INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

JavaPod : une plate-forme à composants adaptable et extensible

Eric Bruneton — Michel Riveill

N° 3850

Janvier 2000

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--3850--FR

JavaPod : une plate-forme à composants adaptable et extensible

Eric Bruneton*, Michel Riveill†

Thème 1 — Réseaux et systèmes
Projet Sirac

Rapport de recherche n° 3850 — Janvier 2000 — 27 pages

Résumé : Dans le cadre de la construction d'applications réparties, nous nous intéressons aux plates-formes logicielles qui servent de support à ces applications. Ces plates-formes prennent en charge une *propriété non-fonctionnelle* : la mise en œuvre de la communication à distance entre les composants. CORBA permet de donner aux applications d'autres propriétés non-fonctionnelles (transactions, persistance, etc.) mais le programmeur doit les utiliser de façon explicite dans son application. A l'inverse, la plate-forme Enterprise Java Beans (EJB), en utilisant une certaine forme de *réflexivité*, permet de séparer complètement le code fonctionnel et les propriétés non-fonctionnelles. L'approche EJB nous semble intéressante, mais elle est encore assez limitée : en particulier, la liste des propriétés non-fonctionnelles offertes est figée. Nous proposons donc une plate-forme dont l'architecture est inspirée de l'architecture EJB, et qui est mise en œuvre grâce à un modèle original de composition d'objets implémenté par une extension de Java. Le but de ce modèle est de pouvoir offrir aux applications un ensemble de propriétés non-fonctionnelles non limité a priori, et également de pouvoir composer facilement les différentes propriétés.

Mots-clés : applications distribuées, composants, plate-forme logicielle, propriétés fonctionnelles et non-fonctionnelles, composition de propriétés non-fonctionnelles, réflexivité, aspects.

* doctorant dans le projet SIRAC, financé par le CNET (France Telecom)
† professeur à l'Institut National Polytechnique de Grenoble

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

JavaPod : an Adaptable and Extensible Component Platform

Abstract: This work deals with the middleware platforms that are used to support distributed, component-based applications. All middleware platforms offer at least one *non-functional* property : the implementation of the remote communication between components. CORBA offers other non-functional properties (transactions, persistence, etc.) but the application programmer must use them explicitly. On the contrary, the Enterprise Java Beans (EJB) platform, by using a kind of *reflective* mechanism, allows for a complete separation of the functional and non-functional aspects. This approach is promising but is still limited: in particular, the set of provided non-functional properties is small and not extensible. We therefore propose a middleware platform whose architecture is derived from the EJB architecture, and which relies on a new object composition model implemented by an extension of Java. The goal of this model is to enable the definition of a potentially unlimited set of non-functional properties, and to facilitate the composition of these properties.

Key-words: distributed applications, components, middleware platform, functional and non-functional properties, composition of non-functional properties, reflectivity, aspects.

1 Introduction

1.1 Cadre de travail

Notre travail se place dans le cadre de la construction d'applications réparties à base de composants. Un *composant* possède des propriétés analogues à celles d'un objet (encapsulation de code et de données, séparation entre interface et réalisation), mais constitue en plus une *unité de composition* autonome, notamment parce que ses dépendances sont rendues explicites par l'utilisation d'*interfaces requises*. Les composants, ainsi que les concepts associés de *connecteurs* (objet de communication entre composants) et de description d'architecture logicielle sont des outils très utiles pour construire des applications réparties.

Le développement d'une application répartie ne se fait pratiquement jamais à partir de “zéro”, c'est à dire directement au dessus d'un système d'exploitation et de ses primitives de communications de bas niveau comme les *sockets*. On préfère utiliser une plate-forme logicielle (ou *middleware*), située entre le système d'exploitation et l'application, et qui fournit des abstractions de haut niveau définissant un modèle de programmation pour les applications. Il existe plusieurs types de modèles de programmation, les principaux étant le modèle client-serveur, basé sur l'appel de méthode à distance synchrone, et le modèle publication-abonnement, basé sur la diffusion d'événements (messages asynchrones). Java RMI, CORBA, EJB et DCOM sont des exemples de plates-formes “industrielles” qui offrent un modèle de programmation de type client-serveur. iBus est une plate-forme basée sur la diffusion d'événements, qui utilise la “norme” JMS (*Java Message Service*), une sorte d'équivalent de CORBA pour les modèles à événements.

On désigne sous le terme de *propriétés fonctionnelles* d'une application les services qu'elle fournit, généralement décrits par des spécifications d'interfaces. Les *propriétés non-fonctionnelles* désignent les autres caractéristiques de l'application, liées à la façon dont sont implémentées les propriétés fonctionnelles : performances, fiabilité, disponibilité, qualité de service, etc. Conceptuellement, les propriétés fonctionnelles et non-fonctionnelles concernent des aspects différents d'une application et sont donc indépendantes. On souhaite donc les séparer au niveau du code de l'application, pour permettre une plus grande facilité de réutilisation à la fois du code fonctionnel et du code non-fonctionnel.

Cette séparation est malheureusement assez difficile à obtenir, ce qui fait que les propriétés non-fonctionnelles sont la plupart du temps “noyées” dans le code des applications. Le code qui implémente la communication à distance entre les composants est assez facile à séparer du code fonctionnel, et toutes les plates-formes logicielles permettent cette séparation (il suffit en effet d'utiliser les techniques classiques du *remote procedure call*). Les autres propriétés non-fonctionnelles sont plus complexes à séparer. Un début de solution consiste à les regrouper dans des objets ou composants fournissant des “services systèmes”. C'est par exemple l'approche de CORBA, qui définit plusieurs *Corba Object Services* : persistance, transactions, nommage, etc. Mais cette solution n'est que partielle, car le programmeur doit utiliser ces services de façon explicite dans son code. On peut en fait aller plus loin en rendant l'utilisation de ces services complètement implicite. C'est le cas des EJB, où les transactions, la persistance et la sécurité sont spécifiés de façon déclarative, en dehors du code des composants.

Les EJB permettent cette séparation complète grâce à l'emploi d'une certaine forme de *réflexivité* (cf section 4.1). En effet chaque composant EJB est encapsulé dans un *conteneur* qui intercepte toutes les communications du composant avec l'extérieur. Avant de déléguer un appel entrant au composant, le conteneur réalise les traitements associés aux propriétés non-fonctionnelles de ce composant : vérification des droits d'accès, lancement, le cas échéant, d'une nouvelle transaction, sauvegarde éventuelle de l'état du composant, etc. L'un des avantages de cette séparation est que les conteneurs et les composants peuvent être programmés par des personnes différentes : les conteneurs par des spécialistes des aspects “systèmes”, et les composants par des spécialistes du domaine d'application visé.

1.2 Motivations et objectifs

Dans le cadre présenté ci-dessus, notre travail se situe au niveau des mécanismes pour la séparation des propriétés fonctionnelles et non-fonctionnelles, qui ne sont pas encore assez satisfaisants. L'utilisation de la réflexivité est en effet intéressante (elle apporte une solution uniforme et générique pouvant s'appliquer à tout type de propriétés non-fonctionnelles) mais sa mise en œuvre dans les plates-formes “industrielles” (essentiellement les EJB) est limitée : seules deux ou trois propriétés non-fonctionnelles sont actuellement offertes, et on ne peut ni supprimer, ni modifier, ni ajouter de propriétés dans cette liste prédéfinie, qui ne saurait suffire aux besoins très variés des applications.

Notre travail vise à s'affranchir de ces limites, tout en conservant les idées de base de l'approche (autrement dit la notion de conteneur transmettant ses propriétés aux composants). Plus précisement, l'objectif de notre travail est de définir une plate-forme à composants qui permette d'associer aux composants, de façon transparente, les propriétés non-fonctionnelles requises par chaque application, choisies parmi une liste de propriétés ouverte et extensible. Nous proposons pour cela de construire la plate-forme logicielle et les conteneurs à partir d'un noyau minimal, par assemblage d'extensions de ce noyau, chaque extension apportant une propriété non-fonctionnelle.

Nous souhaitons également que la plate-forme logicielle n'impose pas un modèle de programmation trop spécifique et, en particulier, qu'elle autorise la construction de n'importe quel type de connecteur. Enfin, comme pour les composants, on souhaite pouvoir associer des propriétés non-fonctionnelles aux connecteurs : compression, cryptographie, etc.

La section 2 présente l'architecture de la plate-forme à composants que nous proposons, et que nous appelons la plate-forme *JavaPod*. Cette section est décomposée en deux sous sections, qui présentent l'architecture du noyau, puis le modèle de composition utilisé pour composer entre elles les extensions du noyau. La section 3 présente l'implémentation du modèle de composition, puis celle du noyau. La section 4 contient une introduction rapide sur la réflexivité dans les langages de programmation, puis présente les travaux apparentés au nôtre : composition de méta-objets ou d'aspects, et plates-formes logicielles extensibles. La section 5 conclut ce document.

2 Architecture de la plate-forme

Comme on l'a dit dans l'introduction, la plate-forme à composants JavaPod doit permettre, comme les EJB, d'associer des propriétés non-fonctionnelles aux composants. De plus, l'ensemble des propriétés non-fonctionnelles offertes ne doit pas être limité.

Pour satisfaire le premier objectif, la plate-forme à composants JavaPod utilise une architecture inspirée de celle des EJB, qui reprend la notion de conteneur. Cependant, une telle architecture ne suffit pas à elle seule pour satisfaire le second objectif, comme le montrent les implémentations actuelles de l'architecture EJB, qui n'offrent qu'un nombre figé de propriétés non-fonctionnelles. Cela est essentiellement du au fait que les conteneurs EJB sont monolithiques. Pour résoudre le problème, nous proposons de construire les conteneurs[1] par composition de “composants”. Pour éviter la confusion avec les composants de l'application, nous appellons ces “composants” des *extensions*. Cette approche permet de construire des conteneurs de façon modulaire, chaque extension du conteneur pouvant conférer une propriété non-fonctionnelle spécifique au composant. Pour faciliter cette composition, nous avons introduit un modèle de composition original, inspiré de la réflexivité par méta-objets (cf section 4.1.2) et de l'héritage de classes.

Le reste de cette section présente les différents éléments de l'architecture de la plate-forme JavaPod (section 2.1), puis le modèle de composition (section 2.2).

2.1 Éléments d'architecture

Comme on l'a indiqué dans l'introduction, chaque plate-forme logicielle offre un modèle de programmation d'applications. Nous avons choisi pour notre plate-forme une version extrêmement simplifiée[2] du modèle computationel d'[ODP] (*Open Distributed Processing*), car c'est un modèle abstrait très général, dont les modèles client-serveur et publication-abonnement ne sont que des instances particulières. Selon ce modèle très simplifié, une application se décompose en objets, chaque objet ayant une ou plusieurs interfaces d'accès. Les *objets de liaison* sont des objets spécialisés dans les communications entre les autres objets. Ils ont la particularité d'être constitués de plusieurs morceaux pouvant être répartis sur des sites différents. Un objet de liaison peut connecter un nombre arbitraire d'interfaces, qui peuvent avoir des types différents. Un tel objet peut fournir n'importe quel type de liaison : synchrone, asynchrone, flot de données, etc.

Le modèle applicatif précédent n'est pas encore défini rigoureusement, car ce n'est pas notre objectif principal : nous nous intéressons en premier lieu à la plate-forme logicielle qui permet d'offrir ce genre de modèle de programmation aux applications. Dans l'architecture JavaPod, une telle plate-forme est constituée d'un certain nombre d'éléments qui sont la mise en œuvre, au niveau de la plate-forme logicielle, des concepts du modèle de programmation. La figure 1 présente ces différents éléments, ainsi que les concepts du modèle dont ils dérivent. Comme le montre la figure, une plate-forme à composants est constituée de quatre types d'éléments[3], qui ont une existence concrète à l'exécution : des serveurs, des conteneurs, ainsi que des talons et des squelettes, qui, regroupés, forment les connecteurs. Ces éléments sont inspirés du modèle EJB, mais ils sont également proches

1. mais aussi les autres éléments de l'architecture : serveurs, talons et squelettes. Voir la section 2.1.
2. le modèle de référence ODP est très détaillé et très précis. Nous n'avons retenu que les idées générales.
3. les composants font partie de l'application : ils ne sont donc pas comptés comme un élément de l'architecture.

de certains concepts du modèle d'*ingénierie* d'ODP (qui est un modèle de mise en œuvre de modèle computationel). Ainsi, un serveur correspond à un serveur EJB, et à une capsule ODP. Un conteneur correspond à un conteneur EJB et, plus ou moins, à un *cluster* ODP. Les talons et les squelettes sont également présents, sous des noms parfois différents, dans les EJB, CORBA, RMI, ou ODP.

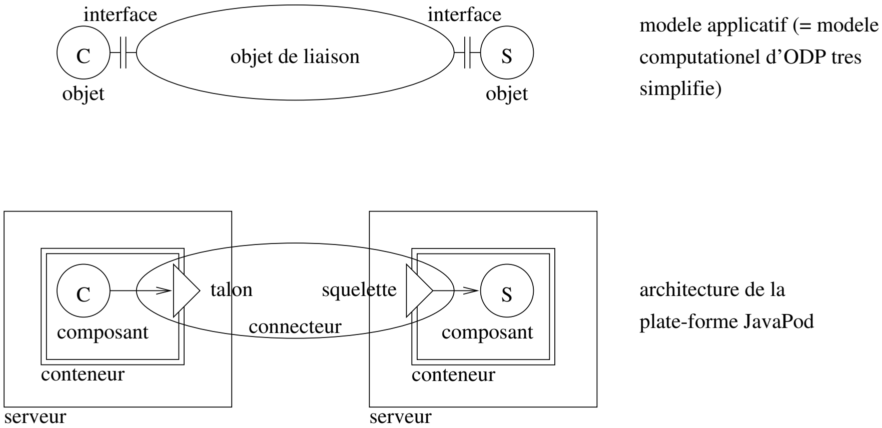

FIG. 1 – *les éléments d'architecture de la plate-forme JavaPod*

Le découpage en serveurs, conteneurs, talons et squelettes n'a donc rien de très original en soi. Par contre, le rôle que nous attribuons à ces éléments est plus étendu que dans les plates-formes classiques. Ces rôles sont détaillés dans le reste de cette section.

Note : dans la suite de ce document, lorsque nous parlons d'un conteneur, nous désignons non seulement le conteneur au sens strict, mais aussi les talons et les squelettes qu'il *contient*. Ainsi, nous disons qu'un conteneur intercepte les communications du composant encapsulé bien que l'interception soit réalisée par un talon ou un squelette (qui peut ensuite interagir avec le conteneur au sens strict).

2.1.1 Serveur

Un serveur est une structure d'accueil, un support d'exécution pour les conteneurs, qui sont eux même une structure d'accueil pour composants. Un serveur fournit tous les services systèmes dont peuvent avoir besoin les conteneurs : protocoles de communication, service de persistance, gestion de ressources, etc. Il peut également offrir une interface de surveillance et d'administration des applications.

2.1.2 Composant et conteneur

Un composant désigne, comme dans ODP, un objet pouvant avoir plusieurs interfaces d'accès. Cependant, nous n'utilisons pas le terme "objet" pour éviter tout risque de confusion avec la notion habituelle d'objet, celle des langages de programmation. En effet, un objet ODP peut être représenté par un ou plusieurs objet(s) d'un langage objet, chaque interface d'accès pouvant donner accès à un objet différent. C'est pourquoi nous parlons plutôt de composant.

Le modèle ODP permet de définir des objets composites. La plate-forme JavaPod n'interdit pas de voir, au niveau applicatif, un groupe de composants comme un composant. Cependant, au niveau de la plate-forme, nous ne considérons que des composants non composites, pour simplifier (a priori, associer des propriétés non-fonctionnelles à un groupe de composants semble assez difficile). Un composant peut donc être constitué d'un ou plusieurs objets, mais doit rester non distribué (il doit être contenu en entier dans un seul serveur).

Un conteneur est la partie système correspondant à un composant, de la même façon qu'il existe une structure système associée à un processus dans un système d'exploitation, ou à un objet dans une machine virtuelle Java. Le conteneur d'un composant "encapsule" le composant, au sens ou toutes les interactions du composant avec l'extérieur doivent normalement passer par le conteneur. Grâce à cette interposition, le conteneur peut gérer les propriétés non-fonctionnelles du composant : modèle d'exécution, de persistance, de synchronisation, etc. Comme nous l'avons déja dit, un conteneur *contient* des "portes", c'est à dire des talons et des squelettes qui permettent au composant de communiquer avec l'extérieur.

2.1.3 Connecteur, talon et squelette

Un connecteur est un objet distribué, qui relie deux composants ou plus entre eux. Plus exactement, un connecteur contient et relie entre eux un certain nombre de talons et de squelettes situés dans des conteneurs différents. Un connecteur désigne exactement la même chose qu'un objet de liaison ODP.

Un talon ou un squelette est à l'interface entre un conteneur et un connecteur, et appartient aux deux à la fois. Un talon permet à un composant d'envoyer des messages ou des données vers l'extérieur, alors qu'un squelette permet d'en recevoir. Chaque talon ou squelette a un type, qui est un ensemble de signatures de méthodes. Ce type définit les méthodes que le composant peut appeler sur un talon, ou qu'un squelette peut appeler dans un composant.

Dans les plates-formes classiques, les talons et les squelettes ont un rôle assez limité. Dans la plate-forme JavaPod, ces objets implémentent quasiment toutes les fonctions d'un connecteur (ils peuvent cependant s'appuyer sur des protocoles de bas niveau fournis par le serveur). Les extensions qui constituent ces objets correspondent à peu près aux éléments d'une pile de protocoles. Certaines extensions fournissent des fonctions de base, d'autres peuvent apporter des propriétés non-fonctionnelles au connecteur : adaptation d'interface, compression, cryptographie, etc. Enfin, du fait que les talons et les squelettes appartiennent également à un conteneur, certaines extensions peuvent prendre en charge une partie des propriétés non-fonctionnelles des composants (voir la section 2.2.2 pour un exemple).

2.1.4 Référence de connecteur

La plate-forme JavaPod n'utilise pas de références d'objet comme dans CORBA, ni de références d'interface comme dans ODP, mais des références de connecteur. Une référence de connecteur a un double rôle :

- elle identifie un unique connecteur, comme toute référence.
- elle décrit les extensions à assembler pour construire un nouveau talon ou un nouveau squelette pour ce connecteur, en fonction de l'interface d'accès souhaitée. Ce faisant, elle décrit en quelque sorte la nature (client-serveur, diffusion de groupe, etc.) et les propriétés non-fonctionnelles (qui se rapprochent ici de la notion de qualité de service) de ce connecteur.

Le choix de réferencer les connecteurs plutôt que les composants peut paraître étrange. En effet, sans référence d'objet ou de composant, il est impossible de référencer explicitement un composant (mais on peut le référencer implicitement : la référence d'un connecteur client-serveur désigne implicitement le composant serveur). Ce choix est en fait motivé par les raisons suivantes. Premièrement, les références d'objets à la CORBA ou à la RMI sont trop restrictives, car elles imposent des connecteurs de type essentiellement client-serveur.

Deuxièmement, bien que les références d'interfaces d'ODP n'imposent pas de contraintes sur le type des connecteurs, elles sont moins souples que la notion que l'on propose. En effet, une référence d'interface ne contient pas directement une description du connecteur auquel elle appartient : elle contient à la place l'identité de l'*usine à liaisons* qui a servi à fabriquer cette référence. Une usine à liaisons, comme son nom l'indique, est chargée de construire des connecteurs. Elle contient une description des protocoles à utiliser pour instancier un connecteur. Avec ce modèle, pour associer différentes propriétés non-fonctionnelles à un connecteur (compression, cryptographie, fiabilité, etc.), il faudrait une usine à liaisons par combinaison possible. Avec notre approche, qui consiste à décrire le connecteur directement dans sa référence et non pas indirectement via une usine à liaisons, ce problème disparaît (mais les références sont plus complexes et plus volumineuses).

2.1.5 Exemples de connecteurs

Comme un objet de liaison ODP, un connecteur peut représenter n'importe quel type de liaison : client-serveur, publication-abonnement, flots de données, diffusion de groupe, espace de tuples, etc.

Un connecteur client-serveur contient un squelette serveur et n talons clients. La référence d'un tel connecteur contient l'identité du squelette, ainsi qu'une description des extensions à assembler pour construire un nouveau talon client.

Un connecteur publication-abonnement contient un talon (pour l'émetteur), et n squelettes (un par abonné). La référence d'un tel connecteur contient l'identité du talon, ainsi qu'une description des extensions à assembler pour construire un nouveau squelette abonné. En réalité, un tel connecteur utilise deux interfaces d'accès : une interface fonctionnelle, pour publier et recevoir des événements, et une interface de contrôle, pour s'abonner et se désabonner. Un tel connecteur contient donc en réalité un talon émetteur, n talons de contrôle et n squelettes de réception (et sa référence contient également la description des extensions à assembler pour implémenter l'interface de contrôle).

Un connecteur peut également contenir d'autres objets que des talons et des squelettes JavaPod. Par exemple, il peut connecter des composants à un objet serveur CORBA. La référence d'un tel connecteur contient la référence CORBA de l'objet serveur, et une description des extensions à assembler pour connecter un talon JavaPod à ce connecteur. Inversement, un connecteur peut relier des clients CORBA à un composant JavaPod.

2.2 Modèle de composition

Comme nous l'avons mentionné plus haut, le but du modèle de composition est de pouvoir construire les éléments de l'architecture précédente de façon modulaire, par un assemblage d'extensions, afin de pouvoir offrir aux applications un ensemble de propriétés non-fonctionnelles non limité a priori.

Le modèle de composition que nous proposons est en fait très général, car il peut s'appliquer à tout ce qui peut être vu comme un objet. Autrement dit, c'est un modèle de composition d'objets. Il pourrait donc servir, a priori, en dehors du cadre de la plate-forme JavaPod. Pour notre part, nous l'utilisons uniquement pour la programmation de la plate-forme elle-même, pas pour les applications (mais ces dernières devraient en profiter, indirectement, puisque ce modèle devrait permettre de leur associer de façon modulaire des propriétés non-fonctionnelles).

Après une présentation rapide du modèle, cette section montre ses avantages, pour ce que l'on veut en faire, par rapport aux modèles classiques de composition : héritage, délégation, piles de protocoles.

2.2.1 Définition

Le modèle de composition permet de composer des objets, au sens général du terme (c'est à dire une entité qui encapsule un état interne, accessible uniquement par des méthodes). Le résultat de la composition d'un certain nombre d'objets est appelé un *objet composite*. Les objets internes d'un objet composite sont appelés ses *constituants*.

Ces constituants sont totalement ordonnés. Nous désignons par *objet extensible* le constituant le plus "petit" pour cet ordre. Les autres constituants sont ce que l'on a appelé précédemment des *extensions*. L'objet extensible est invariable, alors que les extensions peuvent être ajoutées, supprimées ou remplacées dynamiquement [4]. Nous représentons en général les constituants les uns au dessus des autres, l'objet extensible étant à la base, comme sur la figure 2.

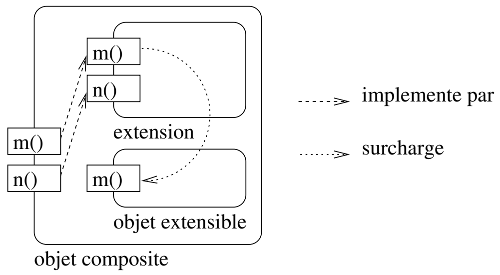

FIG. 2 – *un exemple d'objet composite*

Un objet composite, comme son nom l'indique, est également un objet : ses méthodes sont l'union des méthodes de ses constituants, et son état interne est l'union des états internes de ses constituants. La sémantique d'un appel de méthode sur un objet composite est la suivante :

- si la méthode appelée n'est définie dans aucun constituant, l'appel échoue,
- sinon, l'appel est exécuté par le constituant le plus "haut" (ou encore, par rapport à l'ordre total précédent, le plus "grand") qui définit cette méthode.

4. cette distinction n'est pas fondamentale, et on pourrait la supprimer au niveau modèle. Elle vient en fait de la plate-forme JavaPod, qui est construite à partir d'un noyau minimal, invariable, une sorte de *framework* de plate-forme à composants.

Autrement dit, si une méthode est définie dans plusieurs constituants, le constituant le plus haut masque, ou surcharge, les définitions des autres constituants. Nous supposons cependant qu'il est possible, dans le code d'une extension, d'appeler la méthode surchargée par un appel spécial. La sémantique d'un tel appel est alors la même que précédemment, mais en se limitant aux constituants situés (strictement) en dessous de l'extension appelante.

On considère que l'état interne d'un constituant n'est accessible que par ses méthodes. Autrement dit, nous interdisons les attributs accessibles directement de l'extérieur. C'est pourquoi le modèle ne dit rien sur une éventuelle composition d'attributs. Dans un composite, chaque constituant a accès à son état interne, mais pas à celui des autres constituants.

Remarques :

- Une extension ne peut pas faire partie de plusieurs objets composites à la fois.
- Une extension peut exister en dehors d'un objet composite, mais n'est alors pas utilisable : un appel de méthode sur une telle extension échouera.
- Pour des raisons de cohérence, il est nécessaire que la modification dynamique des extensions d'un composite ne puisse pas intervenir pendant un appel de méthode sur ce composite.

2.2.2 Évaluation

Cette section montre, *dans le cadre de la plate-forme JavaPod*, les avantages du modèle de composition précédent par rapport aux techniques classiques de composition et/ou de réutilisation : délégation, héritage, piles de protocoles. Il ne s'agit donc pas d'une comparaison complète, qui indiquerait pour chaque modèle de composition le contexte dans lequel il est le plus adapté.

La discussion qui suit s'appuie sur le scénario suivant. On se place dans un cadre client-serveur, et on considère deux propriétés non-fonctionnelles [5] : protection des composants par capacités cachées, et duplication des composants avec maintien de la cohérence des copies.

La protection par capacités cachées [Hagimont] serait trop longue à présenter ici. Ce qui importe est que sa mise en œuvre requiert d'intercaler deux filtres FC et FS entre un client et un serveur (fig. 3). Un filtre client (comme FC) est essentiellement chargé de fixer les droits que le client souhaite accorder sur les objets qu'il passe en référence aux autres composants. Pour cela, un tel filtre installe des filtres serveur (comme FS') sur chaque objet *exporté* (i.e. passé en référence) par le composant client. Un filtre serveur (comme FS ou FS') est essentiellement chargé de filtrer les appels reçus pour rejeter ceux qui ne sont pas autorisés par la politique de protection (en réalité, chaque filtre est à la fois client et serveur, car on peut passer des objets en référence non seulement lors d'un appel, mais aussi lors du retour du résultat).

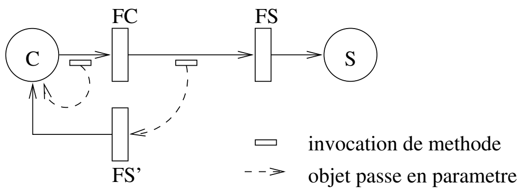

FIG. 3 – *protection par capacités cachées*

La duplication des composants consiste, lors d'un appel de méthode sur un composant "dupliqué", à amener une copie de ce composant dans le serveur de composants du client pour optimiser les appels ultérieurs qui se feront en local. Pour maintenir la cohérence des copies, on peut utiliser un gestionnaire centralisé, chargé de distribuer des copies et d'invalider celles qui ne sont plus à jour.

Au niveau applicatif, ces deux propriétés non-fonctionnelles concernent des composants. Au niveau de la plate-forme, leur mise en œuvre concerne *tous* les éléments de l'architecture. En effet, les filtres viennent se placer au niveau des talons et des squelettes. De même, pour intercepter un appel sur un composant "dupliqué", il faut se placer au niveau d'un talon. Le gestionnaire des copies se place au niveau d'un serveur de composants.

5. ce scénario est très simplifié. En réalité, nous souhaitons offrir beaucoup plus de propriétés non-fonctionnelles.

Enfin, les informations sur la politique de protection, ou l'information qui indique qu'un composant est "dupliqué" sont naturellement associées à un conteneur.

Comment peut-on composer les deux propriétés non-fonctionnelles précédentes au niveau d'un talon ? Avec notre modèle de composition, on peut utiliser des extensions ayant les interfaces suivantes (en pseudo Java) :

- objet extensible T
  - `Object invoke (MethodInvocation)` : cette méthode est appelée par chaque méthode de l'interface applicative du talon. Par défaut, elle ne fait rien. Cet objet extensible est par conséquent une abstraction de talon, qui se contente de définir un cadre de travail à compléter par des extensions.
- extension A
  - `Message send (Message, Address)` : envoie un message à l'adresse indiquée et retourne un message de réponse.
- extension B
  - `Object invoke (MethodInvocation)` : surcharge la méthode correspondante de T pour en fournir une implémentation concrète. Cette méthode réalise tous les traitements classique d'un RPC. Elle utilise pour cela la méthode `send`, et les méthodes suivantes :
    - `MethodInvocation export (MethodInvocation)` : remplace chaque objet passé en référence dans l'invocation de méthode par la référence d'un squelette donnant accès à cet objet.
    - `MethodInvocation import (MethodInvocation)` : remplace chaque référence reçue en un talon connecté au squelette correspondant à la référence.
- extension C
  - `Object invoke (MethodInvocation)` : teste s'il existe une copie du composant serveur en local et en ramène une au besoin en s'adressant au gestionnaire des copies. Appelle ensuite la méthode surchargée pour réaliser l'invocation de méthode proprement dite (on suppose qu'elle utilise un mécanisme optimisé lorsque le composant serveur est disponible localement).
- extension D
  - `MethodInvocation export (MethodInvocation)` : appelle la méthode correspondante de B, puis ajoute, le cas échéant, des extensions de filtrage sur les squelettes retournés par la méthode surchargée.
  - `MethodInvocation import (MethodInvocation)` : de la même façon, ajoute le cas échéant une extension de filtrage sur les talons retournés par la méthode surchargée.

Avec cette organisation, on peut facilement construire un talon réalisant ou non du filtrage pour la protection, et ramenant ou non une copie du composant serveur en local : il suffit de créer des instances des extensions voulues, puis de les réunir dans l'ordre adéquat dans un objet composite. Toutes les "connexions" entre les extensions se font automatiquement, grâce à la sémantique spéciale de notre modèle de composition.

Si l'on veut faire la même chose avec de l'héritage de classes, on peut garder essentiellement la même organisation en termes d'interfaces. Par contre, on est obligé de définir une classe par combinaison possible (fig. 4) : TAB, TABC héritant de TAB, TABD héritant de TAB, et TABCD héritant de TABC. Cela devient infaisable si on envisage plus d'extensions et/ou plusieurs implémentations d'une même extension. De plus, une fois créé, on ne peut pas changer la composition d'un talon : il faudrait pouvoir changer sa classe dynamiquement.

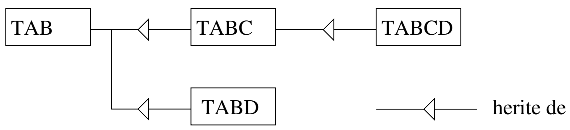

FIG. 4 – *composition par héritage*

On peut par contre utiliser un modèle de délégation (fig. 5), toujours en gardant plus ou moins le même découpage, mais cela reste moins simple et moins naturel qu'avec notre modèle. Par exemple, il n'y a pas de moyen simple pour composer B et D de façon à ce que D surcharge les méthodes de B. Le seul moyen est de déléguer, dans B, les méthodes `export` et `import` à un objet séparé B1. On pourrait alors intercaler D entre B et B1. Autre problème : T délègue la méthode `invoke` à B, mais il masque les autres méthodes (`import`, `export`, `send`, etc.). Pour résoudre le problème, il faudrait que T (et B également) implémente lui aussi ces méthodes, par délégation. Ce qui augmenterait fortement les dépendances entre les extensions.

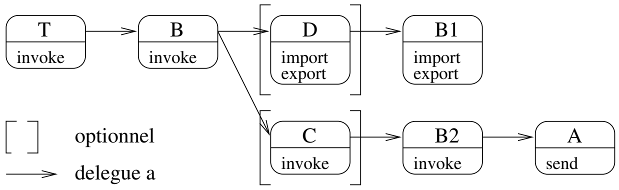

FIG. 5 – *composition par délégation*

On peut également penser à un modèle de composition du type “pile de protocoles”. Ce modèle permet en effet de composer des protocoles de façon relativement souple. Cependant, ce n'est rien d'autre qu'un modèle par délégation, avec des contraintes supplémentaires (une couche ne peut communiquer qu'avec la couche immédiatement inférieure).

Conclusion

Bien que l'architecture de la plate-forme présente quelques aspects originaux (par exemple les références de connecteurs), nous avons choisi dans ce document de mettre l'accent sur l'aspect qui nous semble le plus original, à savoir le modèle de composition utilisé pour mettre en œuvre cette architecture. Comme l'a montré cette section, ce modèle est plus adapté (pour notre plate-forme) que les modèles classiques. Il est également plus adapté et plus efficace que les modèles réflexifs basés sur des méta-objets, dont il est pourtant très proche (voir les sections 4.1 et 4.2 plus loin dans ce document).

Ce modèle devrait faciliter la composition de propriétés non-fonctionnelles, mais il ne résout évidemment pas tous les problèmes. En particulier, le point le plus délicat, qui se pose également avec tous les autres modèles de composition, est de trouver les bonnes interfaces pour les extensions. Par exemple, si l'extension B n'exposait pas les méthodes `import` et `export`, D serait obligée de réimplementer complètement B. En fait, il faut exposer les méthodes qui peuvent être utiles pour d'autres extensions, mais il ne faut pas non plus exposer trop de méthodes, sinon les performances finiraient par se dégrader fortement.

3 Mise en œuvre

Le modèle de composition présenté dans la partie précédente a été intégré dans un nouveau langage, appelé ejava car c'est une extension de Java. L'architecture de la plate-forme à composants JavaPod a été implémenté, en ejava, sous forme d'un noyau minimal fournissant une classe d'objet extensible par élément de l'architecture.

3.1 Le langage ejava

Le langage ejava est conçu pour pouvoir définir des objets composites, au sens du modèle présenté dans la section 2.2. Il aurait été possible de se passer de ce langage : on peut en effet, au prix de certaines contraintes de programmation, obtenir des objets composites en Java “pur”. C'est d'ailleurs ce que nous avons fait dans une première phase de ce travail. Cependant, les conventions de programmation étaient vraiment lourdes, et cela conduisait à fixer dans le code une stratégie de mise en œuvre du modèle de composition. Avec un langage, on peut facilement changer cette stratégie : il suffit de modifier le compilateur.

3.1.1 Présentation

Les principales caractéristiques de ejava sont les suivantes :

– ejava est un sur-ensemble de Java : tout programme Java est aussi un programme ejava, ayant exactement la même sémantique.

– seuls les objets programmés avec les extensions du langage peuvent servir de constituant d'un objet composite.

– ejava a exactement la même syntaxe que Java, mais une sémantique légèrement différente pour les objets composites (reconnus grâce à l'utilisation de classes et d'interfaces spéciales).

La première chose que définit le langage ejava est une correspondance entre les concepts du modèle, et ceux de Java. Un objet extensible est représenté par un objet Java dont la classe hérite de la classe prédéfinie `ExtensibleObject`. De même, une extension est représentée par un objet Java dont la classe hérite de la classe prédéfinie `Extension`. Enfin, un objet composite n'a pas de correspondance concrète en Java : on ne peut donc pas référencer l'objet composite lui même, mais seulement ses constituants. Pour appeler une méthode d'un objet composite, il suffit d'appeler cette méthode sur l'un de ses constituants (le résultat ne dépend pas du constituant utilisé). Toutefois, en pratique, il vaut mieux référencer et utiliser un objet composite en utilisant son objet extensible, qui est sa seule partie invariable.

Le reste de cette section montre de façon intuitive, à l'aide d'un exemple, comment ejava peut s'utiliser. La définition précise du langage est donnée dans la section suivante. La figure 6 montre comment programmer en ejava l'objet extensible et l'extension de la figure 2 (p. 7), et comment on peut les utiliser.

```
public class C extends ExtensibleObject {
  private int i;
  public C (int i) {
    this.i = i;
  }
  public void m () {
    ++i;
  }
}
```

```
public final class D extends Extension {
  public void m () {
    System.out.println("m called");
    super.m();
  }
  public void n () {
    System.out.println("n called");
  }
}
```

`C c = new C(0); D d = new D();`	*résultat de l'exécution du code ci-contre :*
`c.m();`	
`// ((D)c).n();`	`// NoSuchEMethodException`
`c.setExtensions(new Extension[] {d});`	
`c.m();       // equivalent a d.m();`	`m called`
`((D)c).n();  // equivalent a d.n();`	`n called`

FIG. 6 – *un exemple de programme en ejava*

Cette exemple montre qu'il est possible d'ajouter dynamiquement une extension à un objet extensible, ce qui permet de surcharger des méthodes (l'appel à la méthode `m` après l'ajout de l'extension n'a pas le même effet qu'avant), et d'ajouter des méthodes (la méthode `n` est définie uniquement après l'ajout de l'extension).

Comme nous l'avons indiqué ci-dessus, un objet composite peut être utilisé en utilisant n'importe lequel de ses constituants. Dans l'exemple, une fois que `c` et `d` sont réunis, on peut utiliser `c` ou `d` pour appeler `m` ou `n` (bien que l'utilisation de `c` soit préférable). Pour appeler une méthode qui n'est pas définie dans le constituant que l'on utilise, il faut une conversion de type (*cast* en anglais) généralisée. Par exemple, pour appeler `n` en utilisant `c`, il faut écrire `((D)c).n()` car `n` n'est pas définie dans `C` mais dans `D` (le but de ce *cast* est de pouvoir faire un minimum de vérifications de type à la compilation).

Comme on peut le voir dans l'exemple de la figure 6, la sémantique de ejava est légèrement différente de celle de Java. Par exemple, l'appel à `super.m()` dans la classe `D` ne serait pas correct en Java, car `D` n'est pas une sous-classe de `C`. Il est par contre correct en ejava, car la sémantique de ce genre d'appel est étendue pour les classes héritant de `Extension`. De même, l'expression `((D)c).n()` ne serait pas correcte en Java, pour la même raison. Elle est par contre correcte en ejava car la sémantique du *cast* est étendue dans ce genre de cas.

3.1.2 Définition

La correspondance entre les concepts du modèle de composition et ceux de Java, mentionnée précédemment, doit ici être précisée. L'état interne des constituants est représenté par des attributs Java, qui doivent être privés (cf la discussion sur les attributs des constituants dans la section 2.2.1). Les méthodes d'un constituant sont les méthodes publiques non statiques de sa classe, exceptés les méthodes hérités des classes prédéfinies de Java ou de ejava (`Object`, `Extension`, `ExtensibleObject`, etc.). Ce qui signifie que la sémantique spéciale de l'appel de méthode définie dans le modèle ne s'applique qu'à ces méthodes, que l'on appelle des *eméthodes*. Un appel à `toString`, à `setExtensions`, à une méthode statique, ou à une méthode privée d'un constituant suit donc la sémantique usuelle de Java.

Classes prédéfinies

Le langage ejava utilise 5 classes et interfaces prédéfinies, réunies dans le paquetage `ejava` :

- La classe `ExtensibleObject` dénote les classes d'objets extensibles. Cette classe contient les méthodes suivantes :
  - `Extension[] getExtensions()` : retourne la liste ordonnée des extensions de cet objet.
  - `void setExtensions(Extension[] extensions)` : modifie la liste des extensions de cet objet. Le tableau est supposé être ordonné comme dans la méthode précédente.

  Chaque classe héritant de `ExtensibleObject` doit être qualifiée uniquement par `[public|protected] [final]` (en ordre quelconque), doit avoir uniquement des attributs `private`, et doit implémenter uniquement des interfaces héritant de `EInterface`. De plus, chaque méthode non statique doit être qualifiée uniquement par `[private|public] [synchronized]`, et chaque eméthode ne doit lancer que des exceptions héritant de `EException`, de `RuntimeException` ou de `Error`.
- La classe `Extension` dénote les classes d'extension. Cette classe contient la méthode suivante :
  - `ExtensibleObject getExtensibleObject()` : retourne l'objet extensible auquel est attachée cette extension, ou `null`.

  Chaque classe héritant de `Extension` doit respecter les mêmes contraintes que précédemment, et doit de plus être déclarée `final`.
- L'interface `EInterface` dénote les interfaces que peuvent implémenter les objets extensibles et les extensions. Toutes les super-interfaces d'une interface qui hérite (directement ou non) de `EInterface` doivent elles-mêmes hériter (directement ou non) de `EInterface`. Les classes n'héritant ni de `ExtensibleObject` ni de `Extension` ne doivent implémenter que des interfaces n'héritant pas de `EInterface`. Ces règles ont pour but de pouvoir distinguer statiquement un objet normal d'un constituant d'un objet composite.
- La classe `EException` dénote les exceptions que peuvent lancer les eméthodes. Bien que cette classe hérite de `RuntimeException`, elle doit être capturée ou déclarée comme une exception normale.
- Enfin, la classe `NoSuchEMethodException`, sous-classe de `RuntimeException`, est l'exception qui est lancée lorsqu'un appel de méthode sur un objet composite échoue (cf la section 2.2).

Syntaxe et sémantique

La syntaxe de ejava est celle de Java. La sémantique de Java est modifiée dans les trois cas suivants :

- *cast* vers une classe d'objet extensible ou d'extension, ou vers une interface héritant de `EInterface`, *uniquement lorsque celui-ci apparait juste avant un appel d'eméthode*. Dans ce cas, le type statique de l'objet sur lequel porte le *cast* doit être soit une super-classe soit une sous-classe de `ExtensibleObject`, `Extension` ou `EInterface` (sinon c'est une erreur de compilation). A l'exécution, ce genre de *cast* ne donne lieu à aucune vérification (la vérification a lieu lors de l'appel d'eméthode qui suit).
- appel d'une eméthode. Dans ce cas, l'appel de méthode doit satisfaire à toutes les contraintes statiques normales (en particulier, la méthode doit être définie dans le type statique de l'objet appelé, le type statique des paramètres doit être conforme à la signature de la méthode, etc.). Par contre, à l'exécution, la méthode appelée est traitée par l'objet composite référencé conformément à la sémantique définie dans le modèle.
- appel à `super` dans une méthode d'une classe héritant de `Extension`. La méthode appelée doit être une eméthode de la classe courante, et le type statique des paramètres doit être conforme à la signature de la méthode. A l'exécution, la méthode appelée est traitée par l'objet composite courant conformément à la sémantique du modèle.

Dans tous les autres cas, c'est la sémantique normale de Java qui s'applique. En particulier, les mots clefs `this` et `synchronized` et l'opérateur `instanceof` gardent leur sémantique habituelle. Ce qui signifie que cette sémantique n'est pas étendue pour faire apparaître tous les constituants d'un composite comme un seul et même objet. Chaque constituant garde son identité, sa classe, etc.

La méthode `setExtensions`, déclarée `final` afin de ne pas pouvoir être surchargée, est mutuellement exclusive avec les eméthodes et avec la méthode `getExtensions`. Cela assure que la liste des extensions ne peut pas être modifiée pendant que l'objet composite est en cours d'utilisation par une eméthode. Par contre, les eméthodes et les autres méthodes de l'objet composite peuvent a priori être exécutées en parallèle.

Chaque extension est attachée à un objet extensible au plus. Si on tente d'attacher une extension à un objet extensible alors qu'elle est déja attachée à un autre objet extensible, une exception est lancée par la méthode `setExtensions`. Enfin, un appel d'eméthode sur une extension qui n'est pas attachée à un objet extensible retourne une exception.

3.1.3 Implémentation

Le langage ejava peut être mis en œuvre de différentes façons. La solution la plus simple et la plus portable consiste à traduire les programmes ejava en programmes Java équivalents, ce qui permet de les exécuter sur n'importe quelle machine virtuelle Java. Une autre solution est de modifier la machine virtuelle Java, ce qui permet d'être plus efficace a priori. Pour des raisons de portabilité, nous avons choisi la première solution.

La traduction de ejava vers Java peut se faire de deux façons :

- traduction des fichiers sources ejava en Java, puis compilation avec javac des fichiers traduits.
- compilation directe des fichiers sources ejava avec un compilateur spécial.

Notre mise en œuvre permet en fait de faire les deux. Nous avons en effet modifié un compilateur Java écrit en Java, et nous avons implémenté la traduction par des transformations au niveau de l'arbre de syntaxe abstraite. A partir de cet arbre, le compilateur d'origine est capable de produire soit le code correspondant, soit les sources Java correspondantes.

Le principe de la traduction consiste à transformer tous les appels à des eméthodes en un appel à une méthode générique `invoke`, qui prend en argument un nom de méthode et un tableau d'objets contenant les paramètres de l'appel. Cette méthode, définie dans une super-classe des classes `ExtensibleObject` et `Extension`, consulte la table de méthodes associée à l'objet composite appelé pour savoir à quelle extension ou objet extensible il faut confier chaque appel. Cette table de méthodes est recalculée à chaque fois que la liste des extensions est modifiée. Il existe également une table de méthodes par extension, qui indique pour chaque méthode surchargée dans cette extension l'objet extensible ou l'extension à qui il faut confier les appels à `super`. La figure 7 montre les tables de méthodes correspondant à l'exemple de la figure 2.

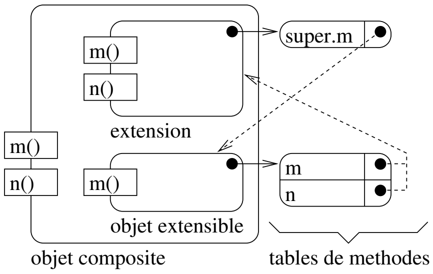

FIG. 7 – *tables de méthodes correspondant à un objet composite*

En plus de la traduction des appels d'eméthodes, les *casts* généralisés sont supprimés, et enfin les classes héritant de `ExtensibleObject` et de `Extension` sont transformées pour réunir toutes les eméthodes dans une seule méthode générique `handle`, appelée par la méthode `invoke`[6]. La figure 8 montre une traduction possible en Java de l'exemple de la figure 6 (p. 11).

3.2 Le noyau JavaPod

Le noyau JavaPod est l'implémentation des concepts de l'architecture présentée dans la section 2. Il ne fournit aucune fonctionnalité, et se contente de définir un cadre de travail pouvant servir à la définition d'extensions du noyau. Ce noyau est programmé en ejava.

6. la méthode `handle` n'est pas indispensable, on pourrait utiliser l'introspection de Java, mais ce serait beaucoup plus lent à l'exécution.

<pre>public class C extends ExtensibleObject {
  private int i;
  public C (int i) {
    this.i = i;
  }
  public void m () {
    throw new IllegalAccessError();
  }
  protected Object handle (String m,
                           Object[] args) {
    if (m.equals("m()")) {
      ++i;
    }
    return null;
  }
}</pre>
	<pre>public final class D extends Extension {
	  public void m () {
	    throw new IllegalAccessError();
	  }
	  public void n () {
	    throw new IllegalAccessError();
	  }
	  protected Object handle (String m,
	                           Object[] args) {
	    if (m.equals("m()")) {
	      System.out.println("m called");
	      invokeSuper(m,args);
	    } else if (m.equals("n()")) {
	      System.out.println("n called");
	    }
	    return null;
	  }
	}</pre>

<pre>C c = new C(0);
c.invoke("m()",null);
// c.invoke("n()",null);
c.setExtensions(new Extension[] {new D()});
c.invoke("m()",null);
c.invoke("n()",null);</pre>	

FIG. 8 – *une traduction possible de ejava en Java*

3.2.1 Les classes du noyau

Le noyau JavaPod définit les classes et interfaces suivantes : `Server`, `Container`, `Component`, `Stub`, `Skeleton`, `Reference`. Ces classes correspondent aux concepts de même nom dans le modèle (le concept de connecteur n'est pas représenté par une classe, car un connecteur est un objet distribué abstrait, qui n'a pas d'existence concrète, sous forme d'un seul objet, à l'exécution).

Server

La classe `Server` se contente de définir les méthodes suivantes :

- une méthode statique `init`, semblable à la méthode `ORB.init` dans les implémentations classiques de CORBA. Cette méthode prend en argument une liste d'extensions, et retourne une instance de la classe `Server` munie de ces extensions. Cette méthode ne peut être appelée qu'une seule fois, ce qui implique qu'il y a au plus un serveur par machine virtuelle Java.
- une méthode statique `getServer`, qui retourne une référence sur le serveur local (c'est à dire l'objet retourné par la méthode `init` précédente).
- une méthode `createContainer`, qui permet de créer un conteneur pour un composant, à partir d'un composant et d'une liste d'extensions.
- une méthode `getContainer`, qui permet de retrouver le conteneur d'un composant donné.

Seules les deux premières méthodes sont destinées au programmeur d'application. Les deux autres sont réservées à la programmation des extensions.

Component

La classe `Component` est encore plus simple. Elle ne contient qu'un attribut privé, qui pointe sur le conteneur de ce composant. Elle définit également un constructeur qui prend en arguments une liste d'extensions. Ce constructeur appelle la méthode `createContainer` de la classe `Server` pour allouer un conteneur pour le composant, et stocke la référence obtenue dans l'attribut précédent. La classe `Component` n'hérite pas de `ExtensibleObject`, contrairement aux autres classes.

Remarque : la correspondance entre le concept de composant et les objets de Java est la suivante. Un composant est représenté par un ensemble d'objets Java, dont un et un seul est une instance de la classe

`Component`. Cet objet contient un lien vers le conteneur du composant, et peut être utilisé par le conteneur lorsqu'il a besoin de *callbacks* fournis par l'application : dans ce cas, cet objet doit implémenter une ou plusieurs interfaces non fonctionnelles (par exemple `Serializable`, `Moveable`, etc.).

Stub et Skeleton

Les classes `Stub` et `Skeleton` sont quasiment identiques. Elles contiennent les méthodes `getContainer` et `getInterface` qui retournent le conteneur et l'interface du talon ou du squelette considéré. La méthode `getReference` est plus intéressante : elle retourne la référence du connecteur auquel appartient le talon ou le squelette. Par défaut, cette méthode retourne la référence qui a été utilisée lors de la création. Elle fonctionne donc uniquement pour les talons ou les squelettes qui ont été créés à partir d'un connecteur existant. Pour les autres, cette méthode doit être surchargée par les extensions, afin de retourner une description du nouveau connecteur, créé en même temps que le talon ou le squelette.

La classe `Skeleton` contient une méthode spécifique `getObject`, qui retourne l'objet interne du composant qui est relié à ce squelette. C'est l'objet qui est appelé par le squelette lorsqu'il doit convertir des messages ou des données venus de l'extérieur en appel de méthode sur le composant. Cet objet doit donc implémenter l'interface du squelette.

La classe `Stub` contient également une méthode spécifique `invoke`, qui est appelée, via une indirection, lorsque le composant appelle une méthode de l'interface du talon. Chaque méthode de cette interface redirige en effet les appels vers cette méthode générique, qui prend en argument un objet `Method`, et un tableau d'objets représentant les arguments. Cette méthode est vide. Elle doit donc être surchargée par les extensions pour que le talon soit utilisable.

Reference

L'interface `Reference` correspond à la notion de référence de connecteur. Cette interface contient les méthodes `getSkeletonExtensions` et `getStubExtensions` qui retournent une liste d'extensions en fonction d'une interface d'accès. Ces deux méthodes permettent de spécifier les extensions à assembler pour construire un nouveau talon ou un nouveau squelette pour un connecteur, en fonction du sens d'accès et de l'interface d'accès. Indirectement, les extensions retournées peuvent encapsuler les informations nécessaires pour identifier le connecteur.

Container

La classe `Container` est la plus complexe. Elle contient un lien vers un objet `Component`, accessible avec la méthode `getComponent`. Elle contient également 11 autres méthodes pour gérer les listes de talons et de squelettes de ce conteneur :

- Les méthodes `getStubs` et `getSkeletons` permettent d'obtenir ces listes.
- Les méthodes `addStub`, `addSkeleton` et `removeSkeleton` permettent aux extensions, en surchargeant ces méthodes, d'être averties lorsque des changements interviennent dans ces listes[7].
- Les méthodes `createSkeleton` et `createStub` permettent de créer un squelette ou un talon qui créent en même temps un nouveau connecteur.
- Les méthodes `attachSkeleton` et `attachStub` permettent de créer un squelette ou un talon pour se connecter à un connecteur déja existant.
- Enfin, les méthodes `importReference` et `exportReference` utilisent les méthodes précédentes pour gérer l'importation et l'exportation de références de connecteur.

Les quatre dernières méthodes réutilisent les talons et les squelettes déja existants afin de ne pas dupliquer inutilement ces derniers (pour un composant, un connecteur, et une interface d'accès donnés, il existe au plus un talon et au plus un squelette). Par exemple, la méthode `attachStub(itf,ref)` retourne un talon qui implémente l'interface `itf` et appartenant au connecteur `ref`. Si un tel talon existe déja, il est réutilisé.

La méthode `importReference(itf,ref)` retourne un objet utilisable par l'application correspondant à l'interface `itf` du connecteur `ref`. Cette méthode cherche tout d'abord un talon d'interface `itf` et appartenant

7. il n'y a pas de méthode `removeStub` : le ramasse-miettes s'en charge.

au connecteur `ref`. Si elle en trouve un, elle le retourne en résultat. Sinon, elle cherche ensuite un squelette d'interface `itf` appartenant au connecteur. Si elle en trouve un, elle retourne l'objet interne correspondant (cf `getObject` dans `Skeleton`). Sinon, elle créé un talon avec la méthode `attachStub` et le retourne.

La méthode `exportReference(itf,obj)` retourne une référence sur un connecteur correspondant à `obj`. Si `obj` est un talon, sa référence (donnée par la méthode `getReference`) est retournée. Sinon, cette méthode cherche un squelette d'interface `itf` donnant accès à `obj`. Si elle en trouve un, elle retourne sa référence. Sinon, elle crée un nouveau squelette avec la méthode `createSkeleton` et retourne sa référence.

3.2.2 Génération des talons

L'utilisation d'un bus logiciel va de pair avec celle d'un compilateur de talons et de squelettes. C'est par exemple le cas avec CORBA, RMI, ou le modèle EJB. Dans le noyau JavaPod, comme dans RMI 1.2, seul un compilateur de talons est nécessaire. Il est en effet possible, grâce aux possiblités d'introspection de Java, de définir une seule classe de squelette, capable d'invoquer n'importe quelle méthode sur n'importe quel objet.

Le noyau JavaPod inclut donc un compilateur de talons. Ce compilateur est générique, au sens ou les talons générés peuvent servir pour n'importe quel connecteur. Cela vient du fait que ces talons ne contiennent aucune fonctionnalité : celles-ci sont définies dans les extensions.

Ce compilateur a la particularité d'être invoqué automatiquement, à la dernière minute, lorsque le système à besoin d'une classe de talon et qu'il ne la trouve pas dans les classes existantes. Cela permet au programmeur de ne pas avoir à utiliser manuellement ce compilateur, et lui permet ainsi une plus grande transparence vis-à-vis de la distribution. D'un autre coté, rien n'empêche le programmeur d'utiliser manuellement ce compilateur : dans ce cas, les classes nécessaires étant présentes à l'exécution, le compilateur ne sera plus invoqué dynamiquement.

Pour être le plus rapide possible lorsqu'il est invoqué dynamiquement, ce compilateur ne procède pas de la façon classique, en générant des sources et en les faisant compiler par un compilateur Java. Il génère en fait directement le byte-code que l'on obtiendrait avec la méthode précédente. Cela lui permet d'être 100 fois plus rapide (de l'ordre de 10ms au lieu d'une ou deux secondes).

3.2.3 Exemple

Cette section montre comment programmer une application client-serveur très simple avec le noyau JavaPod. Comme en CORBA ou en RMI, il faut d'abord définir l'interface du composant serveur (fig. 9). Cette interface doit hériter d'une interface prédéfinie (ce n'est pas imposé par le noyau, mais par les extensions implémentant les connecteurs client-serveur). Nous avons utilisé ici l'interface `Remote` de RMI.

```
public interface Hello extends Remote {
  void hello (String msg) throws RemoteException;
}
```

FIG. 9 – *interface du composant serveur*

Il faut ensuite définir le code du serveur (fig. 10). Le serveur est un composant, il doit donc hériter de la classe `Component` définie dans le noyau. Il doit également implémenter l'interface (fonctionnelle) du serveur. Enfin, le lancement du serveur se fait dans la méthode `main` : il faut initialiser le serveur de composants, puis créer le composant serveur. Pour cela, il faut spécifier les extensions appropriées (non précisées ici, car cette partie dépend des extensions du noyau). Enfin, il faut stocker la référence du serveur quelque part, par exemple dans un serveur de noms (là encore, cette partie dépend des extensions du noyau).

La programmation du client est similaire (fig. 10). En particulier, c'est un composant, et il doit donc hériter de `Component`. C'est là une différence, minime, avec RMI ou CORBA. La création et le lancement du client est similaire au cas du serveur.

Remarque : dans l'exemple ci-dessus, le programmeur doit associer lui-même les extensions nécessaires aux serveurs et aux conteneurs. On pourrait facilement masquer cela, en définissant une sous classe de `Component`, que l'on pourrait appeler `UnicastRemoteObject`, comme dans RMI. On pourrait donc définir une "personnalité" RMI, comme dans [Jonathan]. On pourrait également, plutôt que de coder en dur les extensions à utiliser, utiliser la ligne de commande ou un fichier de configuration pour pouvoir lancer l'application avec différentes propriétés non fonctionnelles, sans toucher au code.

```
public class HelloServer extends Component implements Hello {
  public HelloServer (Extension[] extensions) {
    super(extensions);
  }
  public void hello (String msg) {
    System.out.println(msg);
  }
  public static void main (String[] args) {
    Server.init(...); // initialise le serveur
    HelloServer server = new HelloServer(...); // cree le composant serveur
    // ...
    // cree un connecteur (reduit pour le moment a un squelette), et stocke sa
    // reference quelque part (fichier...). Ces traitements peuvent etre
    // realises par un appel du style registry.bind("helloServer",server);
    System.out.println("HelloServer ready.");
  }
}
```

```
public class HelloClient extends Component {
  private static Hello server;
  public HelloClient (Extension[] extensions) {
    super(extensions);
  }
  public void run () {
    server.hello("Hello World");
  }
  public static void main (String[] args) {
    Server.init(...); // initialise le serveur
    HelloClient client = new HelloClient(...); // cree le composant client
    // recupere la reference du serveur, et attache un talon au connecteur ayant
    // cette reference. Stocke ce talon dans 'server'. Ces traitements peuvent etre
    // realises par un appel du style server = registry.lookup("helloServer");
    client.run();
  }
}
```

FIG. 10 – *programmation du serveur et du client*

Établissement de la liaison

Le connecteur entre le client et le serveur peut être créé de la façon suivante :

- création du squelette : coté serveur, l'appel à `bind` (ou à une méthode équivalente), finit par appeler la méthode `createSkeleton` du conteneur pour créer un squelette donnant accès à l'objet serveur. A ce stade, on ne spécifie aucune extension à associer au squelette. Par contre, l'appel à `createSkeleton` est intercepté par une extension du conteneur (spécifiée lors du `new HelloServer(...)`) qui contient une configuration pour le composant. Cette configuration indique que lorsque l'on crée un squelette pour l'interface `Hello`, il faut lui associer telle et telle extension, afin d'obtenir un connecteur client-serveur. Grâce à cette interception, les paramètres de l'appel à `createSkeleton` sont modifiés pour spécifier les extensions adéquates. Le résultat de l'appel à `createSkeleton` est donc finalement un squelette qui contient les extensions implémentant un connecteur client-serveur, plus une extension E qui surcharge la méthode `getReference` du squelette, et dont le rôle apparaît dans la suite.

- récupération de la référence du squelette : la méthode `getReference` du squelette est appellée (par la méthode `bind`). Sans l'extension E précédente, cet appel retournerait `null`. Grâce à E qui surcharge cette méthode, on obtient en résultat un objet `Reference` qui contient l'identité du squelette, ainsi que la liste des extensions à assembler pour obtenir un talon capable de commmuniquer avec le squelette. Cette référence est ensuite enregistrée quelque part (serveur de noms, fichier, etc.).

- création du talon : coté client, l'appel à la méthode `lookup` (ou à une méthode équivalente), finit par appeler la méthode `attachStub` du conteneur pour attacher un talon au connecteur dont on aura préalablement récupéré la référence. La méthode `attachStub`, non surchargée, construit les extensions à associer au talon en appelant la méthode `getStubExtensions` de l'objet `Reference`. Elle construit ensuite un talon et lui associe ces extensions.

3.3 Premiers résultats

L'implémentation du noyau JavaPod s'est faite progressivement. Par exemple, l'introduction du langage ejava et de son compilateur est assez récente, de même que celle du compilateur dynamique de talons. Auparavant, il fallait tout programmer à la main dans un style similaire à celui de la figure 8 (p. 14). Dans l'une de ces anciennes versions, nous avons programmé une dizaine d'extensions, qui permettaient d'offrir des fonctionnalités similaires à celles de RMI : appel de méthode à distance, téléchargement dynamique de code, serveur de noms. Ces extensions permettaient également de définir des connecteurs de type flots de données, et de faire migrer des composants.

Les performances de l'ORB (*Object Request Broker*) ainsi obtenu étaient quasiment égales à celles de RMI. Cela signifie que l'utilisation de notre modèle de composition a un impact négligeable sur les performances, par rapport au coût d'un envoi de message sur le réseau.

Cette première expérience nous permet également de penser que notre approche va permettre de construire une plate-forme à composants modulaire et extensible. En effet, on disposait déja de deux types de connecteurs (client-serveur et flots de données) ainsi que de la migration des composants. Toutefois, la décomposition en extensions n'était pas assez fine, pas assez modulaire, et il n'y avait pas assez d'extensions différentes pour valider vraiment notre approche. Notre objectif est donc maintenant de construire, au dessus de la version du noyau JavaPod décrite dans ce document, une plate-forme à composants vraiment modulaire et extensible, et de l'expérimenter sur des applications représentatives.

4 Travaux apparentés

Les travaux apparentés à l'architecture JavaPod et à son modèle de composition peuvent se classer en deux catégories : ceux qui concernent la réflexivité, et ceux qui ont pour but de construire des systèmes ou des ORBs adaptables et extensibles. L'architecture JavaPod ustilise en effet une certaine forme de réflexivité, et ce à deux niveaux : dans l'architecture (un conteneur peut être vu comme un méta-objet pour son composant) et dans le modèle de composition (une extension intercepte certains appels de méthode, comme un méta-objet). La première section passe en revue les différents types de langages réflexifs, et la seconde présente différents systèmes ou ORB extensibles.

4.1 Langages réflexifs

Un système est dit réflexif s'il possède une représentation de lui même qui lui permet de s'auto-décrire et de s'auto-modifier. Dans le domaine des langages de programmation, on dit qu'un langage est réflexif lorsqu'une partie au moins de ses caractéristiques (sémantique, abstractions de base, etc.) sont *réifiées*, c'est à dire rendues concrètes et manipulables par des méta-programmes [8].

Les langages orientés objets sont un cadre privilégié pour définir des langages réflexifs. On peut classer ces langages en deux catégories : ceux qui utilisent des méta-classes et ceux qui utilisent des méta-objets. Les premiers sont plutôt utilisés pour définir des extensions du langage. Ils essaient de réifier tous les aspects du langage, et pour cette raison sont assez complexes à mettre en œuvre et entraînent un surcoût important à l'exécution. Les seconds sont moins "élégants" conceptuellement, et n'essayent pas de rendre la sémantique du langage entièrement modifiable. Ils sont plutôt utilisés comme un moyen pratique de séparer les aspects fonctionnels et non fonctionnels (synchronisation, persistance, etc.) des objets. lls sont en général assez simple à implémenter et entraînent un surcoût relativement faible à l'exécution.

4.1.1 Langages réflexifs basés sur des méta-classes

La plupart des langages objets distinguent seulement deux niveaux : les objets et les classes. De plus, seuls les objets ont une existence concrète à l'exécution. On peut en fait rendre concrètes les classes, sous forme d'objets manipulables par le programme, et définir une infinité de niveaux supérieurs : les méta-classes, les méta-méta-classes, etc. C'est d'ailleurs cette façon de faire qui est la plus "propre", c'est à dire la plus uniforme.

La classe `Class` en Java est un bon exemple pour comprendre ce que représente une méta-classe. Par exemple, elle définit la méthode `newInstance`, et donc la façon dont sont créés les objets. Elle définit également des méthodes pour obtenir les attributs et les méthodes d'une classe et, indirectement (via les classes `Method` et

8. le terme *méta* est utilisé pour différencier des entités de même nature mais situées à des niveaux d'abstraction différents. Par exemple, l'en-tête d'un mail est une information (format, date, expéditeur, etc.) sur une information (le contenu du message), donc une méta-information. De même, un interpréteur est un programme qui exécute des programmes, donc un méta-programme.

`Field`), la sémantique d'un appel de méthode (méthode `invoke` de la classe `Method`) ou de l'accès à un attribut sur un objet. En bref, la classe `Class` définit la sémantique du modèle objet de Java.

L'idée des langages réflexifs basés sur les méta-classes est d'offrir la possibilité de définir d'autres méta-classes que la méta-classe standard. Cela permet en effet de définir une nouvelle sémantique pour les appels de méthode ou les créations d'instance. Par exemple, on pourrait définir une méta-classe `MonitorClass`, héritant de `Class`, et redéfinissant la méthode `invoke` pour tracer tous les appels de méthodes. Selon les langages, un objet peut ou non changer sa méta-classe dynamiquement.

Les langages CommonLISP (avec son modèle objet CLOS - pour CommonLISP Object System), [TOS] (basé sur Tcl) et ClassTalk (SmallTalk) sont des exemples de tels langages réflexifs.

4.1.2 Langages réflexifs basés sur des méta-objets

Les langages réflexifs basés sur des méta-objets ne font pas intervenir de méta-classes : ils se contentent des objets et des classes. Ils permettent cependant de définir des objets capables de modifier la sémantique d'autres objets, et qui pour cette raison sont appelés des méta-objets (les objets "normaux" sont appelés des *objets de base*).

Tous ces langages permettent généralement d'étendre la sémantique des méthodes d'un objet grâce à un méta-objet. Pour cela, lorsqu'un appel de méthode a lieu sur un objet qui possède un méta-objet, l'appel est intercepté, réifié, et remplacé par un appel à une méthode générique du méta-objet, qui prend en argument l'objet représentant l'appel initial (fig. 11).

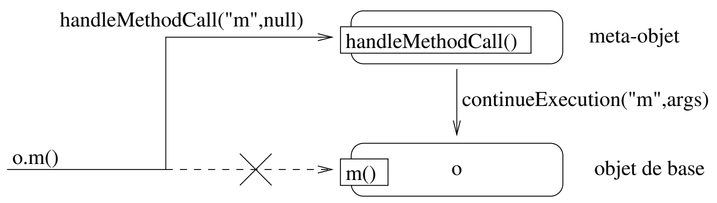

Fig. 11 – *interception d'un appel par un méta-objet*

Le méta-objet peut alors faire le traitement souhaité : il peut traiter lui même l'appel, ou le déléguer tel quel à l'objet de base grâce à un appel à une méthode spéciale appelée selon les langages `reflect`, `continueExecution`, etc. Le méta-objet peut aussi transformer le nom de la méthode ou la valeur des arguments, réaliser des pré et post traitements, etc. La figure 12 montre un méta-objet (écrit dans un pseudo Java réflexif) qui permet de tracer les appels de méthode sur un objet de base (sauf pour la méthode `toString` afin d'éviter une boucle infinie).

```
class MonitorMetaObject extends MetaObject {
  public Object handleMethodCall (String method, Object[] args) {
    if (!method.equals("toString()")) {
      System.out.println(getBaseObject().toString()+"."+method+" : before call");
      Object result = continueExecution(method,args);
      System.out.println(getBaseObject().toString()+"."+method+" : after call");
      return result;
    } else {
      return continueExecution(method,args);
    }
  }
}
```

Fig. 12 – *un méta-objet pour tracer les appels à un objet de base*

On peut également imaginer des exemples plus complexes. Par exemple, un méta-objet peut réordonner les appels sur l'objet de base pour gérer la synchronisation de ces appels. Un méta-objet peut également gérer la persistance d'un objet : il suffit d'enregistrer l'état de l'objet de base après chaque appel.

Les langages [Iguana](basé sur C++) et Actalk (SmallTalk), ainsi que les langages [Dalang], [Guarana] et [metaXa] (basés sur Java) sont des exemples de langages réflexifs utilisant des méta-objets. Cette classe de langages est relativement simple à mettre en œuvre. Par exemple, pour intercepter les appels de méthode sur un objet afin de les détourner vers un méta-objet lorsqu'il y en a un, il suffit de modifier le code original à l'aide d'un pré-processeur.

4.2 Modèles de composition

Cette section présente des modèles de composition moins classiques que l'héritage ou la délégation et les compare avec notre propre modèle.

4.2.1 Composition de méta-objets

Il existe plusieurs techniques pour composer des méta-objets. La première consiste à utiliser des méta-méta-objets. En effet, un méta-objet est un objet comme un autre, et peut donc posséder un méta-objet. Cette technique est cependant assez couteûse, car il faut réifier l'appel à chaque changement de niveau. De plus, au niveau méta-méta, on devrait normalement modifier le comportement du méta-objet, pas celui de l'objet de base. Cette technique n'est donc pas adaptée pour composer plusieurs méta-objets venant modifier chacun un aspect différent de l'objet de base.

La seconde technique consiste à composer plusieurs méta-objets de même niveau. Cette composition peut être *orthogonale* ou non. Dans le cas d'une composition orthogonale, chaque méta-objet s'occupe d'un aspect différent et totalement indépendant des autres. Ces aspects peuvent soit être figés (appel de méthode, accès à un champ, etc.), comme dans [Iguana], ou défini par le programmeur (pour les appels sur la méthode `m`, utiliser tel méta-objet, pour les appels sur `n`, tel autre, etc.). On peut ainsi associer plusieurs méta-objets à un objet de base, mais au plus un par aspect.

Certains langages permettent au contraire de composer plusieurs méta-objets pour un même aspect. Il faut alors spécifier un ordre sur ces méta-objets, le plus simple étant de les organiser en une liste, comme dans [A-TOS, Dalang, metaXa]. Avec une telle organisation, il faut également que lorsqu'un méta-objet décide de confier un traitement à l'objet de base, ce traitement soit en fait confié au méta-objet immédiatement inférieur s'il en existe un, ou à l'objet de base sinon. L'ordre des méta-objets dans la liste est important. Certains langages laissent au programmeur le soin de gérer cet ordre, d'autres essayent d'automatiser cette tâche [A-TOS], mais ce n'est possible qu'avec des hypothèses simplificatrices. D'autres langages permettent de composer les méta-objets en une structure arbitraire [Guarana].

4.2.2 Composition d'aspects

Les techniques de programmation classiques permettent de décomposer un programme en un ensemble d'éléments (qui peuvent être des procédures, des objets, etc.), chaque élément étant bien délimité, facilement accessible et composable avec d'autres. Cette décomposition facilite la programmation et la compréhension des fonctionnalités d'une application. Mais ces techniques ne permettent pas d'isoler de la même manière les propriétés "non fonctionnelles" de l'application : gestion des erreurs, optimisations, synchronisation, gestion de mémoire, etc. Ces propriétés sont donc mélangées directement dans le code fonctionnel de l'application, ce qui le rend moins lisible, moins compréhensible, et plus difficile à faire évoluer.

Le but de la programmation par aspects [AOP] est de séparer la programmation des différents aspects d'une application. Un aspect désigne toute fonction qui ne peut pas être isolée dans une partie du programme (par exemple une procédure ou un objet). Dans le cadre de la programmation par aspects, un programme est constitué d'un programme de base, qui définit les fonctions de l'application, et d'un ou plusieurs programmes séparés, écrits dans des langages éventuellement spécialisés, qui définissent les aspects à associer aux fonctions de base. Il faut ensuite un outil spécial (*aspect weaver*) pour composer ces différents programmes, soit statiquement, soit dynamiquement.

Il existe plusieurs types de langages à aspects. On peut les classer, comme dans [Aspect/J], selon leur niveau d'abstraction et leur domaine d'application. Certains langages sont en effet très spécialisés, et servent pour programmer un aspect bien particulier, avec un haut niveau d'abstraction, alors que d'autres sont de plus bas niveau mais peuvent servir à programmer différents aspects. [Aspect/J], une extension de Java, appartient à cette dernière catégorie.

Comme l'indique [AOP], et comme le montrent [Aksit] et [Lunau], la programmation par aspects est très proche de la réflexivité. Plus précisement, la composition de méta-objets peut servir de support pour implémenter la composition d'aspects. Dans ce cas, les fonctionnalités de l'application sont implémentées au niveau de base, et les aspects sont implémentés sous forme de méta-objets.

4.2.3 Comparaison avec ejava

Les langages à base de méta-objets qui permettent d'associer une liste de méta-objets à un objet de base sont très proches de la sémantique de notre modèle de composition. Il suffit de faire l'analogie entre objet de base et objet extensible, et entre méta-objet et extension.

Comme une extension, un méta-objet peut en effet intercepter les appels à l'objet de base, et ainsi surcharger certaines de ses méthodes. De plus, si on utilise une liste de méta-objets, un méta-objet peut surcharger de nouveau les méthodes déja surchargées par les méta-objets inférieurs, de même qu'une extension peut surcharger de nouveau les méthodes déja surchargées par les extensions inférieures. On peut donc dire que notre modèle de composition offre une certaine forme de réflexivité.

Il y a cependant des différences entre une extension et un méta-objet. Un méta-objet ne peut pas, comme une extension, ajouter une méthode à l'objet de base. Plus exactement, les langages typés à base de méta-objets ne peuvent pas le faire, car cela revient à modifier dynamiquement le type de l'objet de base. Une autre différence est qu'une extension surcharge des méthodes spécifiques et ne peut donc, comme un *wrapper*, être appliquée que sur un certain type d'objet de base, alors qu'un méta-objet est plus générique. Enfin, en conséquence de cette dernière différence, un appel sur un objet extensible est traité seulement par les extensions qui surchargent la méthode appelée (grâce aux tables de méthodes), alors qu'un appel sur un objet de base passe obligatoirement par tous les méta-objets, même par ceux qui ne sont pas intéressés par cet appel. D'où une plus grande efficacité de notre modèle.

En fait, la différence entre notre modèle de composition et la composition de méta-objets est plus profonde. La composition de méta-objets, comme la composition d'aspects, sert essentiellement à associer plusieurs aspects assez indépendants à un objet de base, indépendamment de son type. Le but de notre modèle de composition, est différent : il s'agit de pouvoir composer et réutiliser au maximum des "extensions", en fait des modules fournissant différents services, mais pas complètement indépendants les uns des autres. Pour faciliter la composition et la réutilisation des modules, notre modèle permet au programmeur d'exposer plus ou moins le fonctionnement interne de chaque module, sous forme de méthodes que les autres modules peuvent modifier en les surchargeant.

On a le même phénomène avec les classes dans les langages à objets. Souvent, en effet, une classe expose une partie de son fonctionnement interne sous forme de méthodes et d'attributs `protected`, pour faciliter sa réutilisation par des sous-classes.

Etant donné cette différence de but, on comprend mieux les différences mentionnées précédemment. Une extension n'a pas besoin d'être générique comme un méta-objet : c'est un module qui utilise d'autres modules dont les interfaces sont bien connues. De même, une extension ne vient pas ajouter des méthodes à l'objet extensible au sens ou le ferait un méta-objet avec un objet de base : elle offre simplement un nouveau service, ainsi que des "points d'ancrage" qui permettent aux autres extensions d'adapter et/ou d'observer son comportement.

4.3 ORBs extensibles

4.3.1 Principes et réalisations

Plusieurs travaux, reconnaissant le caractère monolithique et contraignant des ORBs existants, essayent de rendre ces derniers plus ouverts et plus flexibles. La plupart utilisent pour cela une certaine forme de réflexivité.

Tj [McAffer] a été la première plate-forme à utiliser la réflexivité dans un système distribué. Tj est programmé en CodA, un langage réflexif basé sur SmallTalk. CodA permet d'associer plusieurs méta-objets à un objet de base, selon le modèle de composition orthogonale (cf 4.2.1). Le niveau méta est ainsi décomposé en 7 aspects indépendants, chaque aspect étant géré par un méta-objet. Ces aspects sont les suivants : `send`, `accept`, `queue`, `receive`, `protocol`, `execution` et `state`. Chaque méta-objet est un objet normal, qui peut donc avoir ses propres (méta-)méta-objets. Il est également possible de définir de nouveaux aspects. Par exemple, Tj introduit trois nouveaux aspects : `marshalling`, `replication`, et `migration`. La plate-forme Tj permet de séparer le code fonctionnel de l'application, situé au niveau de base, du code non fonctionnel situé au niveau méta : `marshalling`, `migration`, `duplication`. Tj met surtout l'accent sur l'empaquetage des appels à distance, qui peut se configurer très finement à l'aide de descripteurs. On peut ainsi utiliser un format d'empaquetage différent à chaque appel à distance.

[Blair98a] propose une architecture réflexive pour la construction d'ORBs. Cette architecture est basée sur le modèle ODP, et associe à chaque interface d'un objet ODP un niveau méta décomposé en trois aspects (on dit aussi *méta-espace*) indépendants : composition, encapsulation et environnement. Chacun de ses aspects est un objet ODP, qui peut lui-même avoir un niveau méta similaire :

- le méta-objet de composition décrit la composition de l'objet auquel est attachée l'interface de base. Un objet ODP peut en effet être composé de plusieurs objets ODP internes, y compris des objets de liaison. Le méta-objet de composition permet de découvrir et de modifier cette composition.
- le méta-objet d'encapsulation décrit l'interface de base, en termes de méthodes, d'attributs, d'héritage, etc. Selon les cas, il permet également de modifier cette interface, en ajoutant de nouvelles méthodes par exemple.
- le méta-objet d'environnement décrit le modèle d'exécution de l'interface de base. Comme dans Tj, ce méta-objet gère l'arrivée des messages, leur mise en file d'attente, l'empaquetage des messages, la création et l'ordonnancement des threads, etc. Ce méta-objet est en général composé de plusieurs objets et sa composition peut être manipulée avec un méta-méta-objet de composition.

Contrairement à Tj, ce modèle n'est qu'une architecture, qui se veut indépendante de tout langage (il existe cependant un prototype en Python). En particulier, les auteurs souhaitent parvenir à une spécification de cette architecture en CORBA IDL, ce qui permettrait de composer des méta-objets écrits dans des langages différents.

[Singhai] propose un ORB réflexif, qu'il utilise pour des applications temps-réel. Contrairement aux travaux précédents, la réflexivité est ici utilisée sous une forme très limitée. Elle est en effet utilisée uniquement dans les talons et les squelettes, pour réifier trois aspects seulement : l'empaquetage des requêtes, l'envoi et la reception d'une requête. Chaque aspect est géré par un méta-objet, qui ne peut pas avoir de méta-méta-objets. La réflexivité est en fait utilisée pour construire des talons et des squelettes configurables en partie. Cette approche très simple à mettre en œuvre est très efficace : il suffit de déléguer, dans les talons et les squelettes, certains traitements à des objets séparés (`Invoker`, `Marshaller`, `Dispatcher`). Il n'est pas nécessaire d'utiliser un langage réflexif. Les auteurs montrent comment cet ORB peut être utilisé pour des applications temps-réel (en ajoutant des échéances sur les messages), pour faire du partage de charge ou de la tolérance aux fautes.

[FlexiNet] est un ORB réflexif en Java. Là encore, la réflexivité est utilisée sous une forme limitée, et ne requiert pas l'utilisation d'un langage réflexif (FlexiNet est écrit entièrement en Java). Comme dans [Singhai], la réflexivité est utilisée uniquement dans les talons et les squelettes, pour les rendre configurables. Pour cela, FlexiNet décompose les talons et les squelettes en une pile de protocoles ou de micro-protocoles. Le niveau de base est constitué de talons génériques, dont le code est généré automatiquement, et d'un squelette générique. Ce niveau ne fournit aucune fonctionnalité autre que la conversion des appels en une forme générique (réification) et inversement (réflection). Les autres niveaux sont constitués de protocoles de communication : empaquetage, envoi de message, etc.

[OpenCorba] est un ORB réflexif basé sur des méta-classes (les travaux précédents sont tous basés sur des méta-objets). Il est basé sur le langage réflexif NeoClassTalk, lui même basé sur SmallTalk. La réflexivité est utilisée pour réifier trois aspects du bus : l'invocation à distance dans les talons, les vérifications de type coté serveur, et la gestion du répertoire d'interfaces CORBA. L'utilisation de méta-classes pose le problème de la composition des méta-classes, plus complexe que celle des méta-objets [Bouraqadi-Saâdani].

[Jonathan] est un noyau d'ORB ouvert et extensible, basé sur ODP, et qui permet de définir des liaisons arbitraires entre les objets (client-serveur, flots de données, etc.). Il permet également de définir des "personnalités", qui offrent au programmeur une interface de programmation cohérente, comme par exemple RMI, CORBA ou EJB. Ce noyau d'ORB n'utilise pas la réflexivité.

4.3.2 Comparaison avec l'architecture JavaPod

Le critère de comparaison principal, celui qui nous intéresse le plus étant donné nos objectifs, est la facilité avec laquelle l'architecture proposée permet d'associer des propriétés non-fonctionnelles aux composants ou aux objets de l'application, de les composer entre elles, et également d'introduire de nouvelles propriétés non-fonctionnelles. Malheureusement, nous n'avons pour le moment aucune donnée concrète tirée de l'expérience, que ce soit pour notre proposition ou pour les autres, qui permettrait de faire cette comparaison. Nous en sommes donc réduit à des spéculations, basées sur la comparaison des architectures proposées. Les autres critères importants sont les suivants : possibilité de définir n'importe quel type de connecteur, possibilité de programmer la *plate-forme* dans plusieurs langages, performances, etc.

Comme la plupart des travaux précédents, l'architecture JavaPod utilise une certaine forme de réflexivité, et ce à deux niveaux : dans l'architecture (un conteneur peut être vu comme un méta-objet pour son composant) et dans le modèle de composition (une extension intercepte certains appels de méthode, comme un méta-objet). La réflexivité utilisée dans notre approche :

- est de type méta-objet, comme dans les autres approches (sauf [OpenCorba]),
- est utilisée seulement à certains endroits, comme dans les autres approches (sauf [McAffer] et [OpenCorba], qui utilisent des langages réflexifs ou tous les objets peuvent avoir des méta-objets ou des méta-classes). Elle est en effet utilisée dans les talons et les squelettes (comme dans toutes les approches), dans les conteneurs (comme dans [McAffer] et [Blair98a]), mais aussi, et c'est une nouveauté, dans les serveurs.
- n'impose pas de découpage fixé à l'avance en méta-espaces (comme [FlexiNet], mais contrairement aux autres approches).
- utilise un modèle de composition de "méta-objets" original : à part [FlexiNet], les propositions qui offrent la composition de méta-objets utilisent une composition "orthogonale", basé sur un découpage en méta-espaces.

En conclusion, notre approche se distingue sutout par son modèle de composition, qui est justement censé faciliter la composition des propriétés non-fonctionnelles, ainsi que l'introduction de nouvelles propriétés. En ce qui concerne les autres critères, notre architecture permet de définir n'importe quel type de connecteur (comme [Blair98a], [FlexiNet] et [Jonathan]). Sa mise en œuvre est mono-langage (seul [Blair98a] a pour ambition de définir une architecture permettant de faire interopérer des méta-objets écrits dans des langages différents). Enfin, ses performances, d'après les premiers résultats obtenus, devraient être bonnes (celles des autres propositions ne sont pas connues quantitativement).

5 Conclusion

Notre travail se place, dans le cadre de la construction d'applications réparties à base de composants, au niveau de la plate-forme logicielle qui sert de support à ces applications.

Une application distribuée requiert en effet l'utilisation d'une plate-forme logicielle, comme CORBA ou RMI, pour prendre en charge, au minimum, les communications à distance entre les composants. CORBA offre d'autres *propriétés non-fonctionnelles* (transactions, persistance, etc.) mais le programmeur doit les utiliser de façon explicite dans son application. A l'inverse, la plate-forme EJB, en utilisant une certaine forme de réflexivité, permet de séparer complètement le code fonctionnel et les propriétés non-fonctionnelles.

L'approche EJB nous semble intéressante, mais elle est encore assez limitée : en particulier, la liste des propriétés non-fonctionnelles offertes est figée. L'architecture de la plate-forme logicielle que nous proposons, la plate-forme JavaPod, est donc inspirée de celle des EJB. Elle est toutefois un peu plus générale, pour que la plate-forme puisse accueillir toute application respectant le modèle ODP. Enfin, pour pouvoir offrir un nombre de propriétés non-fonctionnelles non limité a priori, nous proposons de construire chaque élément de l'architecture par composition d'extensions. L'aspect le plus original de notre proposition ne vient d'ailleurs pas de l'architecture, mais du modèle de composition utilisé pour la mettre en œuvre.

Ce modèle de composition, dans le cadre de notre plate-forme, est plus ouvert et plus flexible que l'héritage, la délégation, et les méta-objets classiques, et est plus efficace que l'utilisation de méta-objets. Il ne résout cependant pas tous les problèmes, le plus difficile étant de trouver les bonnes interfaces pour les extensions.

D'autres travaux proposent d'utiliser la réflexivité pour rendre les plates-formes logicielles plus ouvertes et plus flexibles. Ces propositions permettent de séparer quelques propriétés non-fonctionnelles du code fonctionnel, mais le problème de leur composition reste peu évoqué (ou alors en perspectives de recherche). Nous proposons un modèle censé faciliter leur composition, mais nous n'avons pas encore fait d'expériences concrètes pour juger de son efficacité. Il reste donc maintenant à essayer d'offrir au dessus du noyau JavaPod une série d'extensions prédéfinies offrant un jeu assez large de propriétés non-fonctionnelles facilement composables.

Remerciements

Nous remercions Roland Balter, Sacha Krakowiak et Vania Marangozova pour leur commentaires et corrections qui ont permis de beaucoup améliorer la version préliminaire de ce document.

Références

[Aksit] M. Aksit, and B. Tekinerdogan, *"Solving the Modeling Problems of Object-Oriented Languages by Composing Multiple Aspects Using Composition Filters"*, ECOOP'98 AOP Workshop, Brussels, Belgium, July 1998. `http://wwwtrese.cs.utwente.nl/aop-ecoop98/position.html`

[AOP] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin, *"Aspect-Oriented Programming"*, ECOOP'97, Jyväskylä, Finland, June 1997. `http://www.parc.xerox.com/spl/-groups/eca/pubs/complete.html`

[Aspect/J] C. V. Lopes, and G. Kiczales, *"Recent Developments in AspectJ"*, ECOOP'98 AOP Workshop, Brussels, Belgium, July 1998. `http://wwwtrese.cs.utwente.nl/aop-ecoop98/position.html`

[A-TOS] R. Pawlak, L. Duchien, and G. Florin, *"An Automatic Aspect Weaver with a Reflective Programming Language"*, Reflection'99, Saint-Malo, France, July 1999. `http://tulipe.cnam.fr/personne/duchien/-bib.html`

[Blair98a] G. Blair, G. Coulson, P. Robin and M. Papathomas, *"An Architecture for Next Generation Middleware"*, Middleware'98, The Lake District, England, November 1998. `http://www.comp.lancs.ac.uk/-computing/research/mpg/index_mods.html`

[Blair98b] F. Costa, G. Blair and G. Coulson, *"Experiments with Reflective Middleware"*, ECOOP'98 Reflection Workshop, Brussels, Belgium, July 1998. `http://www-users.cs.york.ac.uk/~stuart/ecoop98/-papers.html`

[Bouraqadi-Saâdani] N. M.N. Bouraqadi-Saâdani, T. Ledoux, and F. Rivard *"Safe Metaclass Programming"*, OOPSLA'98 , Vancouver, Canada, October 1998. `http://www.emn.fr/ledoux/papers.html`

[Dalang] I. Welch, and R. Stroud, *"Dalang - A Reflective Java Extension"*, OOPSLA'98 Reflection Workshop, Vancouver, Canada, October 1998. `http://www.cs.ncl.ac.uk/people/i.s.welch/home.formal/-research.htm`

[EJB] *"Enterprise Java Beans"*, `http://java.sun.com/products/ejb/`

[FlexiNet] R. Hayton, A. Herbert, and D. Donaldson, *"Flexinet: a flexible, component oriented middleware system"*, SIGOPS'98, Sintra, Portugal, September 1998. `http://dedanann.dsg.cs.tcd.ie/~vjcahill/-sigops98/papers.html`

[Guarana] A. Oliva, and L.E. Buzato, *"The Design and Implementation of Guaraná"*, COOTS'99, San Diego, California, USA, May 1999. `http://www.dcc.unicamp.br/~oliva/guarana/index.html`

[Hagimont] D. Hagimont, and L. Ismaïl, *"A Protection Scheme for Mobile Agents on Java"*, Proc. Third ACM/IEEE Int. Conf on Mobile Computing and Networking (MobiCom'97), Budapest, September 1997. `http://sirac.inrialpes.fr/Biblio/publi.html`

[Iguana] B. Gowing, and V. Cahill, *"Meta-Object Protocols for C++: The Iguana Approach"*, Reflection'96, San Francisco, California, April 1996. `http://dedanann.dsg.cs.tcd.ie/~coyote/coyote-documents.html`

[Jonathan] B. Dumant, F. Dang Tran, F. Horn, and J.-B. Stefani, *"Jonathan: an open distributed processing environment in Java"*, Middleware'98, The Lake District, U.K., September 1998. `http://www.objectweb.org/en/jonathan/Jonathan/doc/Jonathan.html`

[Lunau] C.P. Lunau, *"Is Composition of Metaobjects = Aspect Oriented Programming"*, ECOOP'98 AOP Workshop, Brussels, Belgium, July 1998. `http://wwwtrese.cs.utwente.nl/aop-ecoop98/position.html`

[McAffer] J. McAffer, *"Meta-level architecture support for distributed objects"*, Proceedings of Reflection'96, G. Kiczales (ed), 39-62, 1996. `ftp://ftp.yl.is.s.u-tokyo.ac.jp/pub/members/jeff/docs/-iwooos95.a4.ps.gz`

[metaXa] M. Golm, *"metaXa and the Future of Reflection"*, OOPSLA'98 Reflection Workshop, Vancouver, Canada, October 1998. `http://www4.informatik.uni-erlangen.de/Projects/PM/Java/-publications.html`

[MOP] G. Kiczales, J. des Rivières, and D.G. Bobrow, *"The Art of the Metaobject Protocol"*, MIT Press, 1991.

[ODP] *"ODP Reference Model: Overview"*, ITU-T | ISO/IEC Recommendation X.901 | International Standard 10746-1, 1995.

[OpenCorba] T. Ledoux, *"OpenCorba: a Reflective Open Broker"*, Reflection'99, Saint-Malo, France, July 1999. `http://www.emn.fr/ledoux/papers.html`

[Singhai] A. Singhai, A. Sane, and R.H. Campbell, *"Reflective ORBs: Supporting Robust Time-Critical Distribution"*, ECOOP'97 Reflection Workshop, Jyväskylä, Finland, June 1997. `http://choices.cs.uiuc.edu/-Papers.html`

[TOS] R. Pawlak, *"TOS: a class-based reflective language on Tcl"*, Technical Report 9902, Laboratoire CEDRIC-CNAM, 1999. `http://tulipe.cnam.fr/personne/duchien/bib.html`

A Compléments

Cette section regroupe quelques remarques d'ordre technique sur la mise en œuvre de la plate-forme JavaPod.

Compléments sur le noyau JavaPod

Le noyau JavaPod maintient l'invariant suivant (cf section 3.2.1) : pour un conteneur, un connecteur, et une interface d'accès donnés, il existe au plus un talon et au plus un squelette. Cet invariant n'est peut être pas assez fort. Rien n'empêche en effet d'avoir, dans un composant, plusieurs squelettes, appartenant à des connecteurs différents, mais ayant la même interface et donnant accès au même objet interne. Dans ce cas, lorsque le composant souhaite exporter cet objet, quelle référence de connecteur doit on utiliser ? Actuellement, on retourne celle du squelette le plus ancien. On pourrait donc interdire, par un autre invariant, une telle situation. D'un autre coté, l'exemple précédent peut être pratique, par exemple pour rendre accessible un objet à la fois par CORBA et par RMI. Il faudrait donc peut être trouver un moyen de combiner notre notion de référence avec le mécanisme de [Jonathan] qui permet d'avoir plusieurs types de données de liaison dans une même référence. Mais ce mécanisme a également un problème, coté client : quand on reçoit une telle référence, quelles données de liaison doit on utiliser ? Actuellement, dans [Jonathan], ce sont les premières trouvées qui permettent d'établir la liaison.

Les sous-classes de `Stub` générées pour les talons héritent de `ExtensibleObject`, et héritent d'une interface fonctionnelle qui n'hérite pas elle même de `EInterface`, contrairement aux spécificationss de ejava. Ces classes étant générées directement, sans passer par un compilateur ejava, on peut facilement contourner les règles du langage. Pour que ce ne soit plus une entorse à la règle, il faudrait élargir cette dernière : dans ce cas, les méthodes qui implémentent une interface "normale" ne devraient pas être considérées comme des eméthodes, et suivre la sémantique habituelle de Java. Mais cela risque d'embrouiller un peu le programmeur.

Compléments sur ejava

Le *cast* généralisé, du type `((D)c)`, n'est autorisé que juste avant un appel d'eméthode. Il serait possible d'autoriser ce genre de *cast* n'importe où. Le problème est qu'il n'aurait pas la sémantique habituelle d'un *cast*, sauf peut être en modifiant la sémantique d'un certain nombre de constructions de Java. On aurait par exemple (en reprenant l'exemple de la figure 6) `((D)c) == c`, comme on peut s'y attendre, mais `((D)c) instanceof D` retournerait `false`, puisque `c instanceof D` est faux. Il faudrait peut être alors généraliser la sémantique du `instanceof`. Mais ce genre de *cast* pose également des problèmes avec l'accès aux attributs, même s'ils sont privés, et aux méthodes normales des extensions. Par exemple `((D)c).toString()` devrait-il appeler la méthode `toString` de `c` ou celle de son extension de type `D` ?

Les eméthodes ne doivent lancer que des exceptions qui héritent de `EException` (cf section 3.1.2). Cette restriction, qui peut paraître étrange, permet de faciliter la traduction de ejava vers Java. En reprenant l'exemple de la figure 6, supposons que `m` puisse lancer une exception de type `Exception`. Puisque `invoke` peut appeler `m`, elle devrait être déclarée comme pouvant lancer n'importe quel type d'exception. Mais du coup, l'appel à `n`, qui est traduit par un appel à `invoke`, serait considéré comme pouvant lancer une exception, alors que ce n'est pas le cas. Pour résoudre le problème, il faudrait encapsuler chaque appel à `invoke` dans un bloc `try catch`, afin d'ignorer les exceptions dont on sait, au niveau ejava, qu'elles ne peuvent pas être lancées. Le problème est qu'un bloc `try catch` ne peut pas apparaître dans une expression (alors qu'un appel à une eméthode le peut). Grâce à notre restriction, les eméthodes ne peuvent plus lancer que des exceptions qui héritent de `RuntimeException` (que l'on n'est pas obligé de capturer ou de déclarer, contrairement aux autres exceptions). Cela nous permet d'éviter l'utilisation des blocs `try catch`.

Table des matières

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399