un langage de programmation.

Les premières versions de CORBA spécifiaient uniquement le BOA. Malheureusement, le BOA fut sous-spécifié entraînant ainsi des incompatibilités entre différents produits CORBA. L'OMG a depuis défini le nouvel adaptateur d'objets POA dans la norme CORBA 2.2. Pour plus d'informations, reportez-vous à la norme [CORBA 98] et aussi à la série d'articles de [Schmidt 98]. Malheureusement, à l'heure où nous rédigeons ce document, un seul produit CORBA fournit le POA, c'est l'ORB ACE/TAO [TAO 98]. Ainsi, nous utiliserons dans ce document une partie de la spécification du BOA fourni par ORBacus. Celle-ci fournit l'interface *BOA* offrant une opération pour se mettre en attente des requêtes aux objets (*impl_is_ready)* et une opération dans l'ORB pour initialiser l'adaptateur d'objets (*BOA_init*).

```
// Le BOA est devenu obsolète depuis CORBA 2.2 remplacé par le POA.
module CORBA {
  interface ImplementationDef { … }; // non standardisé par l'OMG

  interface BOA {
    // Se mettre en attente des requêtes aux objets.
    void impl_is_ready (in ImplementationDef impl);
  };

  interface ORB {
    typedef sequence<string> arg_list;
    typedef string OAid;                        // Obtenir l'adaptateur d'objets.
    BOA BOA_init (inout arg_list argv, in OAid oa_identifier);
  };
};
```

Un serveur CORBA peut accueillir simultanément plusieurs implantations d'objets. En fait, la bibliothèque CORBA, implantant l'adaptateur d'objets et liée au serveur, gère un dictionnaire associant aux références d'objet des objets d'implantation (pour plus d'informations sur cette structure nous vous conseillons de « plonger » dans les sources d'un ORB :-). Lorsqu'une requête IIOP arrive sur le serveur, celle-ci est prise en charge par la bibliothèque. Celle-ci recherche alors l'implantation associée à la référence d'objet contenue dans la requête et délègue la requête au squelette dont hérite cette implantation. Le squelette, automatiquement généré et dépendant de ce bus (voir section 2.5), déballe les paramètres de la requête et invoque alors l'implantation de l'opération IDL. Ainsi la coopération de la bibliothèque et des squelettes permet de router automatiquement la requête IIOP vers la « bonne » méthode du « bon » objet d'implantation sans aucune intervention de la part des développeurs de l'application. Cependant, l'exécution des méthodes peut être problématique : utilisation de code non réentrant et non protection de variables globales. Ces problèmes sont à la charge des développeurs.

Après ce retour sur quelques concepts du bus CORBA, nous allons poursuivre la construction de notre application d'annuaires en étudiant la construction des applications serveurs et clientes.

3.6. Un simple serveur de répertoire

3.6.1. Le scénario général d'un serveur

Maintenant que notre objet répertoire est implanté, il est nécessaire de concevoir l'application serveur offrant la structure d'accueil à notre objet et l'environnement d'exécution des requêtes à l'objet. Un serveur CORBA suit toujours le même scénario quel que soit le langage de programmation utilisé, il doit :

1. initialiser le bus CORBA, c'est-à-dire créer l'objet ORB ;
2. initialiser l'adaptateur d'objets ;
3. créer une implantation de l'objet (ou des implantations) ;
4. enregistrer ces implantations auprès de l'adaptateur d'objets ;
5. diffuser la référence de ces objets aux applications clientes ;

6. se mettre en attente des requêtes aux objets.

Les trois exemples suivants illustrent cette scénario en C++, en Java et en CorbaScript. Comme nous utilisons ORBacus, l'adaptateur d'objets est alors le BOA et l'enregistrement des implantations est implicite, i.e. il suffit de créer les implantations. La diffusion aux applications clientes de la référence d'objet se fait simplement par affichage de l'IOR de l'objet sur l'écran du serveur. Une autre solution aurait pu être de stocker cette IOR dans un fichier. Nous verrons dans la section 3.8 une approche plus généraliste et orientée objet par l'utilisation du service standard de désignation des objets CORBA (Naming Service).

3.6.2. Le serveur en C++

Le source suivant illustre le scénario précédent en C++, les numéros en commentaire désignent les étapes du scénario précédent :

```
// Importation de l'implantation C++ de l'interface IDL annuaire::Repertoire.
#include <RepertoireImpl.h>

int main(int argc, char* argv[], char* env[])
{
  try {
    // Initialisation du bus CORBA pour un processus serveur.
    // Création des objets ORB et BOA.
    CORBA_ORB_var orb = CORBA_ORB_init (argc, argv);                    // (1)
    CORBA_BOA_var boa = orb -> BOA_init (argc, argv);                   // (2)

    // Création de l'objet Repertoire.
    annuaire_Repertoire_var repertoire = new RepertoireImpl ("LIFL");   // (3)
                                                                        // (4) implicite.
    // Obtenir sous forme textuelle l'IOR de l'objet.
    CORBA_String_var chaineIOR = orb->object_to_string (repertoire);
    cout << "L'adresse du répertoire est " << chaineIOR << endl;        // (5)

    // Mettre le serveur en attente des requêtes venant du bus CORBA.
    boa -> impl_is_ready (CORBA_ImplementationDef::_nil());             // (6)

  // en cas de problème lié à l'utilisation de CORBA.
  } catch(CORBA_SystemException&) {
    cerr << "ERREUR : CORBA::SystemException provoquée" << endl;
  }
  return 0;
}
```

Remarquez que pour l'appel de l'opération *CORBA::ORB_init* et *CORBA::ORB::BOA_init*, les arguments sont passés dans le format utilisé couramment en C++ à savoir *argc* et *argv* à la place d'une séquence de *string* comme cela est définie dans la spécification IDL du module *CORBA*. Cette entorse est due au fait qu'une projection IDL tente d'être au plus proche de l'utilisation courante du langage de programmation.

3.6.3. Le serveur en Java

Le code Java suivant illustre de la même manière le scénario de mise en œuvre d'un serveur CORBA :

```
public class Serveur1
{
  public static void main(String args[])
  {
    try {
      // Initialisation du bus CORBA pour un processus serveur.
      // Création des objets ORB et BOA.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (args, null);      // (1)
      org.omg.CORBA.BOA boa = orb.BOA_init (args, null);                // (2)

      // Création de l'objet Repertoire.
      RepertoireImpl repertoire = new RepertoireImpl("LIFL");           // (3)
                                                                  // (4) est encore implicite.
      // Obtenir sous forme textuelle l'IOR de l'objet.
      String chaineIOR = orb.object_to_string (repertoire);
      System.out.println ("L'adresse du répertoire est " + chaineIOR);  // (5)

      // Mettre le serveur en attente des requêtes venant du bus CORBA.
      boa.impl_is_ready(null);                                          // (6)
```

```
    // en cas de problème lié à l'utilisation de CORBA.
    } catch(org.omg.CORBA.SystemException ex) {
      System.err.println("Exception " + ex);
    }
  }
}
```

Notez que la remarque précédente sur le passage des paramètres d'initialisation s'applique encore ici. De plus, l'opération d'initialisation de l'ORB est contenue dans la classe *org.omg.CORBA.ORB* et non dans le package *org.omg.CORBA* tout simplement car un package Java ne peut pas contenir directement une méthode Java.

3.6.4. Le serveur en CorbaScript

Le script suivant illustre l'implantation du serveur avec CorbaScript. Les étapes du scénario présenté en 3.6.1 doivent être respectées. L'initialisation du bus CORBA et de l'adaptateur d'objets (1 et 2) est implicitement réalisée par l'interpréteur CorbaScript. Comme il n'y a pas de génération de squelette CorbaScript, il est nécessaire d'enregistrer explicitement une instance CorbaScript auprès de l'adaptateur d'objets (4) en indiquant quelle est son interface OMG-IDL (e.g. *CORBA.ORB.connect*). Cette opération ajoute dynamiquement l'attribut *_this* à l'instance CorbaScript. Cet attribut désigne la référence d'objet CORBA associée à l'implantation CorbaScript. L'expression *_this._ior* est une forme simplifiée pour obtenir la chaîne IOR associée à la référence d'objet CORBA (i.e. *CORBA.ORB.object_to_string*).

```
# Initialisation implicite du bus CORBA, de l'ORB et de l'adaptateur d'objets    (1) et (2).

# Importation de l'implantation de l'annuaire dont la classe RepertoireImpl.
import annuaireImpl

# Création de l'objet Repertoire.
repertoire = annuaireImpl.RepertoireImpl("LIFL")                                 # (3)

# L'instance repertoire est l'implantation CorbaScript d'un objet CORBA
# d'interface OMG-IDL annuaire::Repertoire.
CORBA.ORB.connect (repertoire, annuaire.Repertoire)                              # (4)

# Obtenir sous forme textuelle l'IOR de l'objet.
chaine = repertoire._this._ior                                                   # (5)
# Forme simplifiée de chaine = CORBA.ORB.object_to_string(repertoire._this)
println ("L'adresse du répertoire est ", chaineIOR)

# Mettre le serveur en attente des requêtes venant du bus CORBA.
CORBA.ORB.run ()                                                                 # (6)
```

3.7. Une application cliente du répertoire

3.7.1. Le scénario général d'un client

Pour pouvoir utiliser notre exemple en réparti, il ne reste plus qu'à réaliser les applications clientes des serveurs de répertoire. Le scénario de base d'une application cliente est le suivant :

1. initialiser le bus CORBA, c'est-à-dire créer l'objet ORB ;
2. obtenir les références des objets utilisés par l'application ;
3. appliquer des traitements sur les objets obtenus.

Nous avons déjà vu comment initialiser le bus CORBA dans la section 3.6 et un exemple de traitement dans la section 3.4. Il ne reste donc plus qu'à étudier comment obtenir les références d'objets.

3.7.2. La connexion aux objets

Le bus CORBA offre un environnement orienté objet et client/serveur : des applications clientes peuvent invoquer des objets gérés par des applications serveurs. Pour cela, l'application cliente doit posséder une référence sur l'objet serveur afin de lui appliquer des traitements. Dans le monde CORBA, une référence est une instance d'une souche IDL. Il faut donc que l'application cliente crée cette instance. Cette opération revient à connecter l'application cliente à l'objet serveur.

Cette opération de connexion est implantée dans la plupart des produits CORBA (Orbix, Visibroker, MICO) par la méthode *_bind* fournie par les souches IDL. Cependant, la méthode _bind n'est pas du tout définie par la norme CORBA, c'est donc un ajout propriétaire à chaque ORB. Les paramètres de cette opération sont de nature différente sur chaque produit impliquant la dépendance des applications vis-à-vis du produit utilisé.

```
// Exemple C++ d'utilisation de l'opération _bind.
annuaire_Repertoire_var un_repertoire = annuaire_Repertoire::_bind ( … dépendant de l'ORB … );
```

Pour cette raison, nous n'utilisons pas cette fonctionnalité de connexion qui a l'avantage d'être simple. Nous allons plutôt montrer le mécanisme standard offert par la norme CORBA. Ce mécanisme repose sur le fait qu'il est possible de convertir une référence d'objet en une chaîne de caractères représentant l'IOR de l'objet et vice-versa, i.e. les opérations *object_to_string* et *string_to_object* de l'interface *ORB*. Dans la section 3.6, les serveurs affichent l'IOR de leur objet à l'écran, il suffit donc qu'une application cliente récupère cette chaîne et la convertisse localement en une référence d'objet.

```
// Exemple C++ de conversion d'une chaîne IOR en une référence d'objet.
const char* une_chaine_contenant_une_IOR = …;
CORBA_Object_var la_reference_d_objet = orb->string_to_object (une_chaine_contenant_une_IOR);
```

Ainsi, l'exemple précédent montre comment une application cliente peut obtenir une référence d'objet de manière standard grâce à une chaîne IOR. Cependant, l'application cliente a besoin d'une référence d'objet du type IDL *annuaire::Repertoire* et non du type *CORBA::Object*. Pour cela, la norme CORBA fournit un opérateur de conversion de type d'objet, l'opération *_narrow* contenue dans les souches IDL.

```
// Exemple C++ d'utilisation de l'opérateur de conversion _narrow.
Annuaire_Repertoire_var un_repertoire = annuaire_Repertoire::_narrow(la_reference_d_objet);
```

Dans le monde CORBA, l'opérateur *_narrow* doit être utilisé chaque fois que l'on veut convertir une référence d'objet O vers une référence d'objet d'un type T. Cette opérateur vérifie alors que le type réel de l'objet O est bien le type T ou un type héritant de T. Si le type réel de O est conforme à T alors l'opérateur *_narrow* crée localement à l'appelant une référence d'objet de type T sinon elle retourne une référence à nil. L'implantation de cet opérateur peut utiliser le référentiel des interfaces pour vérifier cette conformité ou demander à l'objet O s'il est d'un type conforme à T, i.e. invocation de l'opération *_is_a* définie dans l'interface *Object* et héritée implicitement pour toute souche IDL. La plupart des bus CORBA mettent aussi en place un cache local de conformité des types afin d'éviter d'envoyer une requête IIOP pour chaque opération *_narrow* effectué par les applications.

Dans les deux exemples suivants, cette étape de connexion aux objets est numérotée 2.a et 2.b respectivement lors de la conversion d'une chaîne IOR en référence d'objet et lors de la conversion vers un type spécifique de référence.

3.7.3. Un client en C++

Ici, l'IOR de l'objet serveur est passé sur la ligne de commandes au lancement de l'application cliente. Ainsi celle-ci peut se connecter à n'importe quel objet CORBA implantant l'interface *annuaire::Repertoire* et invoquer les opérations sur le répertoire.

```
int main(int argc, char* argv[], char* env[])
{
  try {
    // Initialisation du bus CORBA pour un processus client.
    // Création de l'objet ORB.
    CORBA_ORB_var orb = CORBA_ORB_init (argc, argv);                          // (1)

    // Instanciation d'une souche C++ référençant l'objet Répertoire.
    CORBA_Object_var obj = orb -> string_to_object(argv[1]);                  // (2.a)
    annuaire_Repertoire_var repertoire = annuaire_Repertoire::_narrow(obj);   // (2.b)

    traitement (repertoire);                                                  // (3)
  // en cas de problème lié à l'utilisation de CORBA.
  } catch(CORBA_SystemException& ) {
    cerr << "ERREUR : CORBA::SystemException provoquée" << endl;
  }
  return 0;
}
```

3.7.4. Un client en Java

Le code suivant illustre la réalisation de l'application cliente avec Java :

```
import fr.lifl.annuaire.*;          // Simplification de la désignation des types IDL.

public class Client1
{
  public static void main(String args[])
  {
    try {
      // Initialisation du bus CORBA pour un processus client.
      // Création de l'objet ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (args, null);              // (1)

      // Instanciation d'une souche Java référençant l'objet Repertoire.
      org.omg.CORBA.Object objet = orb.string_to_object(args[0]);               // (2.a)
      Repertoire repertoire = RepertoireHelper.narrow (objet);                  // (2.b)

      traitement (repertoire);                                                  // (3)

    // en cas de problème lié à l'utilisation de CORBA.
    } catch(Exception ex) {
      System.err.println("Exception " + ex);
    }
  }
}
```

Dans la projection IDL/Java, l'opérateur *narrow* de l'interface *Repertoire* est stocké dans la classe Java *RepertoireHelper* et non dans l'interface Java *Repertoire* parce que tout simplement une interface Java ne peut pas contenir de code.

3.7.5. Un client en CorbaScript

CorbaScript a été conçu pour simplifier l'utilisation interactive des objets CORBA. Ainsi, un script n'a pas besoin d'initialiser le bus CORBA (1), i.e. l'interpréteur le fait implicitement. De plus, la connexion à un objet CORBA (2) est réalisée simplement en fournissant le nom de l'interface OMG-IDL et l'adresse IOR de l'objet. Notez que comme CorbaScript est un langage à typage dynamique, l'opération de conversion (*narrow*) est implicitement réalisée.

```
# Initialisation implicite du bus CORBA et de l'ORB.                          (1)

# Instanciation d'une souche CorbaScript référençant l'objet Repertoire.
repertoire = annuaire.Repertoire(sys.arg[1])                                # (2)

import Traitement
Traitement.traitement (repertoire)                                          # (3)
```

Quelques remarques au sujet de CorbaScript : *sys.arg* est un tableau contenant les arguments de la ligne de commande et l'instruction *import* permet de charger un module de code contenu dans un fichier script. Le module chargé est alors vu comme un objet, e.g. *Traitement.traitement*.

3.8. Les services de recherche d'objets

3.8.1. Leur rôle

L'inconvénient de l'application répartie que nous venons de réaliser est que les utilisateurs sont obligés de manipuler manuellement les IOR des répertoires. Pour masquer cette manipulation directe, le bus CORBA fournit des services standards de recherche d'objets offrant des fonctionnalités similaires à l'annuaire téléphonique :

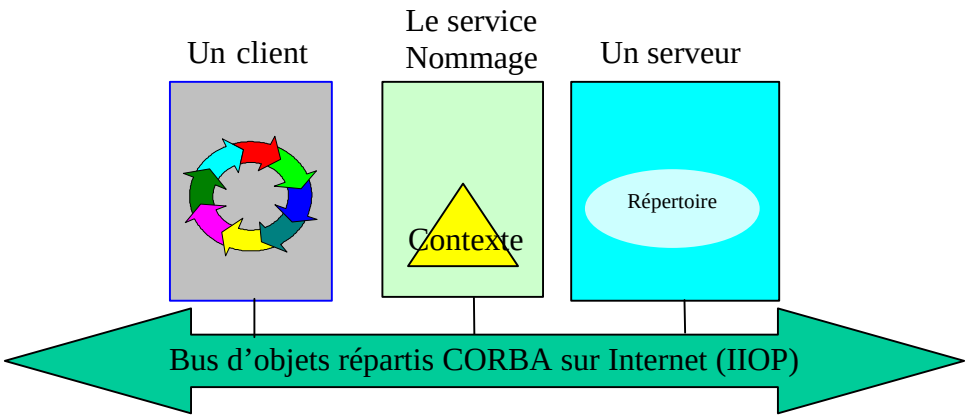
- Le **service Nommage** (Naming Service) permet la recherche des objets en fonction de noms symboliques leur ayant été associés. Par exemple, on recherchera l'objet de nom « répertoire ». C'est donc l'équivalent des « pages blanches » de l'annuaire téléphonique.

- Le **service Vendeur** (Trader Service) permet la recherche des objets en fonction de leurs caractéristiques. Par exemple, on recherchera l'objet d'interface *annuaire::Repertoire*, situé sur une des machines du troisième étage du bâtiment M3 de l'université de Lille I (bref le bâtiment du LIFL) et qui contient la liste des chercheurs. C'est donc l'équivalent des « pages jaunes ».

Le principe de mise en œuvre de ces services est le suivant :

1. L'**enregistrement** : les applications fournissant des objets enregistrent les références de ceux-ci auprès du service de recherche en leur associant soit un nom symbolique soit des caractéristiques.
2. L'**interrogation** : les applications clientes interrogent le service pour obtenir les références des objets qu'elles veulent invoquer. Le critère d'interrogation est soit un nom symbolique soit des caractéristiques.
3. La **recherche** : le service recherche dans l'ensemble des références d'objet enregistrées celles qui correspondent le mieux aux critères demandés par les applications clientes.

Ce document aborde seulement l'utilisation du service Nommage parce qu'il est très couramment utilisé dans des applications CORBA et qu'il est beaucoup plus simple d'emploi que le service Vendeur. Cependant, ce dernier doit être impérativement mis en œuvre lorsque les applications CORBA manipulent des critères de recherche complexes.



3.8.2. Le contrat IDL du service Nommage

Le service Nommage (*module CosNaming*) définit un espace de désignation symbolique des objets. Cet espace est structuré par un graphe de contextes de nommage (*interface NamingContext*). Chaque contexte maintient une liste d'associations entre des noms symboliques et des références d'objet. À l'intérieur d'un contexte, un nom (*struct NameComponent*) doit être unique et désigne soit une référence d'objet soit un autre contexte. La concaténation de plusieurs noms forme alors un chemin d'accès (*typedef Name*) à travers l'espace de désignation symbolique.

Un contexte fournit des opérations pour ajouter une association entre un nom et une référence (*bind*), mettre à jour une telle association (*rebind*), le connecter ou reconnecter à un contexte (*bind_context* et *rebind_context*), rechercher la référence désignée par un chemin (*resolve*), détruire une association (*unbind*), créer un nouveau contexte indépendant (*new_context*), créer un contexte et le connecter (*bind_new_context*), détruire définitivement le contexte (*destroy*), et finalement lister son contenu (*list*). Ces opérations déclenchent selon les différents cas de mauvaises utilisations les exceptions *NotFound*, *CannotProceed*, *InvalidName*, *AlreadyBound* et *NotEmpty*.

```
#pragma prefix "omg.org"                    // Standardisé par l'OMG
module CosNaming                            // Le service Nommage.
{
   typedef string Istring;
   struct NameComponent {                   // Un nom d'association dans un contexte.
     Istring id;
     Istring kind;
   };
   typedef sequence<NameComponent> Name;    // Un chemin d'accès = une suite de noms.

   // La nature d'une liaison : une référence ou un contexte.
   enum BindingType {nobject, ncontext};
```

```
   struct Binding {                          // Une liaison.
     Name binding_name;                      // Un chemin.
     BindingType binding_type;               // Le type de la liaison.
   };
   typedef sequence<Binding> BindingList;

   // Déclaration avancée pour utiliser ce type avant de le définir complètement.
   interface BindingIterator;

   interface NamingContext {                 // Un contexte de nommage.
     enum NotFoundReason {                   // La liste des raisons d'une exception NotFound.
       missing_node,                         // Un nom n'existe pas.
       not_context,                          // Un nom ne désigne pas un contexte.
       not_object                            // Un nom ne désigne pas un objet.
     };
     exception NotFound {                    // Signaler qu'un chemin n'a pas été trouvé.
       NotFoundReason why;                   // La raison.
       Name rest_of_name;                    // La partie du chemin fautif.
     };
     exception CannotProceed {               // Signaler une incapacité de traitement.
      NamingContext cxt;                     // Le contexte fautif.
      Name rest_of_name;                     // La partie du chemin fautif.
     };
     exception InvalidName { };              // Signaler qu'un chemin est invalide.
     exception AlreadyBound { };             // Signaler qu'une association est déjà utilisée.
     exception NotEmpty { };                 // Signaler qu'un contexte n'est pas vide.

     void bind(in Name n, in Object obj)                  // Associer un nom à une référence.
      raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
     void rebind(in Name n, in Object obj)                // Mettre à jour une association.
      raises(NotFound, CannotProceed, InvalidName);
     void bind_context(in Name n, in NamingContext nc)    // Connecter à un autre contexte
      raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
     void rebind_context(in Name n, in NamingContext nc)  // Reconnecter à un autre contexte.
      raises(NotFound, CannotProceed, InvalidName);
     Object resolve(in Name n)                            // Rechercher une association.
      raises(NotFound, CannotProceed, InvalidName);
     void unbind(in Name n)                               // Détruire une association.
      raises(NotFound, CannotProceed, InvalidName);
     NamingContext new_context();                         // Créer un contexte indépendant.
     NamingContext bind_new_context(in Name n)            // Créer et lier un contexte.
      raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
     void destroy( )                                      // Détruire le contexte.
        raises(NotEmpty);
     void list(in unsigned long how_many,                 // Lister le contenu du contexte.
               out BindingList bl, out BindingIterator bi);
   };

   interface BindingIterator {               // Itérateur des éléments d'une consultation.
     boolean next_one(out Binding b);              // Obtenir la liaison suivante.
     boolean next_n(in unsigned long how_many,     // Obtenir les n liaisons suivantes.
                    out BindingList bl);
     void destroy();                               // Détruire l'itérateur.
   };
};
```

L'opération *list* nécessite un peu d'explications. Elle permet d'inspecter le contenu d'un contexte, c'est-à-dire ses associations. Cette consultation retourne seulement les noms et la nature (*BindingType*) des références d'objet contenues, via une liste (*BindingList*) de liaisons (*Binding* nom + nature). Pour obtenir les références d'objet, il faut invoquer l'opération *resolve*. La consultation peut se faire par tranches successives en utilisant l'itérateur *bi BindingIterator* retourné en mode *out*. Cet itérateur permet d'obtenir séquentiellement les liaisons suivantes soit une par une (*next_one*) soit par tranche (*next_n*). À la fin de la consultation, l'itérateur doit être détruit explicitement par l'opération *destroy*.

```
// Exemple C++ de consultation d'un contexte.
void consultation_d_un_contexte (CosNaming_NamingContext* nc)
{
  CosNaming_BindingList_var bl;
  CosNaming_BindingIterator_var bi;
  nc->list (50, bl, bi);                     // Consultation des 50 premières liaisons.
  afficher (bl);                             // Afficher n'est pas détaillée ici.
  if (!CORBA_is_nil(bi)) {                   // Faut-il itérer ?
    CORBA_Boolean continuer = CORBA_TRUE;
    while (continuer) {
```

```
      continuer = bi->next_n(10,bl);        // Consultation des 10 liaisons suivantes.
      afficher (bl);
    }
    bi->destroy ();                         // Détruire explicitement l'itérateur.
  }
}
```

3.8.3. Obtenir le service Nommage

Le service Nommage permet d'obtenir les références d'objet. Mais cela ne fait que reporter le problème d'un cran car comment obtenir la référence du service Nommage. Pour cela, le bus CORBA fournit la notion d'objets « notoires » accessibles par le biais de l'opération *resolve_initial_references* de l'interface *ORB*. Le contexte racine de l'espace de désignation a pour nom *NameService* quel que soit le bus CORBA.

```
// En C++, obtenir la référence du service Nommage.
CORBA_Object_var objRef = orb->resolve_initial_references ("NameService");          // (1.a)
CosNaming_NamingContext_var nsRef = CosNaming_NamingContext::_narrow(objRef);       // (1.b)
```

Ici, l'opérateur *_narrow* permet de convertir la référence de l'objet « notoire » *NameService* en une référence du type IDL *CosNaming::NamingContext*.

```
// En Java, obtenir la référence du service Nommage.
org.omg.CORBA.Object objRef = orb.resolve_initial_references ("NameService");       // (1.a)
org.omg.CosNaming.NamingContext nsRef =
                        org.omg.CosNaming.NamingContextHelper.narrow(objRef);       // (1.b)
```

En CorbaScript, la conversion automatique des références d'objet simplifie le code nécessaire pour obtenir la référence sur le service Nommage :

```
# En CorbaScript, obtenir la référence du service Nommage.
nsRef = CORBA.ORB.resolve_initial_references ("NameService")                        #  (1)
```

À partir de ce moment, il est possible d'invoquer les opérations du service Nommage afin de gérer l'espace de désignation symbolique des objets. Les deux sections suivantes illustrent comment enregistrer une référence d'objet dans le service Nommage et comment retrouver une référence depuis ce service respectivement pour une application serveur et une cliente.

3.8.4. Enregistrer une référence d'objet

Une application serveur doit diffuser aux applications clientes les références de ses objets comme nous l'avons vu dans la section 3.6.1. Ici, l'utilisation du service Nommage remplace élégamment l'affichage à l'écran des références d'objet (remplace la primitive *object_to_string*). Pour cela, il suffit d'obtenir la référence du service Nommage (1), de créer un chemin valide (*CosNaming::Name*) (2) et d'associer à ce chemin la référence d'objet via l'opération *bind* ou *rebind* du service Nommage (3). La différence entre ces 2 opérations vient du fait que *rebind* crée ou remplace l'association si celle-ci existait déjà.

```
// En C++, enregistrer l'objet dans le service Nommage.
CosNaming_Name_var nsNom = new CosNaming_Name();                    // (2)
nsNom->length(1);
nsNom[0].id = (const char*) "REPERTOIRE";
nsNom[0].kind = (const char*) "";
nsRef->rebind (nsNom, repertoire);                                 // (3)
```

L'enregistrement d'une référence dans le service Nommage s'exprime aussi simplement en C++ et en Java. La seule différence notable est que la projection C++ d'une séquence IDL est un type (voir ci-dessus *CosNaming::Name*) tandis qu'en Java, il faut utiliser les tableaux standards (voir ci-dessous *NameComponent[]*).

```
// En Java, enregistrer l'objet dans le service Nommage.
org.omg.CosNaming.NameComponent[] nsNom = new org.omg.CosNaming.NameComponent [1]; // (2)
nsNom[0] = new org.omg.CosNaming.NameComponent("REPERTOIRE","") ;
nsRef.rebind (nsNom, repertoire);                                                  // (3)
```

Ici, CorbaScript montre encore sa simplicité car il permet de manipuler directement les types OMG-IDL :

```
# En CorbaScript, enregistrer l'objet dans le service Nommage.
nsNom = CosNaming.Name(CosNaming.NameComponent("REPERTOIRE",""))                   #  (2)
```

```
nsRef.rebind (nsNom, repertoire._this)                                    #  (3)
```

De plus, CorbaScript permet simplement la création/conversion de valeurs OMG-IDL :

```
nsRef.rebind ([ ["REPERTOIRE",""] ], repertoire._this)                    # (2 et 3)
# Le premier tableau est converti automatiquement en une séquence CosNaming.Name.
# Le second tableau est converti automatiquement en une structure CosNaming.NameComponent.
```

3.8.5. Rechercher une référence d'objet

Une application cliente doit obtenir les références des objets qu'elle désire invoquer. Soit elle utilise la primitive *string_to_object* sur une chaîne codifiant une IOR soit elle peut élégamment utiliser le service Nommage. Pour cela, il suffit d'obtenir la référence de ce service (1), créer un chemin valide (2) et rechercher la référence associée à ce chemin (3) via l'opération *resolve*.

```
// En C++, obtenir l'objet depuis le service Nommage.
CosNaming_Name_var nsNom = new CosNaming_Name();                           // (2)
nsNom->length(1);
nsNom[0].id = (const char*) "REPERTOIRE";
nsNom[0].kind = (const char*) "";
CORBA_Object_var objRef = nsRef->resolve (nsNom);                         // (3)

annuaire_Repertoire_var repertoire = annuaire_Repertoire::_narrow (objRef);    // (4)
```

La définition OMG-IDL de l'opération *resolve* retourne un *Object*. Aussi, il est toujours nécessaire de convertir la référence retournée vers le type nécessaire à l'application en utilisant l'opération de *narrow* (4).

```
// En Java, obtenir l'objet depuis le service Nommage.
org.omg.CosNaming.NameComponent[] nsNom = new org.omg.CosNaming.NameComponent [1];   // (2)
nsNom[0] = new org.omg.CosNaming.NameComponent("REPERTOIRE","");
org.omg.CORBA.Object objRef = nsRef.resolve (nsNom);                      // (3)

annuaire.Repertoire repertoire = annuaire.RepertoireHelper.narrow (objRef);          // (4)
```

En CorbaScript, cela devient trivial :

```
# En CorbaScript, obtenir l'objet depuis le service Nommage.
repertoire = nsRef.resolve ([ ["REPERTOIRE",""] ])                        # (2 et 3 et 4)
```

En réutilisant les quelques fragments de code précédents, nous pouvons ainsi modifier notre première version de l'application répartie annuaire afin d'éviter que les utilisateurs aient à manipuler directement les références d'objet (IOR). De plus, notez alors que l'application serveur est toujours la structure d'accueil de l'objet *Repertoire* mais elle devient aussi une application cliente du service Nommage. Ainsi dans le monde CORBA, un processus peut être à la fois une application cliente et serveur : la notion de client et de serveur se définit uniquement et relativement à un objet CORBA. Cet aspect sera encore illustré dans la section 3.10 où l'application cliente implante aussi un objet.

3.9. Une fabrique de répertoires

Dans l'état actuel, l'application développée n'est pas satisfaisante :-) Premièrement, chaque objet *Repertoire* est accueilli par un serveur : cela veut dire qu'un seul simple objet occupe complètement les ressources d'un processus. Il pourrait donc être intéressant, par économie mémoire, de regrouper plusieurs répertoires dans une même structure d'accueil. Deuxièmement, chaque fois que l'on désire avoir un nouveau répertoire il est nécessaire de lancer un processus pour l'accueillir. Nous allons voir dans cette section comment traiter ces deux problèmes en mettant en place une fabrique de répertoires. Ce nouveau serveur accueille à l'exécution plusieurs instances de répertoires et fournit une nouvelle interface OMG-IDL permettant de créer et de détruire les répertoires accueillis.

3.9.1. Le nouveau contrat IDL

Comme CORBA est un modèle client/serveur ne fournissant pas directement la création d'objets à distance, nous allons construire notre serveur « Fabrique de répertoires ». Ce serveur contient simultanément un objet *Fabrique* et des objets *Repertoire*. Le contrat IDL suivant est une extension du contrat présenté en section 3.1 : un module OMG-IDL peut être étendu simplement en « réouvrant » celui-ci. Nous y ajoutons la nouvelle

interface *Fabrique* permettant de créer un nouvel objet répertoire (i.e. *creerRepertoire*) ou de détruire un répertoire précédemment créé via cette fabrique (i.e. *detruireRepertoire*). Remarquez la manière simple d'exprimer en OMG-IDL le passage et le retour de références d'objet typées (*Repertoire*).

```
module annuaire {
  interface Fabrique {
    Repertoire creerRepertoire (in string libelle);
    void detruireRepertoire (in Repertoire repertoire);
  };
};
```

De manière plus générale, cette spécification IDL illustre le canevas de conception (ou « design pattern ») de gestion du cycle de vie des objets CORBA. La création d'un objet est réalisée par l'invocation d'une opération d'une fabrique d'objets (ou usine). Notez que cette opération de création doit prévoir assez de paramètres afin d'initialiser l'état du nouvel objet. La destruction d'un objet est spécifiée soit par une opération de sa fabrique soit directement par une opération de son interface comme, par exemple, l'opération *destroy* des interfaces *CosNaming::NamingContext* et *CosNaming::BindingIterator* (section 3.8.2). De plus, l'OMG a déjà spécifié un service générique de gestion du cycle de vie des objets : le service Cycle de Vie (module *CosLifeCycle*) définit l'interface *FactoryFinder* de recherche de fabriques, l'interface d'une fabrique générique *GenericFactory* et l'interface de cycle de vie d'un objet *LifeCycleObject*. Ce service spécifie ainsi le scénario de gestion de la vie des objets. Malheureusement, l'implantation de ce service n'est pas encore disponible sur tous les bus CORBA.

3.9.2. L'implantation de la fabrique

L'implantation de la fabrique de répertoires (classe *FabriqueImpl*) est vraiment très simple comme nous le voyons ci-après en C++, en Java et en CorbaScript. La méthode *creerRepertoire* crée une instance de la classe *RepertoireImpl* et retourne la référence de cet objet CORBA. La méthode *detruireRepertoire* déconnecte la référence de l'objet répertoire de son objet d'implantation (en fait, *disconnect* met à jour le dictionnaire géré par la bibliothèque CORBA).

L'implantation C++ de l'interface *annuaire::Fabrique* est la suivante :

```
// Importation de l'implantation de annuaire::Repertoire
#include <RepertoireImpl.h>

class FabriqueImpl : public virtual annuaire_Fabrique_skel
{
public:
  FabriqueImpl () : annuaire_Fabrique_skel() {}
  ~FabriqueImpl () {}

  virtual annuaire_Repertoire_ptr creerRepertoire(const char* libelle)
  {
    RepertoireImpl* repertoire = new RepertoireImpl(libelle);
    // En C++, la valeur retournée doit être une copie.
    return annuaire_Repertoire::_duplicate(repertoire);
  }

  void detruireRepertoire(annuaire_Repertoire_ptr repertoire)
  {
    une_reference_sur_l_objet_orb->disconnect(repertoire);
  }
};
```

L'implantation Java de l'interface *annuaire::Fabrique* est la suivante :

```
public class FabriqueImpl extends fr.lifl.annuaire._FabriqueImplBase
{
  public FabriqueImpl () {}

  public fr.lifl.annuaire.Repertoire creerRepertoire(String libelle)
  {
    return new RepertoireImpl(libelle);
  }

  public void detruireRepertoire(fr.lifl.annuaire.Repertoire repertoire)
  {
    org.omg.CORBA.ORB.init().disconnect(repertoire);
  }
```

```
}
```

L'implantation CorbaScript de l'interface *annuaire::Fabrique* est la suivante :

```
class FabriqueImpl
{
  proc __FabriqueImpl__(self)
  {
    CORBA.ORB.connect (self, annuaire.Fabrique)     # Cette instance est une Fabrique.
  }

  proc creerRepertoire(self, libelle)
  {
    r = RepertoireImpl(description)                 # Création de l'implantation.
    CORBA.ORB.connect (r, annuaire.Repertoire)      # Enregistrement auprès du bus.
    return r._this                                  # Retour de la référence d'objet.
  }

  proc detruireRepertoire(self, repertoire)
  {
    CORBA.ORB.disconnect (repertoire)               # Déconnexion de l'implantation.
  }
}
```

Le code du serveur « Fabrique de répertoires » n'est pas présenté ici car il n'apporte rien par rapport à celui de la section 3.6. Il suffit de créer un objet *FabriqueImpl* à la place d'un objet *RepertoireImpl*.

3.9.3. Une application cliente

L'application cliente illustre l'utilisation de la fabrique de répertoires : elle initialise le bus CORBA (1), obtient le service Nommage (2), obtient la fabrique (3), crée un répertoire (4) et finalement l'enregistre dans le service Nommage (5).

En C++, cela donne le code suivant :

```
// Importation de la souche du service Nommage générée par le compilateur IDL.
#include <CosNaming.h>

// Importation de la souche annuaire générée par le compilateur IDL.
#include <annuaire.h>

void main(int argc, char* argv[])
{
  CORBA_ORB_var orb = CORBA_ORB_init (argc,argv);                                  // (1)
  CORBA_Object_var objet = orb->resolve_initial_references ("NameService");       // (2.a)
  CosNaming_NamingContext_var nc = CosNaming_NamingContext::_narrow(objet);       // (2.b)

  CosNaming_Name_var nsNom = new CosNaming_Name();
  nsNom->length(1);
  nsNom[0].id = (const char*) "FABRIQUE";
  nsNom[0].kind = (const char*) "";

  objet = nc->resolve(nsNom);                                                     // (3.a)
  annuaire_Fabrique_var fabrique = annuaire_Fabrique::_narrow (objet);            // (3.b)

  annuaire_Repertoire_var repertoire = fabrique->creerRepertoire("LIFL");         // (4)

  nsNom[0].id = (const char*) "REPERTOIRE";
  nc->rebind (nsNom, repertoire);                                                 // (5)
}
```

Quelques remarques sur ce code C++ : la ré-affectation d'une variable de type *_var* libère automatiquement la ressource gérée précédemment (e.g. variables *objet* et *nsNom*) et la création d'un répertoire retourne directement une référence de type *Repertoire* ne nécessitant donc pas de la convertir (pas de « narrowing »).

En Java, cela donne le code suivant :

```
import org.omg.CORBA.* ;        // Importation du package CORBA.
import org.omg.CosNaming.* ;    // Importation du package CosNaming.
import fr.lifl.annuaire.* ;     // Importation du package annuaire.
```

```
public class Client3
{
  public static void main(String args[]) throws Exception
  {
    ORB orb = ORB.init (args,null);                                          // (1)
    Object objet = orb.resolve_initial_references ("NameService");           // (2.a)
    NamingContext nc = NamingContextHelper.narrow(objet);                    // (2.b)

    NameComponent[] nsNom = new NameComponent [1];
    nsNom[0] = new NameComponent("FABRIQUE","");

    objet = nc.resolve (nsNom);                                              // (3.a)
    Fabrique fabrique = FabriqueHelper.narrow (objet);                       // (3.b)

    Repertoire repertoire = fabrique.creerRepertoire("LIFL");                // (4)

    nsNom[0] = new NameComponent("REPERTOIRE","");
    nc.rebind (nsNom, repertoire);                                           // (5)
  }
}
```

En CorbaScript, cela donne le script suivant :

```
nc = CORBA.ORB.resolve_initial_references ("NameService")                    # (2)
fabrique = nc.resolve( [ ["FABRIQUE", "" ] ] )                               # (3)

repertoire = fabrique.creerRepertoire("LIFL")                                # (4)
nc.rebind( [ ["REPERTOIRE", "" ] ], repertoire)                              # (5)
```

Ce dernier script illustre parfaitement la simplicité d'utilisation des objets CORBA à travers CorbaScript. Ce langage permet aux utilisateurs d'exprimer interactivement l'essentiel, c'est-à-dire leurs traitements CORBA, sans se soucier des détails techniques de mise en œuvre. Ainsi, à travers cet exemple, on peut voir toute la puissance et la facilité offertes par CorbaScript pour administrer dynamiquement des objets CORBA. Pour cela, il suffit de connaître les spécifications OMG-IDL des objets à utiliser et la syntaxe de ce nouveau langage.

3.10. La notification d'une application cliente

3.10.1. Un nouvel objectif

Jusqu'à présent, les applications clientes doivent aller scruter périodiquement le serveur de répertoires pour être informé des changements : ajout, retrait ou modification d'une personne. Cette scrutation est réalisée selon un mode « poll » : le client consulte à intervalle régulier le serveur pour observer ses changements d'état. Cependant, si la période est trop courte et qu'il n'y a pas de changement alors cela génère des invocations de méthodes inutiles et donc du trafic réseau. Si la période est trop longue cela entraîne des incohérences entre l'état du serveur et la connaissance qu'en a le client. Ainsi, pour éliminer ces problèmes, on utilise souvent un mode « push » où le serveur informe en « temps réel » ses clients de tout changement, dit autrement le serveur notifie ses clients de ses changements. L'OMG a défini deux services standards pour prendre en charge cette fonction : le service Evénement et le service Notification. Cependant, nous ne mettons pas en œuvre ici ces services mais nous étudions plutôt comment décrire ce mode « push » au sein du contrat IDL d'une application CORBA.

3.10.2. Un nouveau contrat IDL

Nous allons une fois de plus compléter le contrat IDL de notre d'application de répertoires en ajoutant deux nouvelles interfaces OMG-IDL. Ces deux interfaces définissent un contrat de coopération entre le serveur et ses clients. Les clients doivent implanter l'interface *ObservateurRepertoire* définissant trois opérations afin d'être notifié de l'ajout d'une personne (e.g. *annoncerCreationPersonne*), du retrait d'une personne (e.g. *annoncerDestructionPersonne*) et de la modification d'une personne (e.g. *annoncerModificationPersonne*). Notez que ces opérations sont asynchrones (*oneway*), c'est-à-dire que le serveur n'attend pas que les clients aient traité les appels d'opérations. Par un choix délibéré, le serveur notifie ses clients uniquement en leur indiquant le type de changement et le nom de la personne sur laquel porte le changement. Nous aurions bien pu notifier plus d'informations comme la fiche complète d'une personne lors de sa création. Ici, le client devra invoquer *obtenirPersonne* pour avoir cette information.

La nouvelle interface *RepertoireObserve* est une extension de *Repertoire* permettant aux clients de s'enregistrer auprès du serveur pour lui indiquer qu'ils veulent être notifié des changements. Cet enregistrement se fait par l'intermédiaire des opérations *ajouterObservateur* et *retirerObservateur*. Cette interface est implantée par le serveur de répertoires.

```
module annuaire {
  interface ObservateurRepertoire {
    oneway void annoncerCreationPersonne (in Nom nom);
    oneway void annoncerDestructionPersonne (in Nom nom);
    oneway void annoncerModificationPersonne (in Nom nom);
  };

  interface RepertoireObserve : Repertoire {
    void ajouterObservateur (in ObservateurRepertoire observateur);
    void retirerObservateur (in ObservateurRepertoire observateur);
  };

  // Types utilitaires pour l'implantation en C++.
  typedef sequence<ObservateurRepertoire> DesObservateursRepertoire;
};
```

On peut constater ici que le langage OMG-IDL permet d'exprimer la partie fonctionnelle d'un contrat IDL mais ne fournit aucune construction pour décrire l'architecture logicielle du contrat, c'est-à-dire quelle application implante quelle interface. Cette information est exprimée dans de possibles commentaires :-)

Les paragraphes suivants illustrent seulement quelques fragments de l'implantation de ce nouveau contrat en attirant l'attention du lecteur sur quels points nouveaux et importants. Le code complet est disponible sur notre site WWW.

3.10.3. L'implantation du répertoire observé

Au niveau du contrat IDL, l'interface *RepertoireObserve* hérite de *Repertoire*. Par contre, cette hiérarchie n'est pas forcément respectée par l'implantation. Ainsi, l'implantation *RepertoireObserveImpl* de l'interface *RepertoireObserve* peut :

- **Tout implanter à partir de rien** : la classe *RepertoireObserveImpl* peut fournir une réalisation pour toutes les opérations de *Repertoire* et de *RepertoireObserve*. Cette solution n'est pas très satisfaisante :-) ;
- **Implanter par héritage** : la classe *RepertoireObserveImpl* peut hériter de la classe *RepertoireImpl*. Elle fournit une réalisation des opérations *ajouterObservateur* et *retirerObservateur* en gérant une structure de données stockant la liste des observateurs. Elle doit redéfinir la réalisation des opérations *ajouterPersonne*, *modifierPersonne* et *retirerPersonne*. Ces redéfinitions réutilisent le code hérité et notifie tous les observateurs. Cette solution est illustrée en C++ ;
- **Implanter par délégation** : la classe RepertoireObserveImpl peut implanter toutes les opérations. L'implantation de certaines opérations est déléguée à un objet RepertoireImpl. Cette solution doit être mise en œuvre lorsque le langage d'implantation ne fournit pas l'héritage multiple, comme c'est le cas pour Java.

Le fragment C++ suivant illustre l'implantation *RepertoireObserveImpl* réalisée par héritage de la classe *RepertoireImpl* et du squelette *annuaire_RepertoireObserve_skel*. Cette classe ajoute une nouvelle structure de données pour stocker la liste des observateurs (la séquence *annuaire_DesObservatoiresRepertoire*). L'opération *ajouterObservateur* ajoute un nouvel observateur à la fin de la liste. L'opération *retirerObservateur* recherche un observateur dans la liste et met à jour celle-ci. L'implantation des opérations *libelle*, *obtenirNom* et *listerNoms* est héritée de la classe *RepertoireImpl*. Les opérations *ajouterPersonne*, *retirerPersonne* et *modifierPersonne* sont redéfinies : ici seul le code *d'ajouterPersonne* est présenté. Leur implantation suit le scénario suivant : elle appelle le code hérité puis parcours la liste des observateurs pour les notifier du changement.

```
class RepertoireObserveImpl : public annuaire_RepertoireObserve_skel // héritage du squelette.
                            , public virtual RepertoireImpl       // héritage d'implantation.
{
  annuaire_DesObservateursRepertoire observateurs;  // La liste des observateurs à notifier.
public:
 RepertoireObserveImpl (const char* libelle)       // Le constructeur appelle les construc-
    : annuaire_RepertoireObserve_skel()            // teurs des classes héritées et
    , annuaire_Repertoire_skel()                   // initialise la liste.
    , RepertoireImpl(libelle)
    , observateurs()
  {}
```

```
  virtual void ajouterObservateur(annuaire_ObservateurRepertoire_ptr observateur)
  { // Agrandit la liste et ajoute le nouvel observateur à la fin de la liste.
    CORBA_ULong len = observateurs.length();
    observateurs.length(len+1);
    observateurs[len] = annuaire_ObservateurRepertoire::_duplicate(observateur);
  }

  virtual void retirerObservateur(annuaire_ObservateurRepertoire_ptr observateur)
  { // Recherche l'observateur dans la liste et tasse la liste.
    CORBA_ULong len = observateurs.length();
    for (CORBA_ULong i=0; i<len; i++) {     // Test d'équivalence entre 2 références d'objet.
      if (observateurs[i]->_is_equivalent(observateur)) {
        for(; i<len-1; i++)
          observateurs[i] = annuaire_ObservateurRepertoire::_duplicate(observateurs[i+1]);
        observateurs[len-1] = 0;
        observateurs.length(len-1);
        return;
      }
    }
  }

  virtual void ajouterPersonne(const annuaire_Personne& personne)
  {
    RepertoireImpl::ajouterPersonne (personne);           // Réutilise ancien code
                                                          // puis notifie les observateurs.
    annuaire_DesObservateursRepertoire observateursEnPanne;
    observateursEnPanne.length(0);

    for (CORBA_ULong i=0; i<observateurs.length(); i++) {
      annuaire_ObservateurRepertoire_ptr o = observateurs[i];
      try {
        o->annoncerCreationPersonne(personne.nom);        // Notifie un observateur.
      } catch (CORBA_COMM_FAILURE &) {                    // Si n'existe plus alors
        CORBA_ULong l = observateursEnPanne.length();     // le conserver dans une liste
        observateursEnPanne.length(l+1);                  // à retirer plus tard.
        observateursEnPanne[l] = annuaire_ObservateurRepertoire::_duplicate(o);
      }
    }
    for (CORBA_ULong j=0; j<observateursEnPanne.length(); j++) {
      retirerObservateur (observateursEnPanne[j]);
    }

  virtual void retirerPersonne(const char* nom) { … };
  virtual void modifierPersonne(const char* nom, const annuaire_Personne& personne) { … };
  // Les implantations de libelle, obtenirNom et listerNom sont héritées.
};
```

Lors du parcours de la liste des observateurs, il faut faire attention que les observateurs soient toujours disponibles, qu'ils n'aient pas été arrêtés brutalement, que le réseau soit toujours disponible, etc. Pour cela, il est nécessaire d'intercepter les exceptions système levées par le bus CORBA, ici *CORBA::COMM_FAILURE* indique un problème de communication réseau. Ainsi, même si CORBA cache beaucoup d'aspects de la répartition et de la communication avec les objets, cela n'empêche pas que « **programmer en contexte réparti impose de prendre en compte les problèmes de pannes** ».

L'implantation Java de la classe *RepertoireObserveImpl* est légèrement différente car le langage Java n'offre pas l'héritage multiple. Ainsi, il est impossible d'hériter à la fois de la classe squelette et de la classe *RepertoireImpl*. La solution est donc d'hériter du squelette et de déléguer les opérations vers l'objet *le_repertoire*. Comme en C++, il faut gérer la liste des observateurs et assurer la notification de ceux-ci.

```
public class RepertoireObserveImpl extends fr.lifl.annuaire._RepertoireObserveImplBase
{
  // Utilisation de la délégation car pas d'héritage multiple.
  protected RepertoireImpl le_repertoire;
  protected java.util.Vector observateurs;

  public RepertoireObserveImpl (String libelle)
  {
    this.le_repertoire = new RepertoireImpl (libelle);   // Création du répertoire encapsulé.
    this.observateurs = new java.util.Vector ();
  }

  public void ajouterObservateur(fr.lifl.annuaire.ObservateurRepertoire observateur)
```

```
  {
    this.observateurs.addElement (observateur);
  }

  public void retirerObservateur(fr.lifl.annuaire.ObservateurRepertoire observateur)
  {
    this.observateurs.removeElement (observateur);
  }

  public String libelle()
  {
    return le_repertoire.libelle();                    // Implantation par délégation.
  }

  // Les autres opérations sont implantées par délégation vers l'objet le_repertoire.
  // Il faut aussi mettre en place la mécanique de notification comme en C++.
}
```

Nous ne détaillons pas ici le code des applications serveurs. Ces applications doivent instancier un objet de la classe *Fabrique2Impl*. Cette classe est implantée de la même manière que la classe *FabriqueImpl* sauf qu'elle crée une instance de *RepertoireObserveImpl* à la place d'une instance de *RepertoireImpl*. Pour plus de détails, voir le code fourni.

3.10.4. Une application cliente notifiée

Le côté client du contrat de notification est facilement illustré par le script CorbaScript suivant :

```
class ObservateurRepertoireImpl      # L'implantation d'annuaire::ObservateurRepertoire
{
  proc __ObservateurRepertoireImpl__(self)
  {
    CORBA.ORB.connect(self,annuaire.ObservateurRepertoire)
  }
  proc annoncerCreationPersonne(self,nom)     { … }  # Dépend de ce que l'on veut faire :-)
  proc annoncerDestructionPersonne(self,nom)  { … }
  proc annoncerModificationPersonne(self,nom) { … }
}

ns = CORBA.ORB.resolve_initial_references ("NameService")  # Obtenir le service de nommage.
repertoire = ns.resolve( [ [ "UnRepertoireObservable", "" ] ] ) # Obtenir le répertoire.

# Vérifier dynamiquement que c'est bien un répertoire observable.
if ( ! repertoire._is_a (annuaire.RepertoireObserve) ) {
  println ("Ce répertoire n'est pas observable.")
  return
}

observateur = ObservateurRepertoireImpl()           # Créer un observateur.
repertoire.ajouterObservateur (observateur._this)   # L'enregistrer auprès du répertoire.
...
repertoire.retirerObservateur (observateur._this)   # Signaler la fin de la notification.
```

Nous ne donnons pas plus d'explications sur ce script car le lecteur ne devrait pas avoir de difficultés à le comprendre. De moins, nous l'espérons ! En C++ ou Java, les traitements sont les mêmes sauf qu'ils sont « pollués » de nombreux détails liés à l'utilisation de la projection IDL vers ces langages.

3.11. Résumé

En guise de résumé, cette section liste les différents aspects de la construction d'une application répartie que nous venons de voir et apporte quelques commentaires sur le choix d'un langage.

3.11.1. Construire une application répartie avec CORBA

Au fil de la présentation, nous avons abordé de nombreux aspects de la construction d'une application répartie avec CORBA :

- La définition, l'extension et la réutilisation par héritage du contrat IDL ;
- L'implantation d'objets CORBA via les squelettes IDL ;

- La réutilisation d'implantation par héritage ou délégation ;
- L'utilisation d'objets CORBA via les souches IDL ;
- Quelques interfaces IDL du bus CORBA ;
- L'implantation d'applications serveurs CORBA ;
- L'implantation d'applications clientes CORBA ;
- Les services de recherche et plus particulièrement le service Nommage ;
- Le cycle de vie des objets CORBA et le « design pattern » Fabrique ;
- La notification des applications clientes et le « design pattern » Observé – Observateur ;

Même si l'exemple traité semble très simple, il peut facilement être adapté à d'autres contextes applicatifs comme, par exemple, toute application donnant accès à une base de données. Pour cela, il suffit de changer les concepts décrits dans le contrat IDL. Par exemple, en remplaçant le type *Personne* par *Ouvrage*, on obtient alors une application de gestion de bibliothèques.

3.11.2. Choisir un langage d'implantation

La construction de notre application a été illustrée à travers trois langages de programmation : C++, Java et CorbaScript. Les fragments de code nous ont montré que globalement la construction dans ces trois langages est très similaire et que la différence provient principalement du niveau de difficulté/simplicité d'utilisation des règles de projection. En C++, le développeur doit faire attention à la mémoire (les variables *_var* et les retours par copie). En Java, il n'y a pas de difficultés majeures. En CorbaScript, les scripts expriment directement l'essentiel et l'interpréteur prend en charge les détails de mise en œuvre (e.g. les opérateurs de conversion implicites). Au vu des exemples, nous pouvons facilement classer le niveau de difficulté d'utilisation de ces langages avec CORBA : C++ > Java > CorbaScript (le > pour « plus difficile à utiliser que »).

Cependant, cela ne veut pas dire que l'on ne doit utiliser qu'un seul langage (CorbaScript !). Dans une application complexe, il sera plus judicieux d'utiliser chacun de ces langages à l'endroit le mieux approprié : le langage C++ est plus adapté pour construire des serveurs fortement sollicités, le langage Java est plus adapté pour construire les applications clientes avec les interfaces graphiques AWT et les applets dans le WWW, le langage CorbaScript est plus adapté pour les tâches de mise au point et d'administration d'une application répartie.

4. Les mécanismes dynamiques de CORBA

4.1. Du statique au dynamique

Le bus CORBA permet de réaliser des applications composées d'objets répartis. Pour cela, le pré-compilateur OMG-IDL génère automatiquement les souches de communication avec les objets. Ces souches utilisent alors le bus pour réaliser la coopération des objets des applications. Cette approche statique est bien adaptée pour la conception et l'exécution d'applications dont les spécifications OMG-IDL sont stabilisées comme nous l'avons vu dans la section précédente. Néanmoins, dans la plupart des applications complexes, certaines spécifications évoluent au cours du temps par l'ajout de nouvelles opérations et/ou de nouveaux types d'objets ou par la modification des spécifications existantes. Dans ces contextes, l'approche statique, par pré-génération automatique de souches, ne convient plus. Elle crée un lien statique (à la compilation) entre les applications clientes et les interfaces IDL des objets utilisés. Ainsi, lorsque les interfaces évoluent, il faut modifier et recompiler toutes les applications clientes et serveurs.

D'un autre côté, CORBA offre un ensemble de mécanismes pour exploiter et implanter dynamiquement des objets répartis : le référentiel des interfaces (IFR), l'interface d'invocations dynamiques (DII) et l'interface de squelettes dynamiques (DSI). Ces mécanismes dynamiques permettent de construire des applications qui s'adaptent automatiquement aux changements/évolutions des spécifications IDL. Ces mécanismes sont mis en œuvre dans notre langage CorbaScript pour permettre aux scripts d'accéder et d'implanter n'importe quel objet CORBA.

4.2. Le référentiel des interfaces

4.2.1. Le rôle

Le bus CORBA est un environnement auto-descriptif car il permet de découvrir et d'exploiter dynamiquement, c'est-à-dire durant l'exécution, les interfaces des objets CORBA. Ces définitions IDL sont entreposées dans une base de données appelée aussi le référentiel des interfaces (ou Interface Repository - IFR). Ce référentiel conserve une version compilée de tous les contrats IDL décrivant tous les objets disponibles sur le bus CORBA. Les informations contenues dans ce référentiel sont désignées sous le vocable de métadonnées.

Le référentiel des interfaces est structuré par un ensemble d'objets. Ces objets sont décrits par des interfaces IDL et sont donc accessibles à travers le bus CORBA. Bien entendu, les interfaces décrivant le référentiel sont elles-mêmes stockées dans le référentiel sous la forme d'objets CORBA. La norme CORBA 2.0 définit les interfaces IDL de consultation, de navigation et de modification des objets de ce référentiel. De plus, la norme précise l'organisation et les relations entre les métadonnées formant le référentiel.

Comme pour les autres composantes de CORBA, la norme laisse une totale liberté sur la façon d'implanter le référentiel. Les fournisseurs de bus CORBA peuvent utiliser une base de données (relationnelle ou objet) ou un système de fichiers pour gérer la persistance des métadonnées.

4.2.2. Quelques contextes d'utilisations

Le référentiel des interfaces fournit donc à l'exécution des métadonnées décrivant les interfaces des objets CORBA. Ces métadonnées peuvent être exploitées dans de nombreux contextes d'utilisation, chaque fois qu'une application CORBA a besoin de découvrir dynamiquement l'environnement dans lequel elle s'exécute. La liste suivante énumère quelques exemples d'utilisation du référentiel des interfaces :

- **L'installation et la distribution des interfaces IDL.** Les fournisseurs d'objets CORBA doivent diffuser à leurs utilisateurs les contrats IDL décrivant leurs objets. À partir de ces interfaces, les utilisateurs pourront générer les souches pour l'accès à ces objets. Le référentiel des interfaces est alors le lieu idéal pour stocker de manière permanente les contrats IDL et pour en assurer la diffusion.
- **Le contrôle des graphes d'héritage.** Lorsqu'un précompilateur IDL analyse un contrat IDL, il doit vérifier qu'il n'existe pas de cycle dans les graphes d'héritage entre les interfaces IDL. Pour effectuer ce contrôle, il peut interroger le référentiel des interfaces.
- **Des outils pour le développement.** Durant le développement des applications ou des objets, il est souvent nécessaire de consulter les interfaces IDL. Les métadonnées contenues dans le référentiel peuvent être alors exploitées par toute sorte d'outils pour les développeurs tels que des générateurs de code « à la volée », des ateliers de génie logiciel ou des navigateurs d'interfaces IDL similaires au browser de classes de l'environnement SmallTalk.
- **La vérification de la signature des opérations.** Lorsqu'une requête est émise vers un objet, le bus CORBA peut consulter le référentiel des interfaces pour vérifier la conformité des types des paramètres. Cette vérification peut être réalisée du côté des applications clientes par les interfaces d'invocations statiques et dynamiques. De même, les interfaces de squelettes statiques et dynamiques peuvent contrôler les requêtes à leur réception par les serveurs.
- **La fédération de bus CORBA.** Lorsque plusieurs implantations du bus CORBA doivent dialoguer, ils utilisent soit un protocole commun comme IIOP soit des passerelles d'interconnexion. Ces passerelles ont besoin de connaître les définitions IDL des objets afin d'assurer les conversions des requêtes entre les différents protocoles des bus. Pour cela, elles peuvent dynamiquement consulter le référentiel des interfaces pour obtenir les métadonnées nécessaires à leur fonctionnement.
- **Des objets auto-descriptifs.** Chaque objet CORBA peut être interrogé à l'exécution pour découvrir son interface IDL. L'opération *_get_interface* de l'interface *CORBA::Object* permet d'obtenir la référence de l'objet du référentiel décrivant l'interface de l'objet. Ainsi, durant l'exécution, toute application peut aller consulter ces informations pour découvrir quelles sont les opérations et le type de leurs paramètres. L'application peut alors construire des requêtes pour les objets *via* l'interface d'invocations dynamiques (DII).

4.2.3. Le graphe des objets du référentiel

Le référentiel contient les métadonnées décrivant les interfaces IDL de tous les objets CORBA. Ces métadonnées sont stockées sous la forme d'un graphe d'objets CORBA accessibles à travers des interfaces IDL.

Chacun des types d'objets de ce graphe représente un élément sémantique du langage IDL. Ce graphe est structuré par les neuf principales interfaces suivantes :

Titre:

Auteur:
idraw
Aperçu:
Cette image EPS n'a pas été enregistrée
avec un aperçu intégré.
Commentaires:
Cette image EPS peut être imprimée sur une
imprimante PostScript mais pas sur
un autre type d'imprimante.

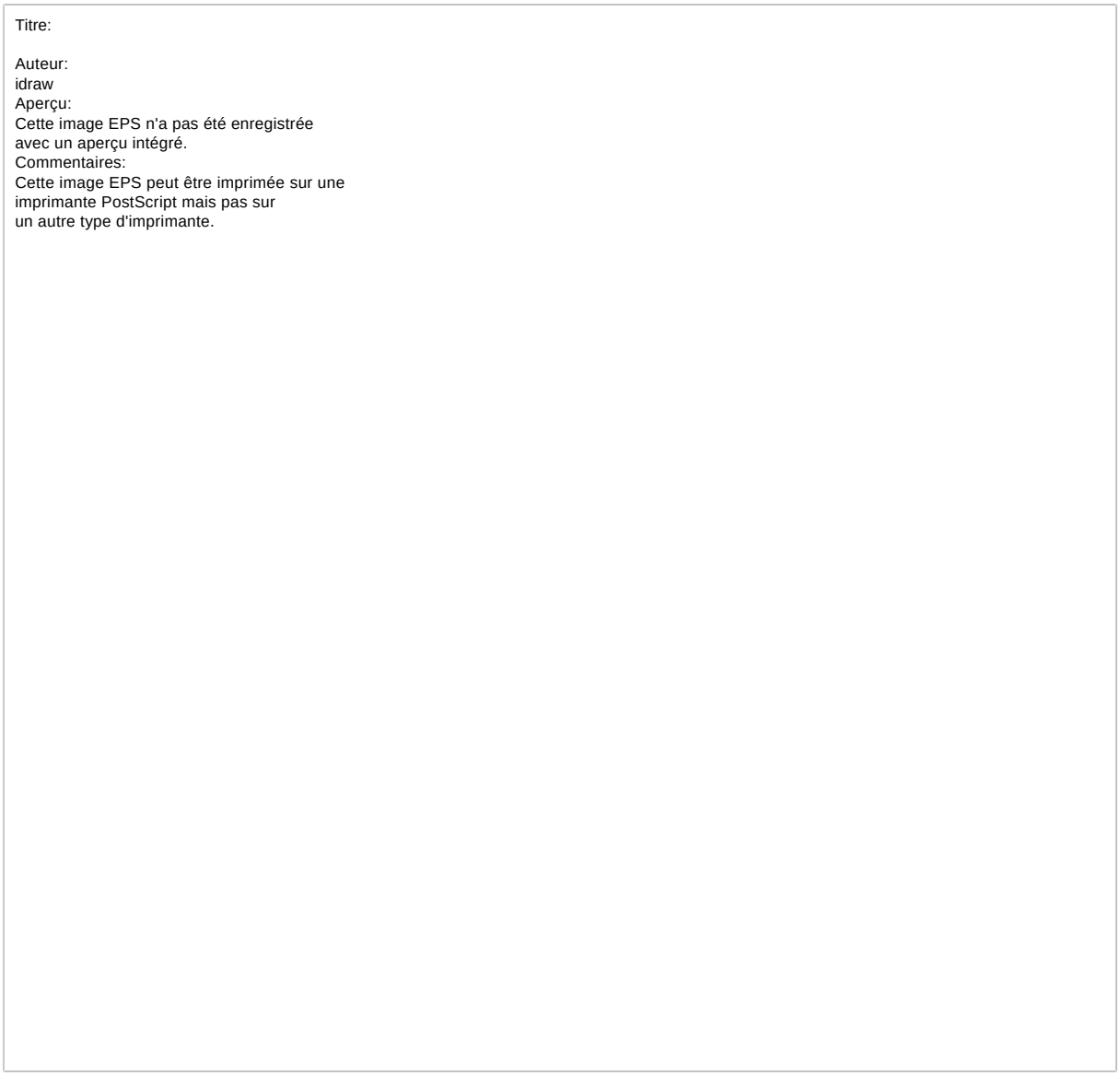
- ***CORBA::Repository*** est l'interface décrivant l'objet racine du référentiel des interfaces. À partir de cet objet, il est possible de découvrir les autres objets contenus dans le référentiel, c'est-à-dire les objets pour les modules, les interfaces, les exceptions, les définitions de type et les constantes IDL.
- ***CORBA::ModuleDef*** est l'interface décrivant les objets encapsulant les modules IDL. Elle fournit des opérations pour découvrir, ajouter et supprimer des objets encapsulant les définitions contenues dans un module.
- ***CORBA::InterfaceDef*** est l'interface décrivant les objets encapsulant les interfaces IDL. Elle permet de découvrir les opérations et les attributs fournis par une interface d'objets. De plus, elle permet de consulter et de vérifier les relations de sous-typage entre les interfaces.
- ***CORBA::AttributeDef*** est l'interface décrivant les objets encapsulant les attributs IDL. Elle permet de connaître le type et le mode d'accès associés à un attribut, c'est-à-dire consultation ou consultation/modification.
- ***CORBA::OperationDef*** est l'interface décrivant les objets encapsulant les opérations IDL. Elle permet d'obtenir la liste des paramètres et des exceptions de chaque opération. Cette information permet alors de construire des requêtes grâce à l'interface d'invocations dynamiques (DII).
- ***CORBA::ParameterDef*** est l'interface décrivant les objets encapsulant les paramètres des opérations IDL. Elle permet de consulter le type, le nom formel et le mode de passage de chaque paramètre de chaque opération IDL.
- ***CORBA::ExceptionDef*** est l'interface décrivant les objets encapsulant les exceptions IDL. Elle permet d'obtenir la liste des champs de chaque exception et fournit le nom et le type de chacun des champs.
- ***CORBA::TypedefDef*** est l'interface décrivant les objets encapsulant les types de données IDL. Elle fournit les informations associées à chaque définition de type IDL comme, par exemple, les valeurs symboliques des énumérations, les champs des structures, les dimensions des tableaux et le type des séquences.
- ***CORBA::ConstantDef*** est l'interface décrivant les objets encapsulant les constantes IDL. Elle permet de découvrir le type et la valeur de chaque constante IDL.

Le parcours du graphe de ces objets permet d'extraire les informations nécessaires, par exemple, à un navigateur d'interfaces IDL ou bien à un programme invoquant dynamiquement des objets via l'interface d'invocations dynamiques.

4.2.4. La hiérarchie des interfaces IDL

Comme les objets du référentiel ont certaines propriétés et certaines opérations communes, l'OMG définit une hiérarchie de leurs interfaces contenue dans le module *CORBA*. Pour cela, l'OMG utilise fortement le

mécanisme d'héritage multiple du langage IDL permettant de factoriser les éléments communs dans des interfaces abstraites. Ces interfaces abstraites sont ensuite dérivées pour obtenir les neuf principales interfaces que nous avons présentées dans le paragraphe précédent.

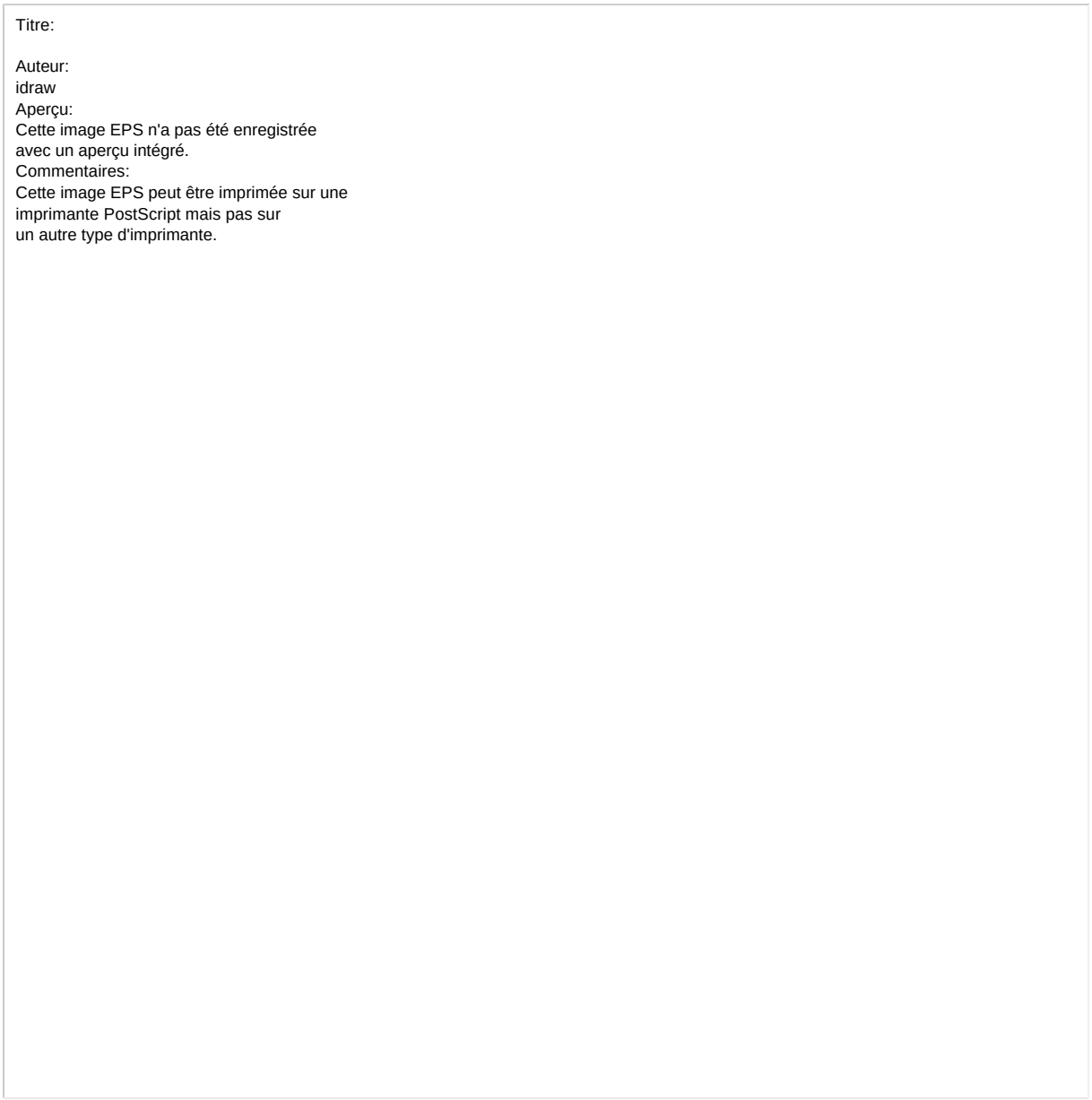


Toutes les interfaces du référentiel (l'IFR) héritent transitivement de l'interface abstraite *CORBA::IRObject*. Cette interface de base fournit l'attribut *def_kind* indiquant le type de définition contenu dans un objet de l'IFR et l'opération *destroy* pour détruire celui-ci. Ensuite, nous trouvons trois autres interfaces abstraites directement sous-types de *CORBA::IRObject* : *CORBA::Contained*, *CORBA::Container* et *CORBA::IDLType*.

L'interface *CORBA::IDLType* est l'abstraction de base des objets de l'IFR encapsulant un type OMG-IDL (l'attribut *type*). Cette interface est dérivée pour représenter les interfaces d'objets (*CORBA::InterfaceDef*), les types nommés (*CORBA::TypedefDef*) et les autres types de données (primitifs, chaînes, séquences et tableaux). L'interface des types nommés est sous-typée pour donner les interfaces pour les structures, les unions, les énumérations et les définitions de types.

L'interface *CORBA::Container* est l'abstraction de base des objets de l'IFR définissant des espaces de nommage IDL, c'est-à-dire ce sont des objets qui contiennent d'autres objets de type *CORBA::Contained*. Cette interface permet de rechercher, de découvrir et de créer les objets contenus dans les espaces de nommage via les opérations *lookup*, *contents* et *create_**. Les interfaces *CORBA::Repository*, *CORBA::ModuleDef*, *CORBA::OperationDef* et *CORBA::InterfaceDef* sont des sous-types de cette interface. Ils héritent donc des opérations de navigation et de création d'objets contenus. Ces interfaces ajoutent les attributs et les opérations

qui leur sont spécifiques comme, par exemple, découvrir la signature d'une opération ou bien rechercher globalement une définition dans le référentiel.



L'interface *CORBA::Contained* définit le comportement abstrait des objets contenus dans des objets *CORBA::Container*. L'opération *describe* retourne pour chacun des objets contenus une structure décrivant les métadonnées associées à l'objet. Il est aussi possible d'obtenir l'objet conteneur par *defined_in*, le nom local à ce conteneur via *name* ou le nom global au référentiel via *absolute_name*. Les interfaces *CORBA::AttributeDef*, *CORBA::ExceptionDef*, *CORBA::ConstantDef*, *CORBA::ParameterDef* et *CORBA::TypedefDef* héritent de cette interface abstraite. De plus, les interfaces conteneurs *CORBA::ModuleDef*, *CORBA::OperationDef* et *CORBA::InterfaceDef* héritent aussi de cette interface car elles définissent également des objets pouvant être contenus.

L'interface *CORBA::InterfaceDef* est la plus complexe de l'IFR (Ce qui est normal car elle assure la fonction principale du référentiel, à savoir fournir les métadonnées décrivant les interfaces IDL. Elle hérite des trois interfaces abstraites - *CORBA::IDLType*, *CORBA::Container* et *CORBA::Contained* - car une interface IDL est à la fois un type IDL, un espace de nommage contenant des déclarations d'attributs et d'opérations ainsi qu'une définition pouvant être contenue dans un module. De plus, elle fournit des opérations pour obtenir l'ensemble des métadonnées décrivant une interface (*describe_interface*) et des opérations pour créer des attributs et des opérations IDL.

Les spécifications du référentiel des interfaces sont complexes comme nous venons de le voir. Néanmoins, nous avons laissé de côté de nombreux détails techniques comme les structures de données utilisées dans le passage de paramètres ou le retour d'opérations. Il faut savoir que l'API du référentiel des interfaces constitue la partie la plus volumineuse des spécifications IDL des composantes techniques du bus. Cependant, il est très rare d'utiliser dans une application l'ensemble de cette API ; on se limite souvent aux opérations de recherche et à l'utilisation de l'opération *describe_interface*.

4.2.5. La fédération de référentiels

Une configuration « classique » d'un environnement CORBA est composée d'un bus réparti sur plusieurs machines qui se partagent un unique référentiel des interfaces. Cependant, il peut être intéressant de dupliquer le référentiel des interfaces afin d'éviter la centralisation des accès sur un seul serveur et donc de répartir la charge sur plusieurs serveurs. De plus, l'utilisation de plusieurs référentiels permet de définir différents domaines de gestion des métadonnées : nous pouvons avoir un référentiel pour les applications en phase d'exploitation, un référentiel pour les développements et un référentiel pour les tests d'applications.

Dans le cas où nous avons plusieurs référentiels, il est nécessaire de rendre cohérent leur contenu afin de disposer d'une vision unique et globale de leurs métadonnées. Ceci permet alors à une application distribuée, utilisant ces différents référentiels, de voir les mêmes métadonnées quelque soit le référentiel accédé. La norme CORBA 2.0 introduit deux formes de désignation des objets du référentiel pour assurer cette vision cohérente du contenu des différents référentiels.

Le premier mécanisme se fonde sur les espaces de nommage du langage IDL. Les définitions de modules, d'interfaces, d'opérations, d'attributs, d'exceptions et de types de données sont stockées dans des objets du référentiel désignés par une suite d'identificateurs séparés par des '*::*', aussi appelée nom étendu. Les identificateurs composant ce nom étendu sont les mêmes que ceux utilisés dans la définition des contrats IDL. Ainsi, l'objet du référentiel désigné par le nom étendu '*MonModule::MonInterface::MonOperation*' contient les métadonnées décrivant l'opération *MonOperation* du contrat IDL suivant :

```
module MonModule {
    interface MonInterface {
      void MonOperation ();
    };
  };
```

L'opération *lookup* de l'interface *CORBA::Container* permet de retrouver les objets du référentiel en les désignant par leur nom étendu. L'attribut *absolute_name* de l'interface *CORBA::Contained* retourne alors le nom étendu des objets contenus. Si ce mécanisme est couramment utilisé, il ne permet pas néanmoins de distinguer des versions différentes d'une définition IDL. Ainsi, CORBA 2.0 propose un second mécanisme reposant sur des identifiants de référentiel globaux et uniques appelés aussi *CORBA::RepositoryID*.

Ces identifiants de référentiel sont associés aux définitions contenues dans les contrats IDL grâce aux pragmas du langage OMG-IDL. Ils servent ensuite à désigner de manière unique les objets décrivant ces définitions. L'OMG définit trois formats de construction de ces identifiants :

- **Les identifiants OMG.** Ce format d'identifiants est celui de base utilisé par l'OMG et les pré-compilateurs IDL. Il construit un identifiant à partir du préfixe '*IDL:*' suivi d'une liste d'identificateurs séparés par '*/*' et terminé par un numéro de version. L'opération *MonOperation* présenté ci-dessus a alors comme identifiant de référentiel la chaîne *IDL:MonModule/MonInterface/MonOperation:1.0*.
- **Les identifiants DCE.** Ce format repose sur des UUID (Universal Unique IDentifier) de l'environnement distribué DCE. Cet environnement fournit un générateur d'UUID qui calcule une valeur globalement unique. Cette valeur est la composition de la date, de l'heure, de l'ID de la carte réseau et d'un compteur haute fréquence qui sont fournis par la machine où est généré l'UUID. Un identifiant de référentiel au format DCE est alors composé de trois champs : le préfixe '*DCE:*', un UUID représenté en hexadécimal et un numéro de version comme dans l'exemple suivant *DCE:700dc518-0110-11ce-ac8f-0800090b5d3e:1*.
- **Les identifiants locaux.** L'OMG n'a pas fixé de convention particulière pour ce dernier format. Un identifiant local doit commencer par le préfixe '*LOCAL:*' suivi d'une chaîne de caractères.

L'attribut *id* de l'interface *CORBA::Contained* permet de consulter l'identifiant de référentiel associé à un objet contenu. L'opération *lookup_id* de l'interface *CORBA::Repository* permet alors de rechercher les objets dans le référentiel en utilisant ces identifiants.

Ainsi, les deux mécanismes de désignation des objets du référentiel permettent de créer des fédérations de référentiel où les applications peuvent facilement retrouver les métadonnées.

4.2.6. Quelques exemples d'utilisation du référentiel

Après toutes ces informations conceptuelles, nous allons passer à un peu de pratique, seulement dans le langage C++ cette fois-ci. Les deux exemples suivants invoquent des opérations des interfaces *CORBA::Repository*, *CORBA::Container*, *CORBA::Contained* et *CORBA::InterfaceDef*.

4.2.6.1. Consulter le référentiel

Le référentiel des interfaces est un objet « notoire » du bus CORBA. Pour obtenir sa référence, il faut donc utiliser l'opération *resolve_initial_references* de *l'ORB* (1). Le nom symbolique *InterfaceRepository* désigne cet objet notoire. Après avoir obtenu la référence, il faut la convertir vers le type *CORBA::Repository* pour pouvoir appliquer les opérations sur le référentiel (*_narrow*) (2).

```
// obtenir la référence sur l'objet notoire IFR                                    (1)
CORBA_Object_var objet = orb->resolve_initial_references ("InterfaceRepository");

// conversion vers le type réel de l'objet                                         (2)
CORBA_Repository_var referentiel = CORBA_Repository::_narrow (objet);

// rechercher une définition par son nom étendu                                    (3)
CORBA_Contained_var definition = referentiel->lookup ("MonModule::MonInterface");

// rechercher une définition par son identifiant d'IFR                             (4)
definition = referentiel->lookup_id ("IDL:MonModule/MonInterface:1.0");
```

Les invocations (3) et (4) illustrent la recherche de l'objet contenant les métadonnées associées à l'interface *MonInterface* du module *MonModule*, respectivement par son nom étendu et par son identifiant de référentiel. Ensuite, la référence retournée est convertie dans le type *CORBA::Container* (5) afin de pouvoir réaliser les traitements suivants.

```
// conversion vers le type voulu                                                   (5)
CORBA_Container_var conteneur = CORBA_Container::_narrow (definition);

// obtenir la liste de tous les objets contenus                                    (6)
CORBA_ContainedSeq_var contenus = conteneur->contents (CORBA_dk_all,CORBA_TRUE);

// parcours de la séquence des objets contenus                                     (7)
cout << "Liste des noms des objets du conteneur" << endl;
for (CORBA_ULong i=0; i< contenus->length(); i++) {
  // consultation du nom de chaque objet contenu                                  (8)
  CORBA_String_var nom = contenus[i]->name ();
  cout << "  " << nom << endl;
}
```

L'invocation de l'opération *contents* (6) sur le conteneur permet d'obtenir les références de tous les objets contenus dans celui-ci car la valeur du premier paramètre est égal à *dk_all*. Le nom de chacun de ces objets est obtenu en consultant l'attribut *name* (8) de chaque élément de la séquence (7).

4.2.6.2. Consulter l'interface d'un objet

Maintenant, nous allons illustrer la manière de découvrir l'interface d'un objet en consultant les métadonnées du référentiel des interfaces. Le code suivant implante une procédure C++ qui affiche les métadonnées associées à une référence d'interface passée en paramètre. La consultation de l'attribut *absolute_name* permet d'obtenir le nom étendu de l'interface (1). L'invocation suivante quant à elle extrait du référentiel la description complète (structure *CORBA::FullInterfaceDescription*) de l'interface via l'opération *describe_interface* (2).

```
void afficher_interface (CORBA_InterfaceDef_ptr interface)
{
  // consultation du nom absolu associé à l'interface                              (1)
  CORBA_String_var absolute_name = interface->absolute_name();
  cout << "Nom absolu = " << nom_absolu << endl;

  // obtenir la description complète de l'interface                                (2)
```

```
  CORBA_InterfaceDef::FullInterfaceDescription_var
                                          description = interface->describe_interface();
```

Les différents champs composant la structure *description* sont ensuite affichés (3). Le champ *operations* de cette même structure contient les métadonnées décrivant la signature de toutes les opérations de l'interface (4). Ces métadonnées sont stockées dans une séquence de structures IDL. Cet exemple se limite à l'affichage du nom de chacune des opérations (5). Bien entendu, il est aussi possible d'afficher la signature complète de l'opération : le type de retour, les noms, les modes de passages ainsi que les types des paramètres et la liste des exceptions et des variables de contextes.

```
  // afficher les informations générales                                          (3)
  cout << "Nom         = " << description->name << endl;
  cout << "ID global   = " << description->id << endl;
  cout << "Défini dans = " << description->defined_in << endl;
  cout << "Version     = " << description->version << endl;

  // la liste décrivant les opérations                                            (4)
  CORBA_OpDescriptionSeq_ptr operations = description->operations;

  cout << "Liste des noms des opérations" << endl;
  for (CORBA_ULong i=0; i<operations->length(); i++)
    cout << "  " << operations[i].name << endl;            // afficher la signature       (5)

  // la liste décrivant les attributs                                             (6)
  CORBA_AttrDescriptionSeq_ptr attributs = description->attributes;

  cout << "Liste des noms des attributs" << endl;
  for (CORBA_ULong j=0; j<attributs->length(); j++) {
    cout << "  " << attributs[j].name << endl;                  // afficher la signature (7)
}
```

De même, les points (6) et (7) permettent d'obtenir la liste des attributs et d'afficher leur nom. Nous aurions pu aussi afficher le mode et le type de ces attributs en consultant les champs des éléments de la séquence *attributs*. Maintenant, il ne nous reste plus qu'à appliquer cette procédure sur la référence de l'interface d'un objet CORBA quelconque (8). Ici, nous utilisons l'opération *_get_interface* disponible sur toute référence d'objet CORBA (9). Cette opération retourne l'objet de l'IFR représentant l'interface de cet objet.

```
  // obtenir une référence d'objet                                                (8)
  CORBA_Object_var objet = ...;

  // obtenir la référence de l'objet d'interface                                  (9)
  CORBA_InterfaceDef_var interface = objet->_get_interface ();

  afficher_interface (interface);
```

Pour découvrir les interfaces des objets, l'opération *describe_interface* est la plus souvent utilisée car grâce à une unique invocation CORBA, nous pouvons découvrir les métadonnées principales d'une interface en consultant la structure retournée *CORBA::FullInterfaceDescription*. Dans cette partie, nous avons fait un tour d'horizon des possibilités et des fonctionnalités du référentiel des interfaces permettant de découvrir dynamiquement les interfaces des objets CORBA. Actuellement, cet outil est souvent sous-utilisé par les fournisseurs de bus CORBA et par les architectes d'applications distribuées. Pourtant, il offre de nombreuses perspectives d'utilisation dynamique du bus CORBA. Nous pensons que c'est un outil très puissant et très intéressant ; il sera la composante-clé utilisée par les futures grandes applications distribuées, surtout dans les applications utilisant les mécanismes dynamiques de CORBA. Par exemple, le langage CorbaScript utilise intensément le référentiel des interfaces pour découvrir les définitions OMG-IDL utilisées par les scripts.

4.3. Le mécanisme DII

4.3.1. Le rôle

Le mécanisme DII permet à une application cliente d'invoquer dynamiquement les opérations des objets sans utiliser les souches IDL précompilées. Un client peut invoquer n'importe quelle opération sur n'importe quel objet via ce mécanisme. Pour cela, le client doit découvrir les informations relatives à l'interface des objets au moment de l'exécution. Il n'a pas besoin de connaître ces informations au moment de la compilation. C'est pour cela que ce mécanisme est qualifié de « dynamique ».

Les nouveaux services offerts par les serveurs sont immédiatement utilisables par les applications clientes sans nécessiter leur modification. Les applications clientes découvrent les fonctionnalités des objets en consultant le référentiel des interfaces. À partir de ces métadonnées, elles peuvent alors construire et invoquer dynamiquement des requêtes via la souche générique offerte par le mécanisme DII. Cette souche est indispensable pour réaliser des applications d'usage général comme, par exemple, un outil de navigation dans les objets (ou « object browser »).

Dans ce contexte, l'outil de navigation doit consulter le référentiel des interfaces pour découvrir les signatures des opérations de l'interface des objets : le nom de l'opération, le type du résultat et les paramètres. Le navigateur affiche alors une représentation visuelle de l'interface de l'objet. L'utilisateur choisit l'opération qu'il veut invoquer et complète les paramètres. Le navigateur construit alors une requête à partir des données saisies par l'utilisateur, invoque l'opération et lui affiche les résultats.

Ainsi, le mécanisme DII est la composante du bus CORBA qui permet de concevoir des applications souples et extensibles. Au moment de la conception de ces applications, le développeur ne connaît aucune information sur la signature des opérations invoquées. L'environnement CORBA se distingue ainsi de tout autre « middleware » existant par son mécanisme innovant d'invocations dynamiques.

4.3.2. Le scénario d'utilisation du DII

Le mécanisme DII permet donc de construire des applications génériques. La réalisation de ces applications est tout de même complexe car il faut développer un programme qui sache invoquer n'importe quelle opération de n'importe quel objet. Afin de donner une idée du travail à réaliser, nous allons décrire le scénario type de l'invocation d'une opération d'un objet. Pour invoquer dynamiquement un objet, il faut :

- trouver la référence d'un objet ;
- obtenir l'interface de l'objet ;
- obtenir la description de l'opération à invoquer ;
- construire la liste des arguments à transmettre ;
- créer la requête ;
- invoquer la requête ;
- traiter les résultats retournés.

Nous allons maintenant détailler chacune de ces étapes. Le mécanisme DII est décrit comme toutes les composantes CORBA via des interfaces IDL. La figure suivante présente les relations entre les différentes interfaces de la souche générique DII ainsi que leurs opérations.

4.3.2.1. Trouver la référence d'un objet

Lorsque nous voulons invoquer dynamiquement un objet, la première étape consiste à obtenir une référence sur cet objet. De nombreuses possibilités nous sont offertes par l'environnement CORBA pour retrouver cette référence. Une chaîne de caractères peut être convertie en une référence d'objet à l'aide de l'opération *string_to_object* de l'interface *CORBA::ORB*. La référence d'objet peut être aussi retrouvée par l'intermédiaire d'un des services de recherche standardisés de CORBA : le service Nommage ou le service Vendeur.

4.3.2.2. Obtenir l'interface de l'objet

Tout objet CORBA est auto-descriptif. Nous pouvons donc obtenir la description de son interface IDL en appliquant l'opération *_get_interface* sur la référence précédemment obtenue. Cette opération retourne une référence sur un objet de type *CORBA::InterfaceDef*. Cet objet du référentiel des interfaces contient les métadonnées décrivant l'interface de l'objet que nous voulons invoquer.

4.3.2.3. Obtenir la description de l'opération à invoquer

Il est ensuite nécessaire de sélectionner l'opération à invoquer. Pour cela, soit l'utilisateur indique quelle opération il souhaite invoquer, soit l'application dynamique lui présente les opérations possibles et le laisse choisir celle qu'il désire.

Dans le premier cas, l'application peut invoquer les opérations *lookup* ou *lookup_name* pour rechercher la référence de l'objet de l'IR décrivant l'opération désirée par l'utilisateur. Il faut ensuite interroger cet objet - par

l'invocation de *describe* - afin d'obtenir la signature IDL de l'opération. Cette signature nous servira dans la suite pour construire dynamiquement la requête.

Dans le second cas, la navigation et la consultation du référentiel des interfaces permettent d'obtenir la liste des opérations de l'interface de l'objet ainsi que les métadonnées de signature. L'invocation des opérations *describe_contents* ou *describe_interface* sur l'objet d'interface permettent d'obtenir ces métadonnées. La différence entre ces deux opérations est que la première permet de sélectionner précisément les informations du contrat IDL tandis que la seconde retourne seulement la description des attributs et des opérations de l'interface IDL. L'application peut alors présenter à l'utilisateur la liste des opérations et lui laisser choisir celle qu'il souhaite.

4.3.2.4. Construire la liste des arguments

À partir des métadonnées décrivant la signature de l'opération choisie par l'utilisateur, l'application peut construire la liste des arguments de la requête dynamique. Cette liste, appelée aussi *CORBA::NVList*, contient chacun des arguments de la requête. Les arguments sont encapsulés dans des objets de type *CORBA::NamedValue*. Ces objets contiennent la valeur de l'argument, le type de celui-ci ainsi que son mode de passage (*in*, *inout* ou *out*). La valeur d'un argument est stockée dans un objet de type *CORBA::Any*. Le type d'un argument est représenté par un objet de type *CORBA::TypeCode*.

L'environnement CORBA fournit des opérations pour construire la liste des arguments d'une requête dynamique. La création d'une liste est réalisée par l'appel de *create_list* de l'interface *CORBA::ORB*. Les opérations *add*, *add_item* et *add_value* de l'interface *CORBA::NVList* permettent d'ajouter des arguments dans cette liste.

Cette liste peut être également construite d'une autre manière en invoquant l'opération *create_operation_list* de l'interface *CORBA::ORB*. Cette opération prend en paramètre la référence d'un objet *CORBA::OperationDef* et elle retourne une liste d'arguments partiellement initialisée. Le type et le mode de passage de chaque argument sont déja fixés ; l'application n'a plus qu'à fixer la valeur de chacun des arguments.

4.3.2.5. Créer la requête

Une fois la liste des arguments initialisée, il est alors possible de créer enfin la requête. L'appel de l'opération *_create_request* sur l'interface *CORBA::Object* instancie un objet *CORBA::Request*. Cet objet doit être initialisé en précisant le nom de l'opération à invoquer, la liste des arguments et une structure pour recevoir le résultat (une *CORBA::NamedValue*).

Plutôt que de construire la liste des arguments puis de créer la requête, il est possible de créer la requête puis de complèter la liste des arguments. L'opération *_request* de *CORBA::Object* initialise une nouvelle requête avec une liste d'arguments vide et un champ pour le résultat. L'application doit alors insérer un à un les arguments dans la requête par des appels aux opérations *add_in_arg*, *add_inout_arg*, *add_out_arg* en fonction du mode de passage.

4.3.2.6. Invoquer la requête

Enfin, nous pouvons invoquer la requête. Le mécanisme DII offre trois modes d'invocation des requêtes :

- **L'invocation synchrone** (*invoke*) bloque l'application jusqu'au retour du résultat de la requête ou simplement la fin de l'exécution de celle-ci. Ce mode a donc le même comportement que l'invocation statique à l'aide de souches IDL.
- **L'invocation différée** (*send_deferred*) permet à l'application de continuer son exécution pendant l'exécution de la requête. Elle obtient ultérieurement le résultat soit par une attente bloquante via *get_response*, soit par scrutation non bloquante via *poll_response*. Ce mode d'invocation est uniquement disponible avec le DII. Pour avoir le même comportement avec les souches IDL, il faut utiliser plusieurs flots d'exécution concurrents (ou « multithreading »).
- **L'invocation asynchrone** (*send_oneway*) est utilisée lorsque l'opération invoquée est déclarée en mode *oneway* ou lorsque l'application ne désire pas connaître le résultat de la requête ni attendre la fin de son exécution.

Une application peut émettre simultanément plusieurs requêtes vers plusieurs objets. Pour cela, elle construit les différentes requêtes et l'interface CORBA::ORB fournit des opérations pour grouper l'émission des requêtes : *send_multiple_requests_deferred* et *send_multiple_requests_oneway*. L'application attend alors les retours de ces requêtes par l'intermédiaire de *get_next_response* et *poll_next_response*.

Le mécanisme DII offre donc de nombreux modes d'invocation plus souples que ceux offerts par les souches IDL. Néanmoins, l'OMG travaille sur la spécification d'un service de messagerie asynchrone (CORBA Messaging) qui permettra d'avoir en invocation statique la souplesse des modes d'invocation du mécanisme DII.

4.3.2.7. Traiter les résultats

Finalement, l'application dynamique peut extraire le résultat de la requête par l'intermédiaire de l'opération *return_value*. Les valeurs des paramètres en sortie - modes *inout* et *out* - sont alors entreposées dans la liste des arguments passée à la création de la requête. Cette liste peut être obtenue en consultant l'attribut *arguments* de l'objet *CORBA::Request*. Lorsqu'une exception a été déclenchée par l'opération invoquée, celle-ci est stockée dans l'environnement de la requête désigné par l'attribut *env* de type *CORBA::Environment*.

Comme nous venons de le voir au cours de ce scénario, le mécanisme DII est assez complexe à mettre en œuvre dans une application. De plus, chaque étape de scénario peut être réalisée de différentes manières comme, par exemple, la création d'une requête via *_create_request* ou *_request*. Le langage CorbaScript permet de simplement exprimer les invocations sur les objets CORBA à travers la notation *objet.methode(p1,…,pn)*. L'interpréteur réalise tout le travail pour invoquer l'objet à travers le mécanisme DII en appliquant grosso modo le scénario présenté ci-avant.

4.3.3. Les interfaces du DII

Comme nous venons de le voir dans le scénario précédent, l'API du mécanisme DII est relativement complexe. Toutefois, ce scénario n'a pas présenté toutes les interfaces et opérations disponibles dans cette API. Voici la liste exhaustive de ces interfaces :

- ***CORBA::ORB*** est l'interface qui fournit les opérations de création d'objets de certaines interfaces du DII. En plus d'instancier des *CORBA::Object*, les opérations *create_*_tc* créent les différentes sortes de *CORBA::TypeCode* : *create_interface_tc* permet de créer une métadonnée décrivant une interface IDL. Les autres opérations *create_** permettent de créer des objets des interfaces correspondantes. De plus, l'*ORB* fournit les opérations pour grouper des invocations de requêtes.
- ***CORBA::Object*** est l'interface de base qui encapsule les références de tous les types d'objets CORBA. Son opération *_get_interface* permet d'obtenir les métadonnées du référentiel des interfaces décrivant l'interface IDL de l'objet. Ces métadonnées sont essentielles pour la construction des requêtes. Ces requêtes peuvent être créées de deux manières différentes : *_create_request* nécessite de créer et d'initialiser la valeur de tous les champs de la requête tandis que *_request* fournit une version simplifiée de la création de requêtes.
- ***CORBA::Request*** est l'interface au centre du mécanisme DII : une requête est représentée sous la forme d'un objet. Cette interface fournit les attributs caractérisant la requête comme l'objet cible, le nom de l'opération, la liste des arguments et le résultat. L'ajout de nouveaux arguments dans la liste est réalisé via les opérations *add_*_arg*. Enfin, cette interface fournit trois modes d'invocation des requêtes, c'est-à-dire les opérations *invoke*, *send_oneway* et *send_deferred*. Dans ce dernier mode, les opérations *get_response* et *poll_response* permettent d'attendre les résultats.
- ***CORBA::NVList*** est l'interface d'aide à la construction de la liste des arguments d'une requête. Cette structure de données fournit des opérations pour ajouter, obtenir et supprimer des arguments dans la liste. Chaque élément de la liste est un objet de type *CORBA::NamedValue*.
- ***CORBA::NamedValue*** est l'interface des objets encapsulant les arguments et le résultat des requêtes. Ces objets sont composés d'un nom symbolique, d'une valeur de type *CORBA::Any* (voir le paragraphe suivant) et d'un drapeau indiquant le mode de passage des arguments.
- ***CORBA::Environment*** est l'interface décrivant l'environnement d'exécution d'une requête. Elle contient l'exception déclenchée en cas de problèmes survenus sur le bus CORBA ou lors de l'exécution de l'opération invoquée par une requête.
- ***CORBA::Exception*** est le type de base de toutes les exceptions.
- ***CORBA::ExceptionList*** est l'interface permettant de préciser la liste des exceptions utilisateurs qu'une opération peut déclencher. Cette liste est définie pendant l'initialisation de chaque requête. Elle peut être construite en consultant la signature de l'opération stockée dans le référentiel des interfaces.

Titre:

Auteur:
idraw
Aperçu:
Cette image EPS n'a pas été enregistrée
avec un aperçu intégré.
Commentaires:
Cette image EPS peut être imprimée sur une
imprimante PostScript mais pas sur
un autre type d'imprimante.

- ***CORBA::ContextList*** est l'interface permettant de préciser la liste des variables de contexte que nécessite une opération. Cette liste est définie pendant l'initialisation de chaque requête. Elle peut être construite en consultant la signature de l'opération stockée dans le référentiel des interfaces.
- ***CORBA::Context*** est l'interface permettant de fixer les valeurs des variables de contextes d'une requête. Elle fournit des opérations pour fixer, détruire et obtenir les valeurs des variables de contextes. De plus, il est possible de définir une hiérarchie des contextes pour organiser et fixer à chaque niveau des valeurs de contextes distinctes.

4.4. Les métatypes auto-descriptifs Any et TypeCode

Le type de données *any* du langage IDL est représenté par la classe C++ *CORBA::Any*. Ce métatype est un type de métadonnées auto-descriptif : il encapsule n'importe quelle valeur IDL et la description du type IDL associé à la valeur, c'est-à-dire un *CORBA::TypeCode*. Ce métatype est principalement utilisé dans les mécanismes dynamiques de CORBA pour fixer les valeurs des arguments ou pour stocker le résultat d'une requête (*c.f. CORBA::NamedValue*). Il peut être également utilisé dans les contrats IDL lorsque les types de données manipulés par une opération ou un attribut ne sont pas connus dans la phase de conception. Il faut savoir aussi que récemment l'OMG a ajouté l'API *DynAny* qui permet de construire dynamiquement des valeurs *any*.

Le métatype *CORBA::TypeCode* est également un type de données auto-descriptif : il encapsule la description de n'importe quel type IDL. Ce métatype est utilisé pour spécifier le type IDL d'une valeur *Any* ainsi que les types manipulés par les mécanismes DII et DSI. Le référentiel des interfaces utilise aussi ce métatype pour représenter les métadonnées de typage (type d'un attribut, type d'un paramètre, etc.). Les valeurs *TypeCode* peuvent être obtenues à partir du référentiel des interfaces, générées automatiquement par les pré-compilateurs IDL ou bien créées par des opérations fournies par l'interface *ORB* (*c.f.* les opérations *create_*_tc*). Les CORBA::TypeCode sont eux-mêmes décrits par une interface OMG-IDL que nous ne détaillons pas dans ce document.

4.5. Le mécanisme DSI

Le mécanisme DII offre une souche générique pour les applications clientes leur permettant d'invoquer dynamiquement n'importe quel objet CORBA. De manière analogue, le mécanisme DSI offre un squelette générique pour les applications serveurs. Ce squelette générique permet alors de recevoir n'importe quelle requête adressée à n'importe quel type d'objets CORBA. Par l'intermédiaire du mécanisme DSI, un serveur n'a pas besoin d'intégrer les squelettes statiques générés par les précompilateurs IDL et il peut implanter n'importe quel type d'objets CORBA.

À l'origine, le mécanisme DSI fut défini pour permettre la mise en place de passerelles entre différents bus CORBA. Lorsque un bus non-IIOP, c'est-à-dire un bus qui implante un protocole propriétaire pour transporter les requêtes entre des objets répartis sur le réseau, veut invoquer des objets d'un bus IIOP, il est nécessaire de transformer les requêtes de son format propriétaire vers le format IIOP. Ces conversions de protocoles sont effectuées par l'intermédiaire d'une passerelle : les requêtes arrivent dans le format propriétaire et elles sont ensuite traduites dans le format IIOP. Deux types de passerelles sont envisageables. Les passerelles statiques reçoivent les requêtes via des squelettes IDL mais, dans ce cas, soit elles ne traitent qu'un nombre limité de types IDL définis à la compilation de la passerelle, soit elles doivent contenir tous les squelettes possibles. Par contre, les passerelles dynamiques reçoivent les requêtes via le mécanisme DSI et peuvent ainsi convertir toute requête.

Le mécanisme DSI peut également être utilisé pour la conception d'outils de développement interactifs comme des interpréteurs ou des outils de déverminage et de trace. À travers une API unique, ces outils peuvent recevoir et ensuite traiter des requêtes pour n'importe quel type d'objets CORBA comme c'est le cas pour le langage CorbaScript. Comme le mécanisme DII, il est souvent nécessaire d'utiliser conjointement le mécanisme DSI avec le référentiel des interfaces.

L'API du mécanisme DSI est composée de l'interface *CORBA::ServerRequest*. Cette interface représente une requête à destination d'un objet implanté dynamiquement par un serveur. Elle fournit des opérations pour obtenir le nom de l'opération invoquée (*op_name*), l'objet du référentiel des interfaces décrivant la signature de l'opération invoquée (*op_def*), les variables de contexte (*ctx*) et la liste des paramètres (*params*). Les opérations *result* et *exception* permettent respectivement d'indiquer la valeur de retour de la requête ou l'exception déclenchée en cas de problème.

```
module CORBA {
  typedef string Identifier;
```

```
    interface ServerRequest {
      Identifier op_name ();
      OperationDef op_def ();
      Context ctx ();
      void params (inout NVList params);
      void result (in any r);
      void exception (in any r);
    };
  };
```

Cette API est simple car elle réutilise une grande partie de l'API du DII : *CORBA::NVList*, *CORBA::NamedValue*, *CORBA::Any*, *CORBA::TypeCode* et *CORBA::Context*. Toutefois, elle n'est pas actuellement clairement spécifiée : ceci implique que chaque implantation du bus CORBA propose sa propre interprétation de cette API. Par exemple dans l'implantation de CorbaScript, nous avons un fragment de code qui doit être adapté à chaque bus CORBA.

5. Conclusion

Ce document nous a permis de faire un tour d'horizon du monde CORBA et d'illustrer concrètement les étapes techniques de la construction d'une application répartie au dessus du bus CORBA. Les sources complets de cette application sont disponibles sur notre site WWW : ils comprennent l'ensemble des codes C++, Java et CorbaScript présentés ci-avant et en plus les sources d'une application Java cliente utilisant AWT pour l'interface graphique.

5.1. Pourquoi choisir CORBA ?

De nombreuses autres solutions « middlewares » telles que DCE, Java et les propositions Microsoft pour la construction et l'exécution d'applications réparties existent. Cependant à travers l'application décrite, nous pouvons constater que la norme CORBA se distingue car elle offre :

- **Une solution ouverte et évolutive** : choisir CORBA revient à ne pas s'enfermer dans une solution propriétaire évoluant difficilement. L'OMG réagit très vite aux demandes du marché. Par exemple, CORBA a su très rapidement intégrer le langage Java et offre des passerelles d'interopérabilité avec d'autres mondes comme DCE, COM et bientôt DCOM. De plus, la portabilité visée par l'OMG permet de changer plus ou moins facilement de fournisseurs de technologies CORBA.
- **Une architecture modulaire** : grâce à l'OMA, une application répartie n'est pas construite à partir de zéro mais réutilise des composants logiciels offrant des fonctions orientées système, utilisateur et/ou métier.
- **L'interopérabilité entre composants hétérogènes** : CORBA fournit les abstractions et mécanismes offrant un bus orienté objet d'intégration de composants logiciels conçus avec des technologies hétérogènes (langages, systèmes d'exploitation, machines et réseaux). Ainsi, diverses technologies peuvent être mises en œuvre et coopérer dans la même application. L'application décrite montre cela à travers trois langages.
- **Le libre choix des technologies d'implantation** : dans notre application, nous n'avons imposé aucune contrainte aussi bien au niveau des réseaux, des machines, des systèmes d'exploitation que des langages de programmation.
- **La portabilité du code** : les fragments de code présentés sont totalement portables d'une implantation du bus à une autre. Aucune supposition technique n'a été faite sur le bus utilisé. Cependant, la projection IDL/C++ n'assure pas encore une totale portabilité à 100% car elle propose des alternatives (e.g. la projection des modules).
- **La simplicité de mise en œuvre** : la mise en pratique de CORBA n'est pas foncièrement complexe, il suffit de se plier au modèle CORBA et à un ensemble de règles de programmation des applications. C'est même très simple à travers CorbaScript.
- **L'encapsulation de l'existant** : la réutilisation de codes existants (voire même d'applications complètes) peut être réalisée par encapsulation de ceux-ci dans des objets CORBA. Ces objets sont alors utilisables pour bâtir de nouvelles applications.

Toutefois, lorsque l'on utilise CORBA pour bâtir une application répartie, il faut faire très attention à ne pas trop tomber dans les pièges tendus par les fournisseurs de bus : il faut éviter au maximum d'utiliser les mécanismes propriétaires (e.g. l'opération *_bind*) qui vous lient à un fournisseur ou alors isoler leurs utilisations afin de pouvoir plus facilement changer de bus CORBA lorsque le besoin s'en fait sentir.

Néanmoins, CORBA s'impose progressivement comme le choix incontournable quel que soit le domaine d'applications réparties abordé.

5.2. L'évolution vers CORBA 2.2

La norme CORBA ne s'est pas faite en un jour, elle est le produit d'un long travail de consensus industriel commencé en 1989. Au cours de son évolution, de nombreux ajouts et modifications lui ont été apportés. CORBA 1.0 avait pour objectif l'interopérabilité entre objets réparties. Cela fut atteint grâce aux briques de base telles que le langage OMG-IDL, l'ORB, l'adaptateur d'objets BOA et les mécanismes dynamiques (IFR, DII). Cependant, cette version ne prévoyait pas grand chose pour l'interopérabilité entre différentes implantations du bus. CORBA 2.0 avait donc comme objectif principal d'offrir cette interopérabilité via le protocole GIOP et IIOP. Actuellement, nous en sommes à la version 2.2 qui corrige certaines imperfections observées et a ajouté de nouvelles composantes. En résumé, CORBA 2.2 définit :

- le modèle abstrait orienté objet et client/serveur (*c.f.* section 1.2) ;
- le langage OMG-IDL (*c.f.* section 2) ;
- les briques de base telles que le module *CORBA*, les interfaces *ORB* et *Object* (*c.f.* section 3.5) ;
- l'adaptateur d'objets portable ou POA ;
- les mécanismes dynamiques tels que le référentiel des interfaces (IFR), l'invocation dynamique DII, l'implantation dynamique DSI et la gestion dynamique des *DynAny* (*c.f.* section 4);
- l'interopérabilité entre bus CORBA via le protocole générique GIOP et son instanciation IIOP ;
- l'interopérabilité avec le monde COM ;
- les intercepteurs pour placer des traitements globaux sur les requêtes ;
- les règles de projection IDL vers les langages C, C++, SmallTalk, COBOL, ADA et Java.

Malheureusement, ce document ne peut pas aborder toutes ces composantes dans les moindres détails. Nous avons donc opéré une sélection sur ce qu'il est indispensable de connaître de la norme CORBA. À partir de cette information, le lecteur sera mieux armé pour étudier les composantes que nous avons laissé de côté. Cependant, il faut bien être conscient que l'OMG définit seulement des spécifications. Ces spécifications sont ensuite implantées dans des produits « à la norme CORBA ». Ainsi, actuellement, aucun produit n'implante complètement les spécifications CORBA 2.2 : chaque produit sélectionne les composantes qui lui paraissent le « plus vendeur ». Aussi, il ne faut pas hésiter à choisir le produit CORBA le plus approprié à chaque partie d'une application : un produit offrant le POA pour écrire des serveurs C++ portables et un produit offrant la projection IDL/Java pour les applications clientes utilisées à travers le WWW.

5.3. Vers CORBA 3.0

La norme CORBA 3.0 sera la prochaine révision majeure annoncée par l'OMG pour la fin de l'année 1998 (voire courant 1999 :-). Il est encore trop tôt pour prédire exactement ce qu'apportera cette révision. Toutefois, voici la liste des travaux actuellement menés au sein du groupe de travail sur le bus CORBA (le groupe ORBOS) :

- **Multiple Interfaces and Composition** fournira les mécanismes permettant à un objet CORBA d'implanter plusieurs interfaces OMG-IDL.
- **Messaging Service** définit un nouveau modèle de communication asynchrone lorsque l'objet appelant et l'objet appelé ne sont pas présents simultanément sur le bus (via des requêtes persistantes).
- **Objects-by-Value** permet aux requêtes de transférer directement des objets CORBA et non pas seulement des structures de données.
- **Java to IDL** permet de transformer automatiquement des objets Java en objets CORBA.
- **DCE/CORBA Interworking** offrira différentes technologies pour assurer l'interopérabilité entre des applications CORBA et des applications DCE.
- **Persistent State Service 2.0** sera une spécification simplifiée (et plus exploitable) du service Persistance.
- **CORBA Component Model** définira un modèle de composants pour le monde CORBA en s'inspirant par exemple du modèle JavaBeans.
- **CORBA Scripting Language** définit un cadre généraliste pour le scriptage d'objets et de composants CORBA. Ce cadre est ensuite instancié vers les langages JavaScript, Python, Tcl et bien entendu vers notre langage de scripts CorbaScript.

- **Minimum CORBA** définira le sous-ensemble des interfaces du bus CORBA strictement nécessaire pour construire des bus et des applications embarquées.
- **Realtime CORBA** définira de nouvelles interfaces du bus CORBA pour un contrôle fin des mécanismes internes du bus afin de supporter des applications temps réel.
- **Firewall** définira les technologies pour utiliser CORBA à travers des réseaux Internet protégés par des pares-feux.
- **Printing Facility** définit les interfaces pour la gestions des impressions. C'est l'utilitaire commun Impression.
- **Interoperable Name Service** définira les interfaces pour fédérer des services Nommage sur Internet.
- **Fault Tolerance** définira les services système prenant en charge les fonctions liées à la tolérance aux pannes comme, par exemple, la réplication de serveurs.

Tous ces travaux ne seront pas forcément terminés ou intégrés dans CORBA 3.0. Mais, ils donnent une idée sur les fonctionnalités de la prochaine génération de bus CORBA. L'objectif sera de mettre l'accent sur la flexibilité et l'adaptabilité du bus aux besoins des utilisateurs (les composants, les scripts) mais aussi aux besoins des architectes système (communication asynchrone, bus minimal, temps réel, tolérance aux pannes).

A. Références bibliographiques

Ce document se termine par un ensemble de références bibliographiques classées en trois catégories : les documents OMG, les ouvrages indispensables et divers pointeurs intéressants.

A.1. Les documents OMG

Tous les documents produits par l'OMG sont dans le domaine public et sont accessibles sur les sites Internet de l'OMG : http://www.omg.org et ftp://ftp.omg.org/pub/docs/.

[OMAG 95] « Object Management Architecture Guide, revision 3.0 »
Richard Mark Soley et Christopher M. Stone.
Object Management Group, Inc. et John Wiley & Sons, Inc., USA, Juin 1995.
OMG TC Document ab/97-05-05
ISBN : 0-471-14193-3

[UML 97] « Unified Modelling Language »
OMG, USA, Novembre 1997.
OMG TC Document ab/97-08-02 à ab/97-08-09
http://www.omg.org/library/schedule/Technology_Adoptions.htm#tbl_UML_Specification

[COS 98] « CORBAservices : Common Object Services Specification »
OMG, USA, Novembre 1997.
OMG TC Document formal/98-07-05
http://www.omg.org/corba/sectrans.htm

[CORBA 98] « The Common Object Request Broker : Architecture and Specification, revision 2.2 »
OMG, USA, Février 1998.
OMG TC Document formal/98-07-01
http://www.omg.org/corba/corbiiop.htm

Appelé aussi « CORBA/IIOP 2.2 Specification ».

[OMG-CM] « CORBA Messaging »
OMG, USA, Mai 1998.
OMG TC Document orbos/98-05-06
http://www.omg.org/library/schedule/Messaging_Service_RFP.htm

En cours d'adoption durant le mois d'août 1998.

[OMG-OBV] « Objects-by-Value »
OMG, USA, Juillet 1998.
OMG TC Document orbos/98-01-18

Adopté au mois de Juillet 1998.

[OMG-MI] « Multiple Interfaces and Composition RFP »
http://www.omg.org/library/schedule/Multiple_Interfaces_and_Composition.htm

Il y a encore trop de propositions pour voir se dégager un standard (donc à suivre). Soumission révisée pour le 19 octobre 1998.

[OMG-MC] « CORBA Component Model RFP »
http://www.omg.org/library/schedule/CORBA_Component_Model_RFP.htm

Il y a encore trop de propositions pour voir se dégager un standard (donc à suivre). Soumission révisée pour le 19 octobre 1998.

[OMG-CSL] « CORBA Scripting Language RFP »
http://www.omg.org/library/schedule/CORBA_Scripting_Language_RFP.htm

Nous proposons CorbaScript comme langage de scripts pour CORBA. Soumission révisée pour le 21 décembre 1998.

L'OMG peut aussi être contacté à l'adresse suivante : Object Management Group, Inc., Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, USA, Tel. +1-508-820-4300, Fax +1-508-820-4303.

A.2. Quelques ouvrages indispensables

[Orfali 96] « Objets répartis - Guide de survie »
Robert Orfali, Dan Harley et Jeri Edwards.
International Thomson Publishing France, Paris, France, 1996.
ISBN : 2-84180-043-1

Très bon ouvrage de référence sur les objets répartis incluant une partie sur CORBA et COM. Voir aussi les autres ouvrages de ces auteurs.

[Geib 97] « CORBA : des concepts à la pratique »
Jean-Marc Geib, Christophe Gransart et Philippe Merle.
Collection InterEditions, Editions MASSON, Paris, France, 1997.
ISBN : 2-225-83046-0
http://corbaweb.lifl.fr

Excellent ouvrage sur CORBA que nous ne pouvons que vous conseiller :-)

A.3. Autres pointeurs intéressants

[Spring 95] « The Spring Object Model »
Sanjay R. Radia, Graham Hamilton, Peter B. Kessler et Michael L. Powell.
Proceedings of USENIX Conference on Object-Oriented Technologies,
Monterey CA, U.S.A., Juin 1995.

[OOC 98] « Object-Oriented Concepts, Inc. Home Page »
Object-Oriented Concepts, Inc., Billerica, MA, USA, 1998.
http://www.ooc.com

Site Web officiel de la société Object-Oriented Concepts diffusant les sources de l'excellent ORBacus (ex OmniBroker) pour C++ et Java.

[TAO 98] « Real-time CORBA with TAO (The ACE ORB) »
Douglas C. Schmidt, 1998.
http://siesta.cs.wustl.edu/~schmidt/TAO.html

L'équipe de Douglas C. Schmidt travaille autour des ORBs pour les applications temps réels. TAO est le premier ORB proposant une implantation du Portable Object Adapter.

[Schmidt 98] « Douglas C. Schmidt's Welcome Page »

Douglas C. Schmidt, 1998.
http://www.cs.wustl.edu/~schmidt/

Douglas C. Schmidt est un gourou CORBA de renommée internationale. Sa page WWW contient les références de nombreux articles scientifiques, des tutoriels, des cours et la série d'articles sur CORBA qu'il a écrit avec Steve Vinoski pour la revue « C++ Report ».

B. Remerciements

Nous tenons à remercier tous nos collègues du LIFL qui ont bien voulu relire et corriger ce document. De plus, n'hésitez pas à nous contacter pour nous faire part de vos commentaires afin d'améliorer ce document.