

---

# Adaptabilité dynamique des aspects pour la construction d'applications réparties ouvertes

Thomas Ledoux<sup>(\*)</sup>

Thomas.Ledoux@emn.fr  
Tel: +33 (0)2 51 85 82 19  
Fax: +33 (0)2 51 85 82 49  
École des Mines de Nantes  
4 rue Alfred Kastler BP 20722  
F-44307 Nantes Cedex 3 France

<sup>(\*)</sup>Les résultats présentés dans cet article ont été obtenus alors que l'auteur était en bourse CIFRE avec IBM Global Services.

---

**RÉSUMÉ** — *Aujourd'hui, la complexité croissante des systèmes informatiques nécessite de trouver de nouveaux mécanismes d'abstraction pour améliorer la conception et l'implémentation des applications. Le paradigme de la séparation des aspects offre une approche intéressante pour construire des composants logiciels réutilisables. Dans cet article, nous proposons d'étendre ce paradigme pour prendre en compte l'évolutivité des systèmes pendant leur exécution. Cette extension, nommée adaptabilité dynamique des aspects, permet à un système de remplacer un aspect spécifique par un autre, pendant son exécution, sans affecter le reste du système. Dans le contexte des applications réparties, les concepts de séparation et d'adaptabilité dynamique des aspects contribuent grandement à la construction d'architectures ouvertes. Ils permettent de modéliser, de réutiliser et d'adapter les nombreux mécanismes liés à la répartition. Nous illustrons cette approche dans le cadre de la plate-forme CORBA. Dans un premier temps, nous établissons une correspondance entre aspects et métaclasses, de façon à mettre en œuvre la séparation et l'adaptabilité dynamique des aspects grâce à la réflexion. Puis, nous présentons OpenCorba, un ORB ouvert, capable d'adapter dynamiquement les stratégies de représentation et d'exécution du bus logiciel.*

**MOT-CLÉS** : *adaptabilité dynamique, aspects, CORBA, langages à objets, métaclasses, OpenCorba, réflexion, Smalltalk*

**ABSTRACT** — *Today, the growing complexity of computer systems requires new mechanisms of abstraction in order to improve the design and implementation of the applications. Separation of aspects brings an interesting approach for building reusable software components. In this paper, we propose to extend this paradigm in order to take into account the evolution of systems at run-time. This extension, named dynamic adaptability of aspects, enables users to replace a specific aspect of a system with another, during its execution, without affecting the rest of the system. In the context of distributed applications, the concept of separation and dynamic adaptability of aspects contribute greatly towards the building of open architectures. They allow reification, reuse and adaptability of distributed mechanisms. We illustrate this approach with the CORBA platform. First, we establish connections between aspects and metaclasses, in order to implement separation and dynamic adaptability of aspects with reflection. Then, we present OpenCorba, a reflective ORB, enabling users to dynamically adapt representation and execution policies of the software bus.*

**KEYWORDS**: *dynamic adaptability, aspects, CORBA, object oriented languages, metaclasses, OpenCorba, reflection, Smalltalk*

---

## 1. INTRODUCTION

Depuis une dizaine d'années, la programmation orientée objet est présentée comme une technologie importante pour aider à la construction de systèmes complexes comme les architectures réparties. Cependant, force est de constater que la technologie objet, malgré ses nombreux atouts — tels que l'encapsulation, le polymorphisme, la modularité — n'apporte pas toujours une réponse aux problèmes de conception et de développement. Ainsi, le concept de *réutilisabilité*, présenté comme le fer de lance de la programmation orientée objet, est souvent mis en défaut [Matsuoka 93, Gamma 95, Kiczales 96, Steyaert 96].

L'un des obstacles majeurs à la réutilisabilité est lié à l'entrelacement dans le programme fonctionnel, de plusieurs aspects techniques (synchronisation, gestion mémoire, gestion d'erreurs) dépendant du domaine de l'application. Aussi, le concept des *aspects* [Hürsh 95, Kiczales 97] vient à émerger dans le but de proposer une nouvelle unité de modularité pour la construction de logiciels. Ce concept permet d'envisager l'isolement d'aspects spécifiques pour leur réutilisation. Le programme fonctionnel et les aspects se retrouvent ainsi séparer et peuvent évoluer indépendamment, formulant ainsi le paradigme de la *séparation des aspects*.

Dans cet article, nous proposons d'étendre ce paradigme pour prendre en compte l'évolutivité des systèmes pendant leur exécution. En effet, les systèmes complexes comme les architectures réparties, doivent supporter la modification dynamique de leurs mécanismes et stratégies (équilibrage de charge, tolérance aux fautes) pour s'adapter à des contextes d'exécution changeants. Nous définissons alors *l'adaptabilité dynamique des aspects* comme un paradigme permettant à un système de remplacer un aspect spécifique par un autre, pendant son exécution, sans affecter le reste du système. Ce nouveau paradigme est nécessaire pour assurer la flexibilité et l'ouverture des applications complexes.

Pour valider cette approche, nous nous sommes intéressés aux architectures réparties. Nous avons établi dans un premier temps une correspondance entre les aspects et la réflexion [Smith 82], de façon à mettre en œuvre la séparation et l'adaptabilité dynamique des aspects. Puis, nous avons développé un ORB réflexif et *ouvert*, nommée OpenCorba [Ledoux 98], capable d'adapter dynamiquement les stratégies de représentation et d'exécution du bus logiciel CORBA [OMG 95].

Cet article est présenté comme suit. La section 2 rappelle le paradigme de la séparation des aspects et introduit celui de l'adaptabilité dynamique des aspects. La section 3 présente une mise en œuvre de ces concepts dans le cadre des langages à classes réflexifs. Puis, la section 4 illustre notre approche dans le contexte de la plate-forme CORBA, en développant plus particulièrement deux aspects du bus logiciel. Ensuite, la section 5 présente les travaux connexes et introduit nos futurs projets. Enfin, nous concluons sur les apports de l'adaptabilité dynamique des aspects dans le contexte des architectures réparties.

## 2. ASPECTS ET ARCHITECTURES OUVERTES

Dans cette section, nous montrons comment les *aspects* contribuent à la construction d'architectures réutilisables et adaptables, i.e. *ouvertes*.

## 2.1 Séparation des aspects

La complexité croissante des applications logicielles est en partie due à la nécessité pour les programmes de prendre en compte des aspects spécifiques des plus variés : concurrence, contraintes de temps réel, distribution, persistance, etc. Quoique clairement identifiés lors de la conception, les principaux aspects se trouvent entremêlés dans l'implémentation rendant le code difficilement lisible, tout juste maintenable et non réutilisable. Aussi, pour Gregor Kiczales, il existe deux types de code dans une implémentation : le *code fonctionnel* qui décrit des fonctionnalités du composant métier et le code des *aspects* qui « traverse » (*cross-cut*) le code fonctionnel pour en affecter sa performance ou sa sémantique [Kiczales 97].

Dans le but de permettre une meilleure réutilisabilité des composants logiciels, le paradigme de la *séparation des aspects*<sup>1</sup> [Hürsh 95, Lieberherr 96, Kiczales 97] propose une implémentation qui respecte une séparation claire entre le programme fonctionnel et les aspects spécifiques de l'application. Par exemple, pour une application bancaire, le code fonctionnel sera les objets métiers (compte bancaire), et les aspects spécifiques, des mécanismes de bas niveau comme la gestion des accès concurrents ou des problèmes de sécurité sur ce compte. Ainsi, il est non seulement possible de réutiliser le code fonctionnel dans d'autres contextes (centralisé vs. distribué), mais aussi les aspects spécifiques dans d'autres domaines d'applications. La séparation des aspects favorise donc considérablement la modularité et la réutilisabilité des architectures logicielles.

## 2.2 Adaptabilité dynamique des aspects

La séparation des aspects est caractérisée par un couplage lâche entre les différents aspects. Un changement sur l'un des aspects n'a en effet aucune incidence sur un autre. Un aspect peut être ainsi remplacé sans pour autant remettre en cause la cohérence de l'application. Le système peut évoluer, par exemple, d'une stratégie de distribution simple vers une stratégie de distribution avec migration, sans affecter le programme fonctionnel et les autres aspects spécifiques. En admettant que l'on puisse remplacer, puis assembler les aspects à l'exécution, le système pourra instaurer dynamiquement de nouvelles stratégies d'exécution — telle que la migration des objets — mieux adaptées au comportement de l'application.

Nous définissons par *adaptabilité dynamique des aspects*, la possibilité pour un système de remplacer à l'exécution un aspect spécifique par un autre aspect *sans* affecter le reste du système. Une application supportant l'adaptabilité dynamique des aspects peut instaurer de nouveaux mécanismes ou modifier des mécanismes existants, pendant son exécution. L'adaptabilité dynamique permet alors de répondre à la problématique délicate des systèmes devant évoluer sans interrompre leur exécution (passage de la numérotation à dix chiffres pour la téléphonie,

---

<sup>1</sup> Le rapport technique « *separation of concerns* » [Hürsh 95] est l'un des premiers articles à formaliser les concepts d'un nouveau paradigme dans le domaine du génie logiciel. Depuis, les travaux de Gregor Kiczales sur AOP (*Aspect Oriented Programming*) [Kiczales 97] ont contribué à l'émergence de ce paradigme. *Separation of concerns* est intraduisible en français ! Notre traduction, *séparation des aspects*, correspond intuitivement à l'idée.

chaîne de montages). L'adaptabilité dynamique des aspects est donc un mécanisme crucial pour assurer l'évolutivité et la flexibilité des systèmes complexes comme les architectures réparties.

### 2.3 Illustration

Pour montrer l'importance de la séparation et de l'adaptabilité dynamique des aspects, nous proposons un exemple associant successivement deux aspects différents à un même objet.

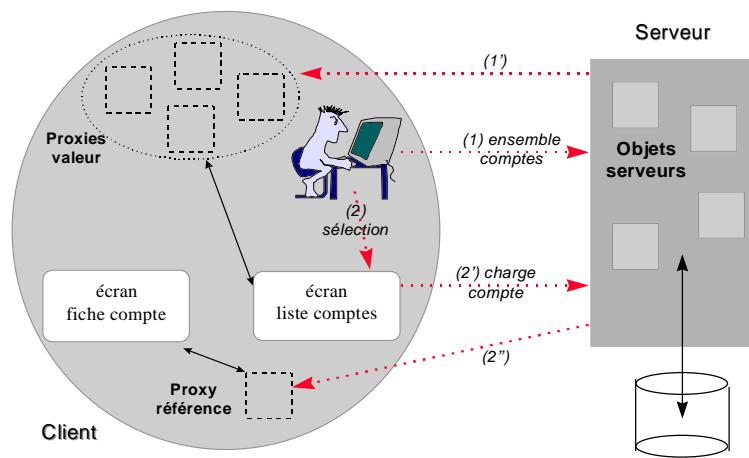


Figure 1-Exemple d'application bancaire

La Figure 1 modélise deux interactions dans une application bancaire de type client/serveur. Dans un premier temps, le guichetier consulte la liste des comptes qu'il peut administrer (1). Un minimum d'informations (nom, numéro compte) lui est renvoyé par copie (1'). Le système utilise pour cela l'aspect **Proxy-valeur** pour gérer la représentation des comptes sur le client. Quand il sélectionne un compte donné (2), une image des informations sur ce compte lui est renvoyée (2') (2''). Le système lui retourne cette fois-ci une référence sur ce compte grâce à l'aspect **Proxy-référence**. Cet exemple montre que, pour un même compte, deux mécanismes différents de passage des données sont utilisés.

L'utilisation des *proxies* en tant qu'aspects va permettre la construction d'une architecture réutilisable et adaptable. En effet, la *séparation des aspects* externalise la gestion des *proxies* de l'application : les aspects *proxies* peuvent être réutilisés dans d'autres domaines d'applications comme l'assurance. L'*adaptabilité dynamique* des aspects permet la modification du mécanisme des *proxies* en fonction du contexte de l'application (passage des arguments). Ainsi, la mise en œuvre des concepts de séparation et d'adaptabilité dynamique des aspects, favorise la construction d'applications ouvertes.

### 3. ASPECTS ET RÉFLEXION

La mise en œuvre du paradigme des aspects nécessite un mécanisme d'assemblage pour lier les aspects entre eux [Hürsh 95]. Ce mécanisme dépend étroitement du processus de séparation initial. Dans cette section, nous présentons une solution de mise en œuvre basée sur le concept de la *réflexion*.

#### 3.1 Rappel de la réflexion

Un système est dit réflexif s'il possède une auto-représentation décrivant la connaissance qu'il a de lui-même. Un tel système peut alors répondre à des questions le concernant, mais aussi s'auto-modifier. Il a toujours une représentation précise de lui-même et son comportement est en accord avec cette représentation [Maes 87]. Les diverses caractéristiques des programmes sont rendues concrètes (i.e. réifiés) et peuvent être manipulées par des *méta*-programmes.

Grâce à l'abstraction des données (i.e. encapsulation) et à leur organisation (i.e. héritage), les langages à objets offrent un cadre privilégié à la mise en œuvre d'architectures réflexives. Les *métaobjets* décrivent la représentation et contrôlent le comportement des objets. Les protocoles à métaobjets (*Meta Object Protocol* ou MOP) règlent la communication entre les objets et les métaobjets [Kiczales 91]. Il est alors possible de spécialiser les métaobjets fournis en standard pour introduire de nouvelles sémantiques de représentation et d'exécution des objets (concurrence [Watanabe 88], localisation des objets répartis [Okamura 94], envoi de messages distants [McAffer 95]).

#### 3.2 Notre approche

Notre mise en œuvre de la séparation et de l'adaptabilité dynamique des aspects est basée sur le langage réflexif NeoClasstalk [Rivard 97]. Ce dernier est le résultat de l'implémentation d'un MOP dans le langage Smalltalk [Goldberg 89]. Sa principale contribution est d'ouvrir les composantes dynamiques de Smalltalk en proposant une solution efficace pour contrôler l'envoi de messages et un protocole de changement dynamique de classe.

##### 3.2.1 Aspects spécifiques = métaclasses

La réflexion permet de distinguer *ce que* fait un objet (son niveau de base) de *comment* il le fait (son méta-niveau) [McAffer 95]. Dans un langage réflexif, il existe donc une séparation clairement définie entre les fonctionnalités de l'application, programmées au niveau de base, et leurs représentations et contrôles, programmés au méta-niveau. Le programme fonctionnel, implémenté au niveau de base, et les aspects spécifiques, implémentés au méta-niveau par des *métaclasses* (i.e. la classe d'une classe), réalisent de cette façon l'adéquation entre aspects et réflexion. Plus précisément, il existe une surjection de l'ensemble des métaclasses vers l'ensemble des aspects spécifiques, i.e. un aspect spécifique peut correspondre à une ou plusieurs métaclasses.

Dans [Ledoux 96], nous avons établi une première taxonomie des aspects pour les langages à classes où nous nous sommes intéressés aux propriétés particulières des classes. Par exemple, le

fait de ne posséder qu'une seule instance, d'interdire la création de sous-classe ou de réaliser des pré/post conditions sur les méthodes. Chacun de ces aspects a été implémenté par une ou plusieurs métaclasses. S'appuyant sur le paradigme de la séparation des aspects, nos métaclasses ont permis d'améliorer considérablement la lisibilité et la réutilisabilité des classes dans nos développements [Ledoux 98].

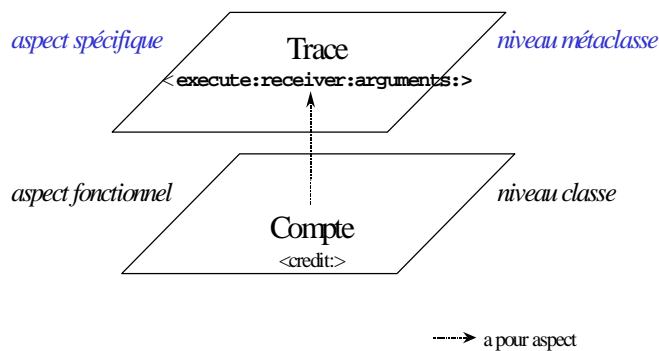


Figure 2-Aspects et (méta)classes

La Figure 2 montre l'exemple d'une classe **Compte** associée à l'aspect spécifique **Trace**, chargé de tracer les messages reçus par les instances de **Compte**. L'aspect **Trace** est implémenté par la métaclasse du même nom. Le contrôle de l'envoi de messages réifié par le MOP de NeoClasstalk – via la méthode **execute:receiver:arguments:** – va permettre à la métaclasse **Trace** d'intercepter les messages reçus par un compte. Nous présentons ci-dessous le code qui permet de tracer par exemple le message **credit:**. La méthode **execute:receiver:-arguments:** permet d'afficher le nom du message, puis d'invoquer la méthode standard d'invocation de messages grâce à l'héritage.

```
Trace>>execute: cm receiver: rec arguments: args Code aspect spécifique
  "Tracer les messages reçus"
Transcript show: 'Je passe par ', cm selector; cr.
^super execute: cm receiver: rec arguments: args
```

```
Compte>>credit: aFloat Code fonctionnel
  "Réaliser un crédit sur le solde courant"
  self solde: self solde + aFloat
```

On constate qu'il y a séparation totale des aspects puisque le code fonctionnel (i.e. réaliser un crédit) n'est pas entremêlé avec la trace. On retrouve ainsi à l'implémentation, la séparation des différents aspects obtenus à la conception.

### 3.2.2 Changement dynamique de métaclasse

Le *changement dynamique de classe* de NeoClasstalk décrit un protocole permettant aux objets de changer de classe à l'exécution [Rivard 96]. Ce protocole a pour but de tenir compte de l'évolution du comportement des objets dans le temps, et permet d'améliorer ainsi l'implémentation de leurs classes respectives.

L'association des classes comme objet de plein droit avec le protocole de changement dynamique de classe permet le *changement dynamique de métaclasse* à l'exécution. Du fait de la correspondance entre aspects et métaclasses, notre mise en œuvre autorise ainsi *l'adaptabilité dynamique des aspects*. Le protocole de changement dynamique de classe rend en effet possible l'ajout et le retrait des aspects à l'exécution sans re-génération de code.

En reprenant l'exemple précédent, on peut associer temporairement l'aspect **Trace** à une classe le temps de sa mise au point. La classe **Compte** « bascule » de sa métaclasse d'origine<sup>2</sup> vers la métaclasse **Trace** pour tracer les messages, puis revient vers son état antérieur par un nouveau basculement une fois le programme réalisé (cf. Figure 3).

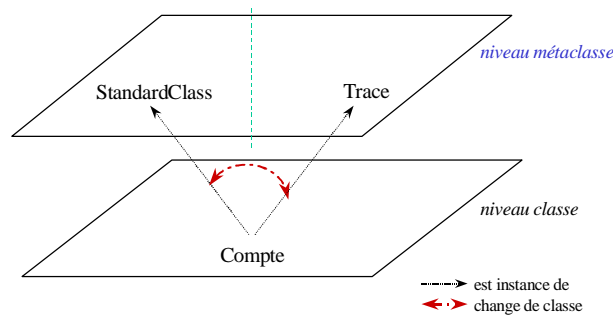


Figure 3-Adaptabilité dynamique des aspects

## 4. OPENCORBA : UN ORB OUVERT

Pour la construction d'architectures réparties ouvertes, le paradigme de la séparation des aspects permet de décorrélérer les différents mécanismes de répartition<sup>3</sup> du programme fonctionnel du système. Notre approche est particulièrement appropriée puisqu'elle autorise l'adaptabilité dynamique de ces aspects spécifiques à l'exécution. Ainsi, une invocation synchrone peut devenir asynchrone, un objet volatil peut devenir persistant, un objet *proxy* peut être migré, etc.

Notre première réalisation d'une architecture ouverte s'est déroulée dans le cadre de la plateforme CORBA [OMG 95] et a donné lieu à l'implémentation d'un bus logiciel nommé OpenCorba [Ledoux 98]. OpenCorba est un *ORB réflexif* implémentant les API CORBA dans NeoClasstalk. Il réifie différents aspects du bus CORBA afin de rendre plus « malléable » les mécanismes internes

<sup>2</sup> **StandardClass** est la racine de toutes les métaclasses de NeoClasstalk. Elle décrit le comportement standard des classes.

<sup>3</sup> et les mécanismes associés comme la coordination, la tolérance aux fautes, la sécurité, etc.

de l'ORB. OpenCorba favorise la séparation des caractéristiques du bus pour mieux les organiser, les rendre modulaires et donc interchangeable (*séparation des aspects*). OpenCorba permet ainsi d'intervenir sur le modèle d'exécution répartie de l'ORB, de façon à changer dynamiquement les stratégies de représentation et d'exécution du bus (*adaptabilité dynamique des aspects*).

Dans les paragraphes suivants, nous choisissons de présenter deux aspects du bus parmi ceux que nous avons réifiés : le mécanisme d'invocation via un *proxy*, et le contrôle de type sur la classe serveur.

#### 4.1 Projection IDL OpenCorba

La mise en œuvre de la séparation des aspects a lieu lors de l'étape de projection IDL. En respectant le *mapping* Smalltalk de la norme CORBA [OMG 95], notre pré-compilateur IDL OpenCorba génère une classe *proxy* sur le client et une classe *template* sur le serveur. La classe *proxy* est alors associée à la métaclasse **ProxyRemote** représentant l'aspect « invocation à distance » ; la classe *template*, à la métaclasse **TypeChecking** représentant l'aspect « contrôle de type sur le serveur ». La Figure 4 présente les résultats de la projection IDL de l'interface **Compte** dans OpenCorba : la classe *proxy* **CompteProxy** est instance de **ProxyRemote** et la classe *template* **Compte** est instance de **TypeChecking**.

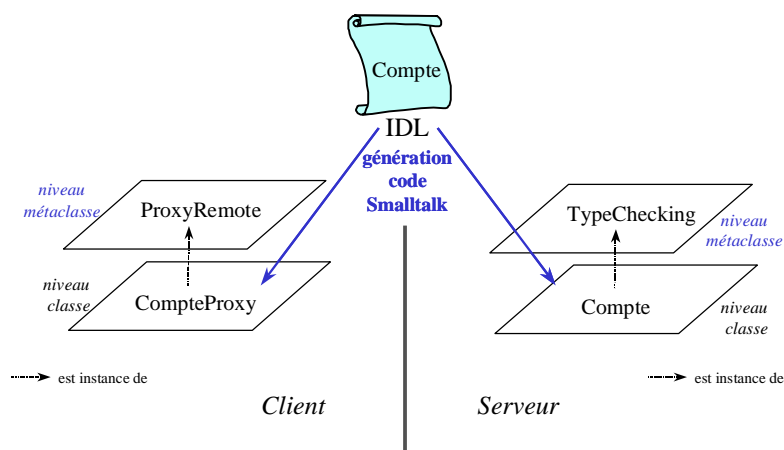


Figure 4-Projection IDL OpenCorba

#### 4.2 Proxy OpenCorba

Dans les architectures réparties, l'objet *proxy* est une représentation locale côté client de l'objet serveur. Sa mission est d'assurer la création des requêtes et leur acheminement vers le serveur, puis de retourner le résultat au client. Pour cela, une classe *proxy* adapte le style d'appel local au mécanisme d'invocation à distance [Shapiro 86].

#### 4.2.1 Séparation des aspects

L'invocation à distance est totalement indépendante de la sémantique de l'interface IDL d'origine. Aussi, l'invocation à distance se rapporte au contrôle de l'application et non au code de l'application : elle relève de la méta-programmation. La Figure 5 symbolise la séparation des aspects qui en résulte.

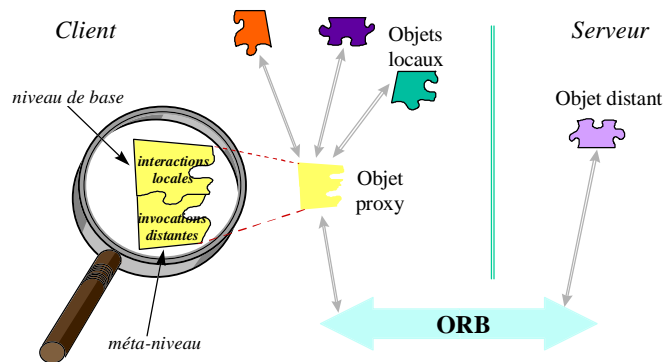


Figure 5- Représentation symbolique du proxy OpenCorba

Le pré-compilateur IDL OpenCorba génère la classe *proxy* au niveau de base de l'application. Le client peut alors manipuler la classe *proxy* avec les outils classiques Smalltalk (butineur, inspecteur). D'autre part, la classe *proxy* est associée à la métaclasse **ProxyRemote**, chargée d'invoquer l'objet réel. L'invocation à distance reste alors invisible au client. Ainsi, la séparation des aspects permet la transparence totale de l'invocation à distance pour le client.

#### Détails au niveau de base

Les méthodes de la classe *proxy* sont purement descriptives et ne présentent qu'une apparence d'interface comme les opérations IDL [Ledoux 97]. Dans le butineur Smalltalk, les méthodes ne contiennent aucun code, seulement un commentaire indiquant qu'elles ont été générées.

attribut/opération IDL	méthodes d'une classe proxy OpenCorba
<code>readonly attribute float solde;</code>	<pre>solde "Generated by OpenCORBA *** DO NOT EDIT ***" ^^aFloat "</pre>
<code>void credit(in float montant);</code>	<pre>credit: aFloat "Generated by OpenCORBA *** DO NOT EDIT ***" ^^nil"</pre>

Tableau 1-Exemples de correspondance IDL vers proxy

Le Tableau 1 présente des exemples de projection pour un attribut et une opération de l'interface IDL `Compte` vers les méthodes Smalltalk correspondantes de la classe `proxy` `CompteProxy`. Le type des valeurs retournées est mis en commentaires pour signaler les informations IDL au programmeur.

#### Détails au méta-niveau

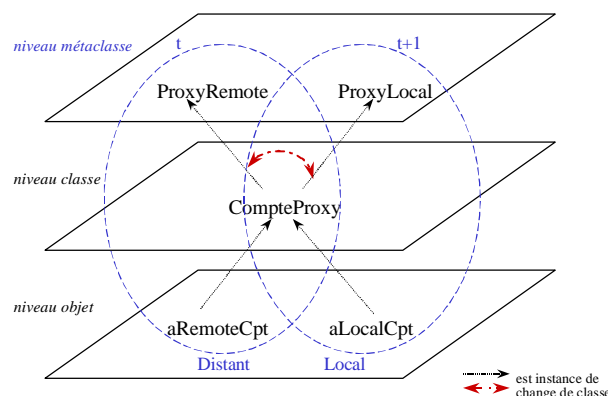
Par définition, l'exécution d'un message sur un objet `proxy` provoque une invocation sur l'objet serveur qu'il représente. L'idée est donc d'intercepter le message au moment de sa réception pour lancer une invocation à distance. Ainsi, la technique du contrôle de l'envoi de messages répond parfaitement à nos attentes. La métaclasse `ProxyRemote` redéfinit la méthode `executer:receiver:arguments:` du MOP NeoClasstalk pour intercepter les messages reçus par un `proxy`. Cette dernière réalise alors l'invocation à distance en utilisant les API du DII CORBA conformément à [OMG 95].

#### 4.2.2 Adaptabilité dynamique des aspects

Pour permettre l'adaptabilité dynamique de l'invocation dans OpenCorba, il est possible de développer d'autres aspects. Nous les avons distingués en deux catégories :

- La première traite des *variations* possibles sur la gestion des *proxies*. Nous pensons à des aspects modélisant Java RMI [SUN 97], une future version du DII CORBA ou tout simplement une invocation locale Smalltalk.

Ce dernier mécanisme a fait l'objet d'une implémentation dans OpenCorba. Nous adaptons le comportement du `proxy` suite au rapatriement de la classe serveur *en local*<sup>4</sup>. Pour cela, nous avons développé une nouvelle métaclasse, `ProxyLocal`, chargée de réaliser des invocations sur un objet serveur local. La Figure 6 montre qu'une instance d'une classe `proxy` peut effectuer des invocations distantes à l'instant  $t$ , puis des invocations locales à l'instant  $t+1$ , uniquement grâce au changement dynamique de métaclasse (i.e. *sans* modifier le code du niveau de base).



**Figure 6-Adaptabilité dynamique des proxies**

<sup>4</sup> Qui réside dans la même image Smalltalk.

- La deuxième catégorie traite des *extensions* de la notion de *proxy* introduisant de nouveaux mécanismes comme la *migration* [Jul 88, Okamura 94] ou la *réplication active* [Birman 91, Felber 97].

La migration d'objet consiste à transférer un objet serveur<sup>5</sup> vers la plate-forme cliente afin d'optimiser les performances du système réparti. Ce mécanisme allège les goulots d'étranglement du réseau et minimise les communications distantes. Le client manipule toujours un *proxy*, mais celui-ci se comporte comme un cache local de l'état de l'objet serveur.

La réplication est un autre mécanisme de gestion de la répartition des objets. Elle consiste en une duplication du serveur en plusieurs réplicas qui sont la représentation exacte du serveur original. Le mécanisme de réplication active suppose que le message est envoyé par le client aux réplicas – via le *proxy* – grâce à un protocole de diffusion atomique (*broadcast*).

En conclusion, l'adaptabilité dynamique des aspects permet d'ouvrir le mécanisme d'invocation du système réparti à de nouveaux aspects, sans bouleverser l'architecture existante.

### 4.3 Serveur et contrôle de type

Dans la norme CORBA, il est spécifié que le dépôt d'interfaces<sup>6</sup> est utilisé pour vérifier la conformité de la signature de la requête, avant et après application de la méthode par le serveur [OMG 95]. Par contre, elle ne précise pas comment doit être effectuée cette vérification de type.

#### 4.3.1 Séparation des aspects

Nous sommes convaincus qu'il est du ressort de la classe serveur de vérifier si une de ses méthodes peut être appliquée ou non. En effet, si ce genre de vérification est effectué plus en amont par l'ORB (au moment du dépaquetage de la requête par le serveur par exemple), cela introduit une gestion plus rigide puisque le moindre changement dans le mécanisme de contrôle de type entraîne une réécriture des couches basses de l'ORB. Au contraire, nous proposons d'externaliser le contrôle de ces couches pour séparer l'aspect « contrôle de type » du reste du bus logiciel. Ce dernier est délégué à la classe serveur qui va tester si l'application de sa méthode est possible ou non.

Cependant, le contrôle de type est indépendant du code défini par la classe serveur : il peut être séparé du code de l'application pour être implémenté au méta-niveau. Ainsi, le code de la classe serveur n'effectue aucun test sur le type des données passés en arguments. Le développeur peut ainsi écrire, modifier, récupérer son code sans se soucier du contrôle de type effectué par la métaclasse **TypeChecking**.

Techniquement, cette métaclasse contrôle l'envoi de messages sur la classe serveur – via la méthode **execute:receiver:arguments:** – pour interroger le dépôt d'interfaces avant et après application de ses méthodes. Le dépôt nous permet alors de vérifier le type des arguments à l'appel et le type du résultat au retour de la méthode.

---

<sup>5</sup> Ou une partie de l'objet serveur, c'est à dire soit ses méthodes, soit ses valeurs.

<sup>6</sup> Rappelons que le dépôt d'interfaces est semblable à une base de données contenant l'ensemble des interfaces IDL sous la forme d'objets accessibles à l'exécution.

En séparant les aspects, OpenCorba externalise le contrôle de type, à la fois des couches basses de l'ORB, et de la classe serveur.

#### 4.3.2 *Adaptabilité dynamique des aspects*

Grâce à notre approche, nous pouvons apporter de nouveaux mécanismes de contrôle de type, sans remettre en cause l'implémentation existante. Par exemple, nous pouvons développer une nouvelle métaclasse gérant un *système de cache* pour les types des paramètres. À la première interrogation du dépôt d'interfaces, OpenCorba sauvegarde le type de chaque argument de la méthode localement<sup>7</sup>. Puis, aux prochaines invocations de la méthode, la métaclasse interroge les informations mémorisées afin de réaliser le contrôle de type.

Comme autre exemple, nous pouvons aussi nous débarrasser du contrôle de type pour des raisons de performance ou lorsque le type des paramètres est connu. Un changement dynamique de métaclasse permettra alors d'associer la classe serveur à la métaclasse par défaut du système OpenCorba.

## 5. TRAVAUX CONNEXES ET PERSPECTIVES

### 5.1 Travaux connexes

#### 5.1.1 *Aspects*

Il existe un certain nombre de modèles et techniques permettant la séparation des aspects : les filtres de composition [Bergmans 94], la programmation adaptative [Lieberherr 96], la programmation orientée aspects (AOP) [Kiczales 97]. Cependant, ces derniers mettent en œuvre des constructions particulières pour achever la programmation par aspects. Aussi, nous avons préféré une approche réflexive qui n'impose pas un nouveau modèle, mais étend les langages existants pour les rendre ouverts. De plus, contrairement à ces modèles, notre solution rend possible l'adaptabilité des aspects à l'exécution.

#### 5.1.2 *Smart Proxies*

L'ORB Orbix de la société IONA [IONA 96] fournit la possibilité de redéfinir par héritage le comportement des talons générés, pour introduire des mécanismes de caches sur le client (*smart proxies*) [Baker 97]. Cette technique a permis une réalisation concrète d'un service de migration en CORBA. Cependant, elle utilise le lien d'héritage pour introduire l'aspect « migration » à la compilation : elle ne permet donc ni la séparation des aspects, ni leur adaptabilité dynamique.

---

<sup>7</sup> Par exemple, dans un dictionnaire partagée Smalltalk ou encore dans le code octal de la méthode compilée.

### 5.1.3 *Middleware réflexifs adaptatifs*

Parallèlement au bus OpenCorba, d'autres projets de recherche sont en cours de réalisation dans le domaine des *middleware* réflexifs adaptatifs. La principale distinction entre OpenCorba et les travaux présentés ci-dessous réside dans l'utilisation d'un modèle réflexif différent.

Le projet ADAPT de l'Université de Lancaster s'intéresse à l'implémentation de plates-formes adaptatives pour les applications multimédia mobiles [Blair 97] et propose une approche pour la conception de *middleware* ouverts [Blair 98].

Fort de son expérience dans l'élaboration de MOP [Gowing 96], l'équipe systèmes distribués du Trinity College de Dublin a récemment démarré le projet Coyote qui a pour but de fournir un support pour la construction de systèmes adaptables. Ce projet comporte de nombreux points communs avec notre thème de recherche puisqu'il s'intéresse à la programmation par aspects [Dempsey 97] et à l'adaptation dynamique du bus CORBA.

## 5.2 Perspectives

### 5.2.1 *Composition des aspects*

Les architectures réparties sont complexes et nécessitent de prendre en compte de nombreux aspects pour leur modélisation. La question de la composition de ces aspects s'impose alors inéluctablement. Par exemple, la fonctionnalité « tracer les méthodes d'une classe réalisant des invocations distantes » utilise l'aspect « trace » conjuguée avec l'aspect « invocation à distance ». Comme nous l'avons vu précédemment, chacun de ces aspects est implémenté par une ou plusieurs métaclasse. Aussi, la combinaison de plusieurs aspects soulève le problème de la composition des métaclasse : cette composition est à priori source de conflits quand il existe un chevauchement des comportements [Bouraqadi 98a].

Pour aborder ce problème, nos récents travaux ont consisté à définir un « modèle de compatibilité » de métaclasse pour offrir un cadre fiable pour la composition des métaclasse [Bouraqadi 98b]. Nos travaux actuels s'attachent à classifier les aspects pour prévenir des éventuels chevauchements de comportement.

### 5.2.2 *Élaboration d'un MOP efficace*

Dans un premier temps, les critères de performance n'ont pas été une priorité dans le développement de OpenCorba. Dans un environnement distribué, le coût engendré par une indirection vers le MOP devient modeste du fait des transmissions réseau [Chiba 93]. Efficacité et réflexion ne sont donc pas incompatibles pour les architectures réparties. Cependant, certaines implémentations des aspects du bus logiciel dans le MOP ne nécessitent pas de communication réseau [Ledoux 98]. Aussi, il est intéressant de réduire le coût engendré par un MOP. Nous sommes en cours d'investigation sur l'utilisation de l'évaluation partielle pour réduire au maximum la phase d'interprétation à l'exécution [Masuhara 98].

## 6. CONCLUSION

Dans cet article, nous avons montré que le paradigme de la séparation des aspects, étendu à l'adaptabilité dynamique des aspects, rend possible la construction d'architectures réutilisables et adaptables, i.e. *ouvertes*.

Notre approche, basée sur la réflexion, établit une correspondance entre aspects et métaclasse pour mettre en œuvre la séparation des aspects. Elle introduit alors le changement dynamique de métaclasse pour modéliser l'adaptabilité dynamique des aspects.

Ainsi, dans le contexte des environnements répartis, les langages réflexifs offrent un modèle dynamique permettant la réutilisation et l'adaptabilité des aspects liés à la répartition. Notre première expérimentation nous a permis d'ouvrir les différents mécanismes associés aux caractéristiques internes du bus logiciel CORBA. Le résultat, OpenCorba, est un ORB ouvert, capable d'adapter dynamiquement les stratégies de représentation et d'exécution du bus logiciel.

### *Remerciements*

Je tiens à remercier mes fidèles relecteurs et plus particulièrement Noury Bouraqadi, Mathias Braux, Philippe Krief, Laure-Anne Lagier, Fred Rivard et Mario Südholt.

## BIBLIOGRAPHIE

- [Baker 97] Sean Baker — *CORBA Distributed Objects Using Orbix*, chapter Smart proxies, p.307-328, Addison-Wesley, ACM Press, 1997.
- [Bergmans 94] Lodewijk Bergmans — *Composing Concurrent Objects*. PhD thesis, University of Twente, Enschede, Pays-Bas, Juin 1994.
- [Birman 91] K. Birman, A. Schiper, P. Stephenson — Lightweight Causal and Atomic Group Multicast. In *ACM Transactions on Computer Systems*, vol.9, n°3, p.272-314, 1991.
- [Blair 97] Gordon S. Blair, Geoff Coulson, Nigel Davies, Philippe Robin et Tom Fitzpatrick — Adaptive Middleware for Mobile Multimedia Applications. In *Proceedings of the 8th International Workshop on NOSSDAV'97*, St-Louis, Missouri, Mai 1997.
- [Blair 98] Gordon S. Blair, Geoff Coulson, Philippe Robin et Michael Papatomas — An Architecture for Next Generation Middleware. In *Proceedings of Middleware'98*, Septembre 1998. (*à paraître*)
- [Bouraqadi 98a] Noury Bouraqadi-Saâdani, Fred Rivard et Thomas Ledoux — Composition de métaclasse. In *Actes des Journées Francophones des Langages Applicatifs (JFLA'98)*, p.141-157, INRIA, Como, Italie, Février 1998.
- [Bouraqadi 98b] Noury Bouraqadi-Saâdani, Thomas Ledoux et Fred Rivard — Safe Metaclass Programming. In *Proceedings of OOPSLA'98*, ACM Sigplan Notices, Vancouver, Canada, Octobre 1998. (*à paraître*)
- [Chiba 93] Shigeru Chiba et Takashi Masuda — Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings of ECOOP'93*, p.482-501, LNCS 707, Springer-Verlag, Kaiserslautern, Allemagne, 1993.

- [Dempsey 97] John Dempsey et Vinny Cahill — Aspects of System Support for Distributed Computing. In *Workshop Aspect-Oriented Programming*, ECOOP'97, Jyväskylä, Finlande, Juin 1997.
- [Felber 97] Pascal Felber, Rachid Guerraoui et André Schiper — A CORBA Object Group Service. In *Workshop CORBA : Implementation, Use and Evaluation*, ECOOP'97, Jyväskylä, Finlande, Juin 1997.
- [Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides — *Design Patterns*, Addison-Wesley, Reading, Massachusetts, 1995.
- [Goldberg 89] Adele Goldberg et David Robson — *Smalltalk-80 : The Language*, Addison-Wesley, Reading, Massachusetts, 1989.
- [Gowing 96] Brendan Gowing et Vinny Cahill — Meta-Object Protocols for C++ : The Iguana Approach. In *Proceedings of Reflection'96*, Ed. Kiczales, p.137-152, San Francisco, California, Avril 1996.
- [Hürsh 95] Walter L. Hürsh and Cristina Videira Lopes — *Separation of Concerns*. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, Février 1995.
- [IONA 96] IONA Technologies — *The Orbix Architecture*. White paper. Novembre 1996.
- [Jul 88] E. Jul, H. Levy, N. Hutchinson et A. Black — Fine-grained mobility in the Emerald system. In *ACM Transactions on Computer Systems*, vol.6(1), p.109-133, Février 1988.
- [Kiczales 91] Gregor Kiczales, Jim des Rivieres et Daniel G. Bobrow — *The Art of the Metaobject Protocol*. The MIT Press, 1991.
- [Kiczales 96] Gregor Kiczales — Beyond the Black Box : Open Implementation. In *IEEE Software*, Janvier 1996.
- [Kiczales 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier et John Irwin — Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, LNCS 1241, Springer-Verlag, p.220-242, Jyväskylä, Finlande, Juin 1997.
- [Ledoux 96] Thomas Ledoux et Pierre Cointe — Explicit Metaclasses as a Tool for Improving the Design of Class Libraries. In *Proceedings of ISOTAS'96*, LNCS 1049, p.38-55, Springer-Verlag, Kanazawa, Japon, Mars 1996.
- [Ledoux 97] Thomas Ledoux — Implementing Proxy Objects in a Reflective ORB. In *Workshop CORBA : Implementation, Use and Evaluation*, ECOOP'97, Jyväskylä, Finlande, Juin 1997.
- [Ledoux 98] Thomas Ledoux — *Réflexion dans les systèmes répartis : application à CORBA et Smalltalk*. Thèse de doctorat, Université de Nantes, École des Mines de Nantes, Mars 1998.
- [Lieberherr 96] Karl J. Lieberherr — *Adaptive Object-Oriented Software : The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
- [Maes 87] Pattie Maes — Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, ACM Sigplan Notices, p.147-155, Orlando, Floride, Octobre 1987.
- [Masuhara 98] Hidehiko Masuhara et Akinori Yonezawa — Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In *Proceedings of ECOOP'98*, LNCS 1445, p.418-439, Springer-Verlag, Brussels, Belgium, Juillet 1998.
- [Matsuoka 93] Satoshi Matsuoka et Akinori Yonezawa — Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In *Proceedings of Research Directions in Concurrent Object-Oriented Programming*, ed. G. Agha, P. Wegner et A. Yonezawa, MIT Press, p.107-150, 1993.
- [McAffer 95] Jeff McAffer — Meta-level architecture support for distributed objects. In *Proceedings of IWOOS'95*, p.232-241, Lund, Sweden, 1995.
- [Okamura 94] Hideaki Okamura et Yutaka Ishikawa — Object Location Control Using Meta-level Programming. In

*Proceedings of ECOOP'94*, p.299-319, LNCS 821, Springer-Verlag, Juillet 1994.

- [OMG 95] Object Management Group — *The Common Object Request Broker : Architecture and Specification*, Revision 2.0, Juillet 1995.
- [Rivard 96] Fred Rivard — Pour un lien d'instanciation dynamique dans les langages à classes. In *Actes des Journées Francophones des Langages Applicatifs (JFLA'96)*, Val-Morin, Canada, Janvier 1996.
- [Rivard 97] Fred Rivard — *Évolution du comportement des objets dans les langages à classes réflexifs*. Thèse de doctorat, Université de Nantes, École des Mines de Nantes, Juin 1997.
- [Shapiro 86] Marc Shapiro — Structure and Encapsulation in Distributed Systems : The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, p.198-204, Cambridge, MA, Mai 1986.
- [Smith 82] Brian C. Smith — *Reflection and Semantics in a Procedural Programming Language*. PhD thesis, MIT, Janvier 1982.
- [Steyaert 96] Patrick Steyaert, Carine Lucas, Kim Mens et Theo D'Hondt — Reuse Contracts : Managing the Evolution of Reusable Assets. In *Proceedings of OOPSLA'96*, ACM Sigplan Notices, vol.31, n°10, p.268-285, San Jose, California, Octobre 1996.
- [SUN 97] Sun Microsystems — *Java RMI*. At <http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html>.
- [Watanabe 88] Takuo Watanabe et Akinori Yonezawa — Reflection in an Object-Oriented Concurrent Language. In *Proceedings of OOPSLA'88*, ACM Sigplan Notices, p.306-315, San Diego, California, Septembre 1988.



## BIOGRAPHIE

*Thomas Ledoux* est actuellement enseignant-chercheur à l'École des Mines de Nantes (EMN). Auparavant ingénieur d'études chez IBM, il a rejoint l'équipe Systèmes et Langages à Objets de l'EMN depuis 1994. Il a obtenu son doctorat en informatique en mars 1998, sur le thème de la réflexion dans les systèmes répartis. Ses travaux, illustrés par l'implémentation d'un ORB réflexif, présentent un modèle dynamique pour la construction d'architectures adaptables et ouvertes.