INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_|_|_|_|_|_|_|_|_|_|

THÈSE

pour obtenir le grade de

DOCTEUR DE l'INPG

Spécialité : « Informatique Systèmes et Communications »

préparée au laboratoire SIRAC dans le cadre de l'Ecole Doctorale
« Mathématiques Sciences et Technologies de l'Information »

présentée et soutenue publiquement par

Eric Bruneton

le ? ? 2001

Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties

---

Directeur de thèse :

Michel Riveill

JURY

M.	?.	?	Président
M.	André	Schiper	Rapporteur
M.	Pierre	Cointe	Rapporteur
M.	Michel	Riveill	Directeur de thèse
M.	Fabienne	Deschamboux	Co-encadrante
M.	Jean-Bernard	Stefani	Examinateur
M.	Jacques	Mossière	Examnateur

Résumé

En général, les propriétés *non-fonctionnelles* des applications distribuées, comme la persistance, la protection ou la mobilité, ont besoin d'être *adaptées* en fonction de l'environnement d'exécution et des besoins des utilisateurs, comme nous le montrons à l'aide d'une application particulière, appelée BAGHERA. Malheureusement cette adaptation est difficile à réaliser à l'aide des plates-formes *middleware* classiques comme CORBA. La plate-forme *Enterprise Java Beans* (EJB), ainsi qu'un certain nombre de plates-formes expérimentales, apportent des débuts de solution à ce problème. Mais, comme nous le montrons, ces propositions ne sont pas encore complètement satisfaisantes.

Nous proposons donc dans cette thèse une nouvelle architecture de plate-forme, afin d'essayer de résoudre les limitations des propositions existantes. Cette nouvelle architecture, inspirée de celle des EJB, et complétée en utilisant un nouveau mécanisme de composition d'objets, offre un modèle de programmation très général, inspiré du modèle *Open Distributed Processing* (ODP).

Nous présentons ensuite la plate-forme expérimentale JavaPod, basée sur l'architecture précédente, et que nous avons réalisé à des fins d'évaluation. Ce prototype est constitué d'un noyau, qui se contente de définir un cadre de travail (ou *framework*), et de plusieurs couches d'extensions du noyau, qui fournissent toutes les fonctionnalités de la plate-forme (protocoles, propriétés non-fonctionnelles, déploiement...). Le tout est mis en œuvre dans un nouveau langage appelé ejava, une extension de Java dans laquelle les objets peuvent être directement composés selon notre nouveau mécanisme de composition.

Finalement, nous présentons comment nous avons réalisé l'application BAGHERA avec la plate-forme JavaPod, et comment nous avons pu adapter ses propriétés non-fonctionnelles. Le résultat de ces expérimentations est que notre architecture permet d'adapter les propriétés non-fonctionnelles des applications distribuées plus facilement qu'avec les plates-formes expérimentales existantes, même s'il reste encore plusieurs problèmes non résolus.

Abstract

The *non-functional properties* of distributed applications, such as persitency, protection or mobility, generally need to be *adapted* to the execution environment or to the users needs, as we show it with a specific application, called BAGHERA. Unfortunately, this adaptation is difficult to do with classical *middleware* platforms, such as CORBA. The *Enterprise Java Beans* (EJB) platform, as well as some experimental middleware platforms, partially solve this problem. But as we show it, these propositions are not yet completely satisfactory.

We therefore propose, in this thesis, a new middleware platform architecture, in order to try to solve the limitations of existing proposals. This new architecture, inspired from the EJB architecture, and completed with a new object composition mechanism, offers to applications a very general programming model, inspired from the *Open Distributed Processing* (ODP) computational model.

We then present the JavaPod platform, an experimental platform based on the previous architecture, and that we implemented for evaluation purposes. This prototype is made of a small kernel, which only defines a framework, and of several kernel's extension layers, which implement all the functionalities of the platform (protocols, non-functional properties, deployement...). The whole platform is implemented in ejava, a Java extension in which objects can be composed by using our new object composition mechanism.

Finally, we present how we implemented the BAGHERA application with the JavaPod platform, and how we were able to adapt its non-functional properties. The result of these experiments is that it is easier to adapt the non-functional properties of distributed applications with our architecture than with existing experimental platforms, even if several open issues remain.

Table des matières

Introduction **1**

1 Motivations **5**

1.1 L'application BAGHERA . . . 5

1.1.1 Le projet BAGHERA . . . 5

1.1.2 Une version simplifiée . . . 6

1.2 Scénarios d'adaptation . . . 7

1.2.1 Persistance . . . 8

1.2.2 Protection . . . 8

1.2.3 Mode déconnecté . . . 9

1.3 Besoins au niveau du middleware . . . 9

1.3.1 Définitions . . . 9

1.3.2 Séparation du code fonctionnel et non-fonctionnel . . . 10

1.3.3 Séparation et composition des propriétés non-fonctionnelles . . . 11

1.3.4 Modularité et extensibilité . . . 11

1.4 Synthèse . . . 11

2 État de l'art **13**

2.1 Rappels sur la réflexivité . . . 13

2.1.1 Généralités . . . 13

2.1.2 Application aux langages orientés objets . . . 14

2.1.3 Composition de méta-objets . . . 16

2.1.4 Programmation orientée aspects . . . 18

2.2 État de l'art . . . 19

2.2.1 La plate-forme Tj . . . 19

2.2.2 Singhai et al. . . . 22

2.2.3 Entreprise Java Beans . . . 24

2.2.4 FlexiNet . . . 27

2.2.5 Blair et al. . . . 29

2.2.6 Autres travaux . . . 32

2.3 Synthèse . . . 32

3 Proposition **35**
3.1 Modèle de programmation . . . 35
3.1.1 Composants, connecteurs et interfaces . . . 36
3.1.2 Références d'interface . . . 38
3.1.3 Références de connecteur . . . 40
3.1.4 Passage de références . . . 41
3.2 Architecture de haut niveau . . . 42
3.2.1 Conteneurs . . . 43
3.2.2 Talons, squelettes et portes . . . 43
3.2.3 Serveurs . . . 44
3.3 Modèle de composition . . . 44
3.3.1 Les besoins . . . 45
3.3.2 Limites des mécanismes existants . . . 46
3.3.3 Un nouveau mécanisme . . . 50
3.4 Résumé de notre proposition . . . 52

4 Mise en œuvre **55**
4.1 Le langage ejava . . . 55
4.1.1 Motivation . . . 55
4.1.2 Choix de conception . . . 56
4.1.3 Définition succincte . . . 58
4.1.4 Mise en œuvre . . . 60
4.2 Le noyau JavaPod . . . 64
4.2.1 Organisation générale . . . 66
4.2.2 Les classes `Component` et `JavaPodInterface` . . . 66
4.2.3 La classe `ServerImpl` . . . 67
4.2.4 La classe `ContainerImpl` . . . 67
4.2.5 Les classes `StubImpl` et `SkeletonImpl` . . . 68
4.2.6 L'interface `Reference` . . . 69
4.3 Les extensions du noyau JavaPod . . . 69
4.3.1 Organisation générale . . . 70
4.3.2 Extensions pour les connecteurs client-serveur . . . 71
4.3.3 Extensions pour la configuration et le déploiement . . . 74
4.4 Exemple d'utilisation . . . 77
4.4.1 Programmation . . . 77
4.4.2 Déploiement . . . 78

5 Expérimentations avec BAGHERA **81**
5.1 Implantation de BAGHERA . . . 81
5.2 Persistance . . . 83
5.2.1 Algorithme . . . 83
5.2.2 Mise en œuvre . . . 84
5.2.3 Application à BAGHERA . . . 85
5.3 Protection . . . 86

5.3.1 Algorithme . . . 86
5.3.2 Mise en œuvre . . . 87
5.3.3 Application à BAGHERA . . . 88
5.4 Mode déconnecté . . . 92
5.4.1 Algorithme . . . 92
5.4.2 Mise en œuvre . . . 93
5.4.3 Application à BAGHERA . . . 97

6 Résultats 101
6.1 Résultats qualitatifs . . . 101
6.1.1 Séparation du code fonctionnel et non-fonctionnel . . . 101
6.1.2 Composition de propriétés non-fonctionnelles . . . 103
6.1.3 Modularité et extensibilité . . . 105
6.1.4 Évaluation du mécanisme de composition . . . 106
6.2 Résultats quantitatifs . . . 107
6.2.1 Performances de ejava . . . 107
6.2.2 Performances de l'appel de méthode à distance . . . 108
6.2.3 Performances de la persistance . . . 109
6.2.4 Performances de la protection . . . 110
6.2.5 Synthèse . . . 111

Conclusion 113

A Définition du langage ejava 117
A.1 Syntaxe . . . 117
A.2 Contraintes statiques . . . 119
A.3 Classes prédéfinies . . . 120
A.4 Sémantique . . . 121

B Un mécanisme de contrôle d'accès pour ejava 125
B.1 Motivation . . . 125
B.2 Modèle . . . 126
B.3 Mise en œuvre . . . 128

Bibliographie 129

Introduction

Dans tous les secteurs économiques, les applications distribuées sont de plus en plus utilisées. L'une des raisons est que les administrations et les entreprises deviennent de plus en plus distribuées géographiquement. Une autre raison est le développement rapide du réseau Internet. On voit donc apparaître de plus en plus d'applications de partage distribué d'informations et de travail coopératif à distance : télé-conférence, enseignement à distance, conception assistée par ordinateur...

Pour répondre efficacement à cette augmentation des besoins dans le domaine des applications distribuées, il faut disposer d'outils permettant de programmer facilement ce type d'application, malgré les difficultés inhérentes à la distribution (hétérogenéité, pannes, indéterminisme...). La solution traditionnelle consiste à utiliser une couche intermédiaire entre le système d'exploitation et les applications, appelée pour cette raison une plate-forme *middleware*[1]. Ces plates-formes fournissent des mécanismes de communication à distance, ainsi que des *services systèmes* qui peuvent être utilisés par les applications pour résoudre les problèmes liés à la distribution. Par exemple, l'impact des pannes et de l'indéterminisme peut être réduit en utilisant des transactions, qui peuvent être gérées par un moniteur transactionnel fournit par la plate-forme.

Selon nous, l'idée d'introduire une couche intermédiaire entre les applications et le système d'exploitation est une bonne idée, qui doit être conservée. Malheureusement, les mises en œuvre actuelles de cette idée ne sont pas encore assez satisfaisantes, notamment parce qu'elles ne permettent pas de programmer des applications *adaptables*. Avant d'expliquer pourquoi, la section suivante précise ce que nous entendons par « adaptable », et montre pourquoi les applications distribuées ont souvent besoin de l'être.

Applications adaptables

Le verbe « adapter » signifie soit « rendre un système (un dispositif, des mesures...) apte à assurer ses fonctions dans des conditions particulières ou nouvelles », soit « harmoniser, rendre conforme à ». Adapter un logiciel (système d'exploitation, plate-forme middleware, application...) signifie donc, selon nous, soit « rendre ce logiciel apte à fonctionner dans un environnement d'exécution (vitesse du processeur, débit du ré-

[1] faute d'un consensus sur une traduction de ce terme, nous utilisons le terme anglais dans le reste de ce document.

seau...) particulier ou nouveau », soit « harmoniser, rendre conforme ce logiciel aux besoins de l'utilisateur (humain ou non) ». Plus précisément, nous distinguons trois types d'adaptations de logiciels :

- l'adaptation *statique* a lieu avant l'exécution du logiciel, c'est à dire qu'elle se traduit soit par une modification du code source, soit par une compilation avec des options différentes, soit par des options de lancement ou déploiement différentes...
- l'adaptation *dynamique* a lieu pendant l'exécution du logiciel, et est initiée par un intervenant extérieur (humain ou non) ;
- l'*auto-adaptation* intervient également pendant l'exécution, mais est initiée par le logiciel lui-même.

Dans le domaine particulier des applications distribuées, l'adaptation statique peut être nécessaire au moment du déploiement de l'application, pour tenir compte des particularités de l'environnement d'exécution (débit disponible, taux de pannes du réseau et des machines...). De même, l'adaptation dynamique peut être nécessaire pour répondre aux évolutions lentes de cet environnement (croissance du nombre d'utilisateurs, ajout de machines, mises à jour de composants logiciels...), et si l'application ne peut pas être arrêtée (sinon il suffit d'arrêter l'application, de l'adapter statiquement, puis de la redémarrer). Enfin, l'auto-adaptation est nécessaire pour réagir aux variations rapides de l'environnement d'exécution, dues par exemple à la mobilité des utilisateurs ou des terminaux.

Les applications distribuées ont donc souvent besoin d'être adaptables. Malheureusement, comme nous l'avons dit au début de ce chapitre, les plates-formes middleware classiques ne permettent pas de programmer facilement des applications adaptables. La section suivante explique pourquoi.

Limites des plates-formes middleware classiques

L'adaptation statique des propriétés *non-fonctionnelles*[2], comme la persistance, les transactions ou la protection, pose problème pour la raison suivante. Les plates-formes middleware classiques obligent le programmeur à inclure du code lié à ces propriétés dans le code de l'application elle-même (par exemple, avec CORBA, le programmeur doit délimiter lui-même les transactions, à l'aide de méthodes du genre `beginTransaction` et `endTransaction`). Par conséquent, pour adapter ces propriétés, il faut modifier le code de l'application. C'est faisable si le code source est disponible, mais peu pratique. Dans le cas contraire, par exemple si l'application est constituée de composants vendus sous forme compilée, c'est plus difficile, voire impossible.

L'adaptation dynamique pose également problème, principalement parce que la plupart des plates-formes n'offrent pas de mécanismes de reconfiguration de base (création, remplacement et destruction de composants ou de connecteurs, migration de composants...). Mais il manque également des outils pour préserver la cohérence de l'application lors des reconfigurations (pas de perte ou de duplication de messages...).

Finalement l'auto-adaptation est la plupart du temps impossible, car elle requiert

[2]nous donnerons une définition précise de ce terme dans la suite du document.

de pouvoir contrôler finement les détails de mise en œuvre de la plate-forme middleware sous-jacente (politique d'ordonnancement, de gestion de tampons, choix des protocoles de communications...). Malheureusement les plates-formes classiques suivent une approche de type « boîte noire », c'est à dire que l'implantation des interfaces fournies est complètement cachée, et donc non modifiable.

Notre problématique

La problématique étudiée dans cette thèse découle du problème exposé ci-dessus, à savoir le manque de plates-formes permettant de programmer des applications distribuées adaptables, dont on a de plus en plus besoin. Plus précisément, nous avons essayé de concevoir, de réaliser et d'expérimenter une plate-forme middleware permettant de programmer des applications *statiquement* adaptables.

En effet, nous ne pouvions pas nous attaquer aux trois types d'adaptation à la fois. Il fallait donc faire un choix. Or, pour résoudre le problème de l'auto-adaptation, il faut déjà savoir faire des adaptations dynamiques. De même, pour pouvoir faire des adaptations dynamiques, il faut déjà pouvoir faire des adaptations statiques sans avoir de code à modifier. Autrement dit, ces trois problèmes sont, dans une certaine mesure, « emboîtés » les uns dans les autres. Nous avons donc décidé de commencer par le plus fondamental, à savoir le problème de l'adaptation statique.

Plan du document

Le chapitre 1 définit plus précisément les objectifs de notre travail, et surtout les motive à l'aide d'un exemple d'application réaliste. Le chapitre 2 présente les principaux travaux déjà effectués dans ce domaine. Nous présentons ensuite, dans le chapitre 3, l'architecture que nous proposons d'utiliser pour pouvoir atteindre nos objectifs. Le chapitre 4 présente le prototype que nous avons réalisé pour tester l'architecture proposée. Les chapitres 5 et 6 présentent ensuite les tests réalisés avec ce prototype, ainsi que leurs résultats. Finalement, le dernier chapitre fait la synthèse des résultats obtenus, et conclut en citant les principaux problèmes restant à résoudre.

Chapitre 1

Motivations

A partir d'une étude de cas réaliste, ce chapitre met en évidence les problèmes qu'une plate-forme « adaptable » doit résoudre. La première section présente BAGHERA, une application d'enseignement de la géométrie assisté par ordinateur. La section suivante présente quelques besoins des utilisateurs de cette application, qui montrent que l'application doit être adaptée différemment selon son contexte d'utilisation. Enfin la troisième section présente les propriétés que doit satisfaire une plate-forme middleware afin de pouvoir répondre aux besoins précédents.

1.1 L'application BAGHERA

L'application BAGHERA est une version simplifiée d'une application d'enseignement assisté par ordinateur, appliquée à la géométrie, qui sera conçue et réalisée dans le cadre du projet BAGHERA [Bal00]. Cette application est présentée ici non seulement pour montrer que la problématique étudiée dans ce document découle de besoins utilisateurs bien réels, mais aussi parce qu'elle a été utilisée pour évaluer notre travail (cf. section 5 page 81). Nous présentons brièvement le projet BAGHERA avant de présenter l'application BAGHERA elle-même.

1.1.1 Le projet BAGHERA

Le projet IMAG BAGHERA regroupe trois équipes de recherche travaillant dans des domaines complémentaires. Il s'agit des équipes MAGMA, ATINF et Did@TIC, qui travaillent respectivement sur les systèmes multi-agents pour l'intelligence artificielle, sur la démonstration automatique de théorèmes, et sur la didactique des mathématiques.

Le but de ce projet est de concevoir et de mettre en œuvre une application pour l'apprentissage de la géométrie. Plus précisément, le but est d'automatiser le plus possible le processus d'apprentissage, tout en faisant en sorte qu'il s'adapte automatiquement au niveau et au rythme de chaque élève. Cette application sera distribuée, et pourra être utilisée par plusieurs élèves et professeurs simultanément. Chaque utilisateur, qu'il soit élève ou professeur, sera assisté par un agent intelligent. L'agent assistant d'un

élève sera chargé de suivre et de faire progresser l'élève. Par exemple, en coopérant avec d'autres agents, il devra déduire si l'élève a assimilé ou non tel ou tel concept de géométrie, et ce à partir des solutions proposées par l'élève à un certain nombres d'exercices antérieurs. Il devra alors proposer à l'élève des exercices adaptés à son niveau et aptes à le faire progresser. L'agent assistant d'un professeur permettra de superviser un groupe d'élèves de même niveau (ces groupes seront construits automatiquement et dynamiquement). Il permettra au professeur d'intervenir manuellement dans le processus d'apprentissage si les décisions prises de façon automatique ne lui semble par correctes.

Le statut actuel du projet BAGHERA est encore loin de ces objectifs. Il existe un prototype, mais les agents qui le composent ne sont pas encore intelligents. Ces agents se répartissent en trois types principaux :

- les agents qui gèrent les *cartables électroniques* des élèves. Un cartable électronique contient les exercices résolus par un élève, ceux qui lui restent à résoudre, ainsi que les messages échangés avec les autres utilisateurs (élèves ou professeurs) ;
- les agents assistants des élèves. Ils se présentent sous la forme d'une interface graphique, qui permet de consulter le contenu du cartable électronique, de proposer une solution pour un exercice à résoudre, et d'envoyer des messages aux autres utilisateurs ;
- les agents assistants des professeurs. Ils se présentent eux aussi sous la forme d'une interface graphique, qui permet de consulter les cartables électroniques des élèves, de leur proposer de nouveaux exercices, de corriger les solutions proposées par les élèves, et enfin d'envoyer des messages aux autres utilisateurs.

1.1.2 Une version simplifiée

Afin de nous concentrer sur les besoins et les problèmes liés à la distribution, nous avons défini une version simplifiée de l'application visée par le projet BAGHERA, dans laquelle tous les aspects liés à l'intelligence artificielle ont été supprimés. Cette application, appelée BAGHERA, est assez proche du prototype présenté dans la section précédente, et est organisée en composants[1]. Les différents types de composants utilisés sont les suivants (cf. figure 1.1 page ci-contre) :

- un composant de type `ElectronicCase` correspond au cartable électronique d'un élève. Il contient les exercices résolus par un élève, ceux qui lui restent à résoudre, ainsi que les messages échangés avec les autres utilisateurs ;
- un composant de type `Mailbox` correspond au cartable électronique d'un professeur. Il contient les messages échangés par un professeur avec les autres utilisateurs[2] ;
- le composant `ExerciseRepository` contient une liste d'exercices (avec leurs solutions). Il permet aux professeurs de partager une base d'exercices, qu'ils peuvent compléter au fur et à mesure, et dans laquelle ils peuvent choisir des exercices à

[1] nous définirons ce terme par la suite (cf. section 3.1.1 page 36). Dans cette section, il est équivalent au concept d'objet, comme dans CORBA, Java RMI...

[2] par analogie avec un cartable réel, il devrait également contenir les exercices à corriger. Ce serait tout à fait possible, mais nous avons choisi de laisser ces exercices dans les cartables des élèves.

proposer à leurs élèves ;
- le composant `VirtualSchool` est un composant central, qui possède des références vers tous les composants `ElectronicCase`, `Mailbox`, et `ExerciseRepository`. Il permet ainsi d'obtenir une référence vers le cartable électronique d'un élève, vers la boîte aux lettres d'un professeur, ou vers la base d'exercices. Ce composant contient la liste des élèves, la liste des professeurs, ainsi que la répartition des élèves en classes (chaque professeur s'occupe d'une classe et une seule) ;
- les composants de type `StudentAgent` et `ProfessorAgent` sont les « agents » assistants des élèves et des professeurs. Ils se présentent sous la forme d'une interface graphique et ont les mêmes fonctions que dans le prototype du projet BAGHERA ;
- enfin, un composant de type `AdminAgent`, qui se présente lui aussi sous forme d'une interface graphique, permet de modifier la liste des élèves, la liste des professeurs, et la répartition des élèves en classes.

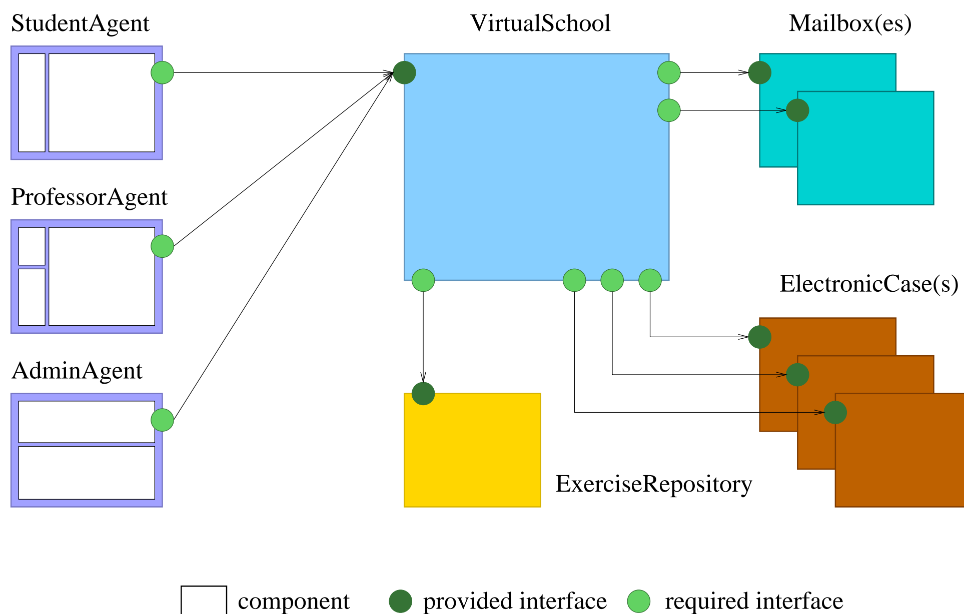


FIG. 1.1 – Architecture de l'application BAGHERA

1.2 Scénarios d'adaptation

Cette section présente trois exemples de *besoins utilisateur* associés à l'application BAGHERA. Bien évidemment ce ne sont pas les seuls, mais nous nous sommes limité à ces trois exemples car ils représentent une bonne palette des besoins, et surtout des difficultés qu'il faut résoudre pour les satisfaire. Tous ces besoins nécessitent des *adaptations* de l'application.

1.2.1 Persistance

Un premier besoin pour les utilisateurs est le besoin de *persistance* : les composants `VirtualSchool`, `ElectronicCase`, `Mailbox`, et `ExerciseRepository` contiennent des données qui ne doivent pas disparaître quand un utilisateur quitte l'application.

Or, dans la plupart des plates-formes middleware, les composants ne sont pas persistants par défaut. C'est à dire que si on n'indique pas explicitement, dans le code des composants ou séparément selon les cas, que l'on souhaite qu'ils soient persistants, alors ils ne le sont pas. Par conséquent, le besoin utilisateur de persistance implique un besoin d'adaptation de la version de base de l'application BAGHERA, supposée programmée sans prendre en compte la persistance.

Bien que ce besoin de persistance soit pratiquement toujours présent, l'adaptation de l'application qui permet de le satisfaire n'est pas la même selon l'environnement considéré :

- si les composants persistants sont déployés sur une machine qui n'est jamais arrêtée et ne tombe jamais en panne, aucune adaptation n'est nécessaire pour satisfaire le besoin de persistance ;
- si les composants persistants sont déployés sur une machine pouvant tomber en panne, mais assez puissante, alors on peut utiliser une base de données comme support de persistance ;
- si par contre un composant persistant est déployé sur un PDA (*Portable Digital Assistant*), alors il faut utiliser un support de persistance plus « léger ». Par exemple, on peut se contenter de stocker le composant dans un fichier.

1.2.2 Protection

Un autre besoin pour les utilisateurs est le besoin de *protection*. Par exemple, les professeurs peuvent souhaiter que les élèves ne puissent pas récupérer la solution d'un exercice dans le composant `ExerciseRepository`, ou dans le cartable électronique d'un autre élève. De même, les élèves peuvent souhaiter que les professeurs ne puissent pas consulter les messages contenus dans leurs cartables électroniques.

Or, comme pour la persistance, les composants ne sont pas protégés par défaut dans la plupart des plates-formes middleware. Par conséquent, le besoin de protection implique un besoin d'adaptation de la version de base de l'application BAGHERA, supposée programmée sans prendre en compte la protection.

Comme pour la persistance, le besoin de protection, bien que pratiquement toujours présent, ne requiert pas les mêmes adaptations de l'application en fonction de l'environnement :

- si les élèves ne connaissent pas la programmation, ils sont obligés d'utiliser l'application par l'intermédiaire de l'interface graphique des agents assistants, qui ne permet pas de faire n'importe quelle opération (comme par exemple de consulter le cartable électronique d'un autre élève). Par conséquent, dans ce cas, aucune adaptation de l'application n'est nécessaire pour satisfaire le besoin de protection ;
- si par contre les élèves connaissent la programmation, alors ils peuvent en principe contourner les limitations imposées par l'interface graphique, et donc, dans ce cas,

une adaptation de l'application est nécessaire ;
- de même pour les professeurs : s'ils connaissent la programmation, et si on ne peut pas leur faire confiance, alors il faut adapter l'application pour les empêcher d'accéder aux messages échangés entre les élèves. Dans le cas contraire, cette adaptation n'est pas nécessaire ;
- enfin, il est parfois nécessaire d'adapter l'algorithme mettant en œuvre la politique de protection choisie : on a en effet le choix entre des listes de contrôle d'accès, des capacités...

1.2.3 Mode déconnecté

Un troisième besoin pour les utilisateurs est de pouvoir utiliser l'application en *mode déconnecté*, c'est à dire depuis une machine non connectée au réseau où sont situés les composants principaux de l'application (i.e. les composants persistants dans le cas de BAGHERA). Par exemple, on peut souhaiter que les élèves puissent écrire des messages et résoudre des exercices alors qu'ils sont déconnectés, et que, lors de la reconnexion, ces messages et ces exercices résolus soient envoyés automatiquement à leurs destinataires.

Or, comme pour la persistance et la protection, ce besoin n'est pas satisfait par défaut. Pour le satisfaire, une adaptation de la version de base de l'application est donc nécessaire.

Comme pour la persistance et la protection, l'adaptation requise pour pouvoir utiliser l'application en mode déconnecté peut a priori dépendre de l'environnement. Mais, de toute façon, cette adaptation n'est pas toujours nécessaire, car le travail en mode déconnecté n'est pas toujours un besoin. Par exemple, si l'application est déployée dans une école, sur des machines connectées en permanence à un réseau local, ce besoin n'apparaît pas. Mais il apparaît si on souhaite que l'application puisse être également utilisée par des élèves à l'hôpital, ne disposant que d'un PDA non connecté en permanence au réseau de l'école.

1.3 Besoins au niveau du middleware

La section précédente montre que certains besoins utilisateurs concernant les applications distribuées (comme par exemple la persistance, la protection, et le mode déconnecté) induisent un besoin de plus « bas niveau », qui est celui de pouvoir *adapter* les applications. Cette section montre que ce besoin induit à son tour d'autres besoins à un niveau encore plus bas, qui est celui de la plate-forme middleware.

1.3.1 Définitions

Avant d'étudier les besoins induits au niveau du middleware, il est utile d'introduire les termes suivants, qui sont utilisés fréquemment dans la suite du document.

Définition 1 *Une* propriété non-fonctionnelle *est une propriété qui peut être associée à une application sans modifier la* nature *du service rendu par cette application à ses utilisateurs.*

La persistance, la protection, l'utilisation en mode déconnecté sont des exemples de propriétés non-fonctionnelles. On peut également citer la duplication, la migration, les transactions... Toutes ces propriétés ne font que modifier la *qualité* du service rendu par une application : elles ne modifient pas la *nature* de ce service. Par exemple, l'application BAGHERA reste une application d'enseignement assisté par ordinateur, qu'on lui associe ou non l'une ou l'autre de ces propriétés. De même pour une application de commerce électronique, de téléconférence...

Définition 2 *Le* code fonctionnel *d'une application est celui qui réalise le service de base rendu par cette application. Le* code non-fonctionnel *est celui qui associe à l'application ses propriétés non-fonctionnelles.*

Par exemple, dans une application bancaire, le code fonctionnel de la méthode `transfer` est celui qui débite un compte et crédite l'autre. Le code non-fonctionnel est celui qui assure que ces opérations sont effectuées de façon atomique et transactionelle.

Définition 3 *Le code nécessaire pour associer une propriété non-fonctionnelle à une application contient généralement une grande proportion de code non spécifique à l'application. Un* service système *est un service qui factorise tout ou partie de ce code non spécifique.*

Un service système permet de simplifier considérablement l'écriture du code non-fonctionnel d'une application, puisqu'il suffit de se concentrer sur la partie de ce code qui est spécifique à l'application. Par exemple, CORBA offre un service système de haut niveau qui permet de manipuler des transactions avec quelques primitives comme `beginTransaction` et `endTransaction`. Dans l'exemple de l'application bancaire, le code qui assure l'atomicité des virements entre comptes se réduit alors à deux appels à ces primitives.

Ces définitions permettent de reformuler de façon plus précise les conclusions de la section 1.2 page 7 : les utilisateurs d'une application distribuée ont besoin de pouvoir associer à cette application différentes propriétés non-fonctionnelles, selon l'environnement dans lequel elle est déployée. Pour cela, ils ont besoin d'adapter *le code non-fonctionnel* de cette application. Les sections suivantes montrent les besoins qui en découlent au niveau du middleware.

1.3.2 Séparation du code fonctionnel et non-fonctionnel

Les applications sont généralement vendues sous forme de code exécutable non modifiable par l'utilisateur. Si on souhaite néanmoins qu'un utilisateur puisse associer à son application des propriétés non-fonctionnelles, qui peuvent être spécifiques à cet utilisateur, alors le middleware doit être organisé de telle façon que le code exécutable fonctionnel puisse être séparé du code exécutable non-fonctionnel, tout en permettant aux deux de fonctionner ensemble correctement.

L'autre raison pour laquelle cette séparation est nécessaire, et sans doute la plus importante, est que l'on souhaite pouvoir adapter les applications au moment du déploiement ou à l'exécution. Autrement dit, on souhaite avoir un minimum de code à

modifier pour faire une adaptation donnée, l'idéal étant de n'en avoir aucun, c'est à dire d'avoir réussi à isoler le code fonctionnel du reste.

1.3.3 Séparation et composition des propriétés non-fonctionnelles

Les besoins utilisateurs présentés dans la section 1.2 page 7 ont été présentés séparément, mais il est évident qu'ils peuvent être combinés. Par exemple, on peut imaginer qu'il y ait besoin d'associer à l'application BAGHERA à la fois des propriétés de persistance, de protection, et d'utilisation en mode déconnecté, ou bien seulement de persistance et de protection, ou bien encore seulement de persistance...

Rien qu'avec ces trois propriétés, on obtient $2^3 = 8$ combinaisons possibles. Si on prend en compte d'autres propriétés non-fonctionnelles, et plusieurs versions possibles par propriété (comme on l'a vu pour la persistance), alors ce nombre de combinaisons, qui croit de façon exponentielle, devient vite extrêmement grand. Par conséquent, il n'est pas envisageable d'avoir à programmer un code non-fonctionnel différent pour chaque combinaison possible. Le middleware doit donc non seulement permettre la séparation du code fonctionnel et non-fonctionnel, mais aussi la séparation du code non-fonctionnel fournissant chaque propriété, tout en permettant à ces codes différents de fonctionner ensemble correctement.

1.3.4 Modularité et extensibilité

Il est impossible de réaliser une plate-forme middleware prenant en compte toutes les propriétés non-fonctionnelles dont pourrait avoir besoin les utilisateurs, de même qu'il est impossible de réaliser un système d'exploitation adapté aux besoins de toutes les applications. Un moyen pour éviter ce problème est d'utiliser une architecture modulaire et extensible, qui permet aux utilisateurs de configurer la plate-forme middleware selon leurs besoins propres. Si on applique ce principe aux propriétés non-fonctionnelles, on en déduit les deux besoins suivants :

- les services systèmes doivent pouvoir être programmés séparément, de façon à pouvoir choisir au démarrage de la plate-forme (voire à l'exécution) les services systèmes nécessaires et eux seulement ;
- le mécanisme permettant de composer le code fonctionnel avec le code non-fonctionnel de chaque propriété doit être applicable à un ensemble de propriétés non connu à l'avance.

1.4 Synthèse

L'exemple de l'application BAGHERA permet de montrer que, de façon générale, les propriétés non-fonctionnelles des applications distribuées ont besoin d'être adaptées en fonction de l'environnement, et notamment en fonction des besoins des utilisateurs. Ce besoin d'adaptation induit à son tour un certain nombre de besoins concernant la plate-forme middleware.

Ce document propose une architecture *de plate-forme middleware dont l'objectif est de pouvoir satisfaire au mieux ces besoins. Plus précisément, les objectifs de cette architecture peuvent être résumés en disant qu'elle devra :*

- *permettre de séparer le code non-fonctionnel du code fonctionnel des applications ;*
- *permettre de séparer et de composer le code de chaque propriété non-fonctionnelle ;*
- *être modulaire et extensible, ce qui implique notamment un mécanisme de composition générique, non limité à un ensemble de propriétés fixé à l'avance.*

Le chapitre 2 présente les travaux existants autour de cette problématique. Il identifie leurs limitations, qui justifient le besoin de solutions nouvelles pour aller plus loin, telle que celle que nous proposons dans le chapitre 3.

Chapitre 2

État de l'art

Ce chapitre présente les travaux déjà effectués autour de la problématique présentée dans le chapitre précédent. Du fait que la plupart de ces travaux sont basés sur des langages ou des architectures réflexives, la première section de ce chapitre effectue quelques rappels sur la réflexivité en général. Ces rappels permettent de présenter, dans la deuxième section, les travaux de recherche passés ayant eu plus ou moins les mêmes objectifs que les nôtres. La dernière section est une synthèse de ces travaux. Elle identifie notamment leurs limitations vis-à-vis de nos objectifs. Ces limitations justifient le besoin de solutions nouvelles, telles que celle que nous proposons, pour essayer d'aller plus loin.

2.1 Rappels sur la réflexivité

Cette section présente brièvement le concept de réflexivité en général. Ce concept peut s'appliquer de différentes façons, sur des systèmes très divers, mais il serait trop long, et hors de propos ici, de présenter tous ces domaines d'application. Cette section se contente donc d'en présenter quelques uns seulement, à savoir ceux qui sont utilisés par les travaux de recherche présentés dans la section suivante.

2.1.1 Généralités

On admet généralement que le concept de réflexivité a été considéré pour la première fois dans toute sa généralité par Brian C. Smith en 1982, qui remarque, dans [Smi82], que :

> In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures.

Autrement dit, un processus réflexif est un processus qui « raisonne » sur lui-même, c'est à dire qui est capable d'observer et éventuellement de modifier en conséquence son propre comportement. De plus, une façon d'obtenir cette propriété est d'inclure dans un processus P un processus interne P' manipulant[1] une représentation formelle des opérations et des structures de P. Lorsque l'on parle « des opérations et des structures de P », on parle en fait des opérations et des structures du processus P *initial*, et non pas du processus obtenu après inclusion de P'. Autrement dit, un processus réflexif construit selon cette méthode ne raisonne pas sur une représentation complète de lui-même[2], mais seulement sur une partie, que l'on appelle le *niveau de base*, et qui correspond au processus P initial. Le *niveau méta* est la partie du processus réflexif qui raisonne sur les opérations et les structures du niveau de base. Ce niveau correspond au processus interne P'. On peut évidemment poursuivre cette construction, en introduisant un processus interne P" pour raisonner sur P', et ainsi de suite. Un niveau méta peut donc servir de niveau de base pour un autre niveau méta.

Deux opérations principales permettent la communication entre un niveau de base et son niveau méta :

- la *réification* consiste à construire une représentation formelle d'une structure ou d'une opération du niveau de base, implicite à ce niveau, de façon à ce qu'elle devienne manipulable par le niveau méta ;
- la *réflexion* consiste à répercuter sur le niveau de base les modifications effectuées par le niveau méta sur la représentation formelle du niveau de base, précédemment réifiée.

Le niveau méta manipule en général une représentation formelle des opérations *et* des structures du niveau base. Lorsqu'en fait il ne manipule qu'une représentation des opérations, on parle de réflexivité *comportementale*. A l'inverse, lorsqu'il ne manipule qu'une représentation des structures, on parle de réflexivité *structurale*.

La définition précédente du concept de réflexivité ne s'applique, en toute rigueur, qu'aux processus, puisqu'eux seuls peuvent « raisonner » (contrairement aux entités statiques comme les programmes ou les langages). Toutefois, par extension, on dit qu'un programme est réflexif si le processus correspondant l'est. De même, on dit qu'un langage est réflexif s'il est conçu pour pouvoir écrire facilement des programmes réflexifs. Enfin, on dit qu'un interpréteur ou une machine virtuelle sont réflexifs si le langage qu'ils interprètent l'est. Puisqu'on peut considérer un système d'exploitation ou une plateforme middleware comme des machines virtuelles, on peut également parler de système d'exploitation réflexif et de middleware réflexif.

2.1.2 Application aux langages orientés objets

Les langages orientés objets réflexifs permettent de programmer le niveau de base d'un processus réflexif à l'aide de classes et d'objets, comme les langages classiques, et le

[1] P' peut s'exécuter en parallèle avec P mais, le plus souvent, son exécution est entrelacée, implicitement ou explicitement, avec celle de P.

[2] quelque soit la méthode utilisée pour construire un processus réflexif, une auto-représentation complète est très difficile, voire impossible à obtenir.

niveau méta à l'aide de *méta-classes* ou de *méta-objets*. Nous présentons ici uniquement le principe de fonctionnement des langages basés sur l'utilisation de méta-objets.

Principe de fonctionnement

Les langages orientés objets réflexifs qui utilisent des méta-objets visent essentiellement une réflexivité comportementale. Autrement dit, ils réifient les *opérations* effectuées par le niveau de base. Mais ils ne les réifient pas toutes : ils se limitent au contraire aux opérations élémentaires liées au modèle objet, comme par exemple la lecture et l'écriture d'une valeur dans un champ, l'invocation d'une méthode, ou encore la création d'un objet.

Dans un langage classique, lorsqu'un objet appelle une méthode, la méthode appelée est exécutée directement. Dans le cas d'un langage utilisant des méta-objets, un appel de méthode sur un objet `o` se déroule selon un algorithme similaire à l'algorithme suivant (cf. figure 2.1 page suivante) :

- si l'objet `o` ne possède pas de méta-objet associé, alors la méthode est exécutée directement ;
- sinon, l'appel est réifié, c'est à dire transformé en un objet, manipulable par le méta-objet, et représentant cet appel (nom de la méthode appelée, nombre et valeurs des paramètres...) ;
- l'appel ainsi réifié est confié au méta-objet associé à `o`, en appelant une méthode que tout méta-objet doit fournir s'il veut pouvoir manipuler les appels de méthodes effectués sur l'objet de base. Si on suppose que cette méthode s'appelle `handleMethodCall`, cette étape consiste à appeler la méthode `handleMethodCall` du méta-objet avec en paramètre l'appel initial, réifié à l'étape précédente ;
- la méthode `handleMethodCall` peut alors effectuer n'importe quel traitement. Elle peut par exemple contrôler, et modifier le cas échéant, le nom et les paramètres de l'appel initial, via l'objet qui le représente. Elle peut ensuite, mais ce n'est pas obligatoire, répercuter ces changements sur le niveau de base, en demandant à ce que l'opération inverse de la réification, i.e. la réflexion, soit effectuée. Concrètement cette opération est généralement effectuée en appelant une méthode prédéfinie, appelée par exemple `reflectMethodCall`, et prenant en argument un appel de méthode réifié. Cette méthode effectue l'appel demandé sur l'objet de base (sans provoquer une nouvelle réification de cet appel, sans quoi l'algorithme ne se terminerait pas).

Le même principe est utilisé pour les autres opérations qui sont réifiées par le langage considéré, les méthodes `handleMethodCall` et `reflectMethodCall` étant remplacées par des méthodes comme `handleFieldAccess` et `reflectFieldAccess`, ou encore `handleObjectCreation` et `reflectObjectCreation`. Ces méthodes, qui permettent la communication entre le niveau de base et le niveau méta, forment ce que l'on appelle le protocole à méta-objets (ou MOP, pour *meta object protocol*) d'un langage réflexif.

Les langages Iguana [GC96] (basé sur C++) et Actalk (basé sur SmallTalk), ainsi que les langages Dalang [WS98], Guarana [OB99] et MetaXa [GK98] (basés sur Java) sont des exemples de langages réflexifs utilisant des méta-objets.

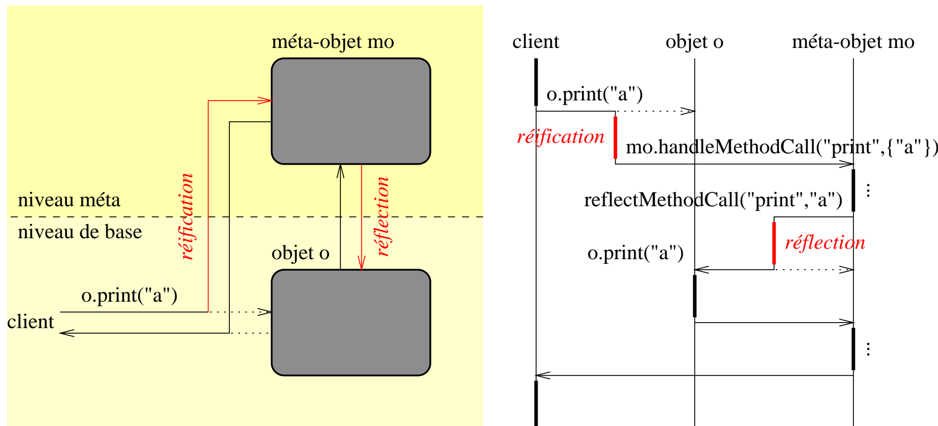


FIG. 2.1 – Principe de fonctionnement d'un méta-objet

Exemple d'utilisation

L'intérêt principal des méta-objets est qu'ils permettent de séparer assez facilement le code fonctionnel et le code non-fonctionnel d'une application (le premier correspondant au niveau de base et le second au niveau méta) comme le montre les exemples suivants :

- pour synchroniser les méthodes d'un objet, on peut lui associer un méta-objet qui, avant de réfléchir vers le niveau de base les appels réifiés qu'il reçoit, attend que certaines conditions soit vérifiées (comme par exemple qu'un certain verrou soit libre) ;
- pour rendre un objet persistant, on peut lui associer un méta-objet qui, après avoir réfléchi vers le niveau de base les accès en écriture réifiés qu'il reçoit, enregistre l'objet de base sur un support persistant ;
- pour protéger un objet, on peut lui associer un méta-objet qui, avant de réfléchir vers le niveau de base les appels réifiés qu'il reçoit, vérifie que l'objet appelant a bien les droits nécessaires pour faire cet appel. Si ce n'est pas le cas, le méta-objet ne réfléchi tout simplement pas l'appel réifié vers l'objet de base, ce qui revient à faire échouer l'appel.

2.1.3 Composition de méta-objets

Les exemples de la section précédente montrent que l'on peut associer *une* propriété non-fonctionnelle à un objet en lui associant *le* méta-objet mettant en œuvre cette propriété. Pour pouvoir associer plusieurs propriétés à un même objet, tout en gardant le code de chaque propriété regroupé dans son propre méta-objet, c'est à dire séparé des autres, il faut pouvoir *composer* plusieurs méta-objets. A cette fin, plusieurs techniques de composition de méta-objets ont été proposées[3].

[3]Puisque les méta-objets sont des objets, on peut naturellement utiliser les techniques de composition d'objets classiques, à savoir l'héritage et la délégation, pour composer des méta-objets. Les méthodes

Méta-espaces

Une première méthode consiste à confier le traitement de chaque type d'opération du niveau de base à des *méta-espaces* séparés et indépendants, chaque méta-espace pouvant recevoir son propre méta-objet. Par exemple, pour un langage qui réifierait les appels de méthodes et les accès aux champs, cette technique consisterait à diviser le niveau méta d'un objet en deux méta-espaces :

- le méta-espace `execution` recevrait un méta-objet fournissant la méthode `handleMethodCall` (en reprenant les notations de la section précédente) ;
- le méta-espace `state` recevrait quant à lui un méta-objet fournissant la méthode `handleFieldAccess`, ce méta-objet pouvant être différent du précédent.

Un tel langage permettrait de composer facilement les méta-objets de persistance et de protection mentionnés dans les exemples de la section précédente, en plaçant le premier dans le méta-espace `state` et le second dans le méta-espace `execution`. Il serait en revanche plus difficile de composer les méta-objets de synchronisation et de protection, puisqu'il faudrait pour cela les placer tous les deux dans le même méta-espace.

Pour réduire la probabilité d'occurrence de tels conflits, il faut décomposer le niveau méta en un plus grand nombre de méta-espaces. Le langage CodA [McA95b], basé sur SmallTalk, utilise ainsi sept méta-espaces différents (cf. section 2.2.1 page 19) : `State` prend en charge les accès aux champs de l'objet de base, `Send` les appels de méthodes sortants, `Accept` les appels de méthodes entrants, `Queue` stocke les appels acceptés mais par encore traités, et `Receive` traite les appels stockés, en utilisant `Protocol` pour trouver le code à exécuter en fonction du nom de la méthode appelée, puis `Execution` pour exécuter ce code.

Listes ou arbres de méta-objets

L'utilisation de méta-espaces permet de composer des méta-objets prenant en charge des opérations de base de nature différentes. Les autres méthodes de composition proposées s'appliquent au contraire à la composition de méta-objets de même nature. Elles sont complémentaires avec le concept de méta-espace : on peut en effet les utiliser, en plus des méta-espaces, pour composer plusieurs méta-objets dans un même méta-espace.

L'une des méthodes les plus simples consiste à composer les méta-objets en une liste ordonnée (cf. figure 2.2 page suivante). Lorsqu'une opération de base est réifiée, elle est confiée au premier méta-objet de la liste. Puis, lorsque celui-ci réfléchit l'appel vers le niveau de base, l'appel réifié est confié au prochain méta-objet dans la liste. Et ainsi de suite jusqu'au dernier méta-objet de la liste, où l'appel réifié est finalement réfléchi pour de bon vers le niveau de base.

La sémantique obtenue par une telle composition de méta-objets peut dépendre de l'ordre des méta-objets dans la liste. Cet ordre est donc important. La plupart des langages, comme Dalang [WS98] et MetaXa [GK98], laissent au programmeur le soin

---

proposées spécifiquement pour composer des méta-objets ne sont d'ailleurs que des variantes de ces techniques, mais qui sont plus faciles à utiliser car elles sont intégrées au MOP.

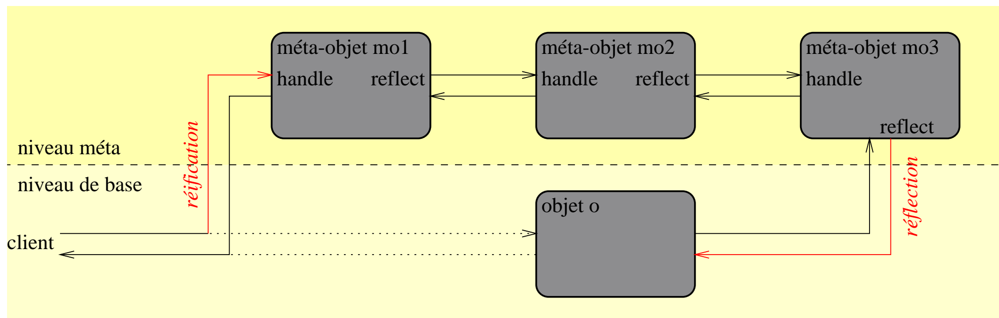


Fig. 2.2 – Composition de méta-objets en une liste

de placer les méta-objets dans le « bon » ordre. A-TOS[4] [PDF99] essaye d'automatiser cette tâche, mais la solution proposée ne s'applique que sous certaines hypothèses simplificatrices.

Le langage Guarana permet de composer les méta-objets de façon arbitraire : en liste, en arbre, de façon séquentielle ou parallèle... Il utilise pour cela deux types de méta-objets : les méta-objets normaux traitent les opérations réifiés du niveau de base, alors que les méta-objets *composeurs* s'occupent de déléguer le traitement d'une opération réifiée à un ou plusieurs méta-objets, qui peuvent être eux-mêmes des composeurs. Le programmeur pouvant définir ses propres composeurs, il peut ainsi composer les méta-objets de façon arbitraire. Enfin, pour ne pas brider les possibilités de composition, Guarana utilise un MOP plus souple que ceux de la plupart des autres langages.

2.1.4 Programmation orientée aspects

Bien que cette première partie de chapitre soit consacrée à des rappels sur la réflexivité, nous présentons ici brièvement la programmation orientée aspects, domaine a priori distinct de la réflexivité. Et ce d'une part parce que cette technique est utilisée dans certains travaux de recherche mentionnés dans la seconde partie du chapitre, et d'autre part parce que la réflexivité est l'une des façons de mettre en œuvre ce type de programmes.

Le but de la programmation par aspects [KLM+97] est de séparer la programmation des différents *aspects* d'une application, les aspects étant justement toutes les « fonctions » d'une application qui ne peuvent être séparées des autres dans le code source du programme à l'aide des concepts de programmation habituels comme les procédures, les méthodes ou les classes. Les propriétés non-fonctionnelles d'une application sont en général des aspects, car il est difficile d'isoler le code de ces propriétés dans des méthodes ou des classes séparées des autres.

Un programme orienté aspects est constitué d'un programme de base, qui définit

[4] A-TOS est basé sur TOS [Paw99], un langage réflexif utilisant des méta-classes et non pas des méta-objets. Cependant, A-TOS utilise TOS pour définir des *wrappers*, qui peuvent être assimilés à des méta-objets.

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, appelé *tisseur d'aspects* (*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 [LK98], 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 [LK98], une extension de Java, appartient à cette dernière catégorie.

Comme l'indique [KLM+97], et comme le montrent [AT98], [Lun98] et [BS99], la programmation par aspects est très proche de la réflexivité. Plus précisément, la composition de méta-objets peut servir de support pour mettre en œuvre la composition d'aspects. Dans ce cas, les fonctionnalités de l'application sont programmées au niveau de base, et les aspects sont réalisés sous forme de méta-objets, et composés en utilisant l'une des techniques de composition précédentes.

2.2 État de l'art

Cette section présente, dans un ordre à peu près chronologique mais sans prétendre à l'exhaustivité, les principaux travaux de recherche ayant eu à peu près les mêmes objectifs que les nôtres ou qui, bien qu'ayant poursuivi d'autres objectifs, proposent des solutions qui peuvent s'appliquer aux nôtres.

2.2.1 La plate-forme Tj

McAffer a été l'un des premier à proposer, dans [McA95a], d'utiliser la réflexivité pour placer au niveau méta le code lié à la distribution. Cette séparation permet au programmeur de tester différents modèles de distribution, sans avoir à changer ou presque le code de son application, situé au niveau de base.

Architecture

L'architecture proposée par McAffer pour pouvoir placer le code lié à la distribution au niveau méta est celle du langage CodA [McA95b], une extension réflexive de SmallTalk. Ce langage associe à chaque objet un niveau méta composé par défaut[5] de sept méta-espaces, contenant chacun un méta-composant (un méta-composant peut appartenir à plusieurs méta-espaces à la fois). Les rôles de ces méta-composants sont les suivants (cf. figure 2.3 page suivante) :

- `Send` définit comment l'objet de base gère les messages (i.e. les appels de méthode) sortants ;
- `Accept` interagit avec le méta-espace `Send` de l'appelant pour déterminer si un message est valide et s'il concerne l'objet de base ;

[5] le programmeur peut définir ses propres méta-espaces.

– `Queue` organise et stocke les messages qui ont été acceptés mais pas encore reçus ou traités. Ce méta-espace permet de découpler l'émetteur et le receveur ;
– `Receive` définit l'ordre dans lequel les messages acceptés ou placés en file d'attente sont traités ;
– `Protocol` recherche le code à exécuter pour traiter un message (en utilisant, par exemple, les règles de l'héritage de classe concernant la surcharge de méthode) ;
– `Execution` définit comment l'objet de base exécute ses méthodes ;
– `State` définit le format de l'état de l'objet de base, ainsi que la façon de consulter et de modifier cet état.

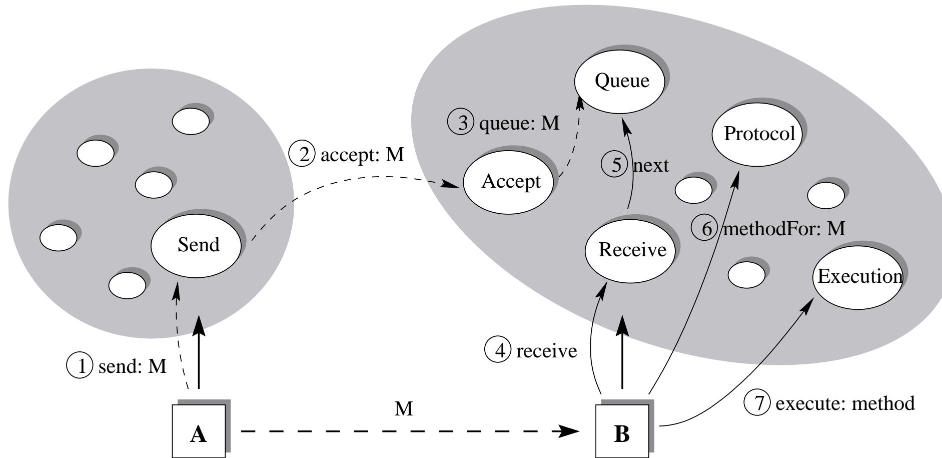


Fig. 2.3 – Organisation du niveau méta dans CodA (tirée de [McA95a])

La première application de cette architecture générale consiste à implanter les appels de méthode à distance au niveau méta. McAffer utilise pour cela, coté client, un talon appelé `RemoteReference` et dont le méta-composant `Accept` est modifié de façon à empaqueter les messages reçus, puis à les envoyer à l'objet serveur par le réseau. Coté serveur, les messages reçus sont « dépaquetés et acceptés par l'objet serveur »[6].

Applications

La plate-forme Tj [McA95a] est une application de l'architecture précédente. Cette plate-forme permet de gérer au niveau méta trois propriétés non-fonctionnelles : l'empaquetage des messages, la duplication et la mobilité.

Dans Tj, l'empaquetage des messages, initié par le méta-composant `Accept` des objets `RemoteReference`, est géré par un mécanisme générique paramétré par des descripteurs d'empaquetage (*marshaling descriptors*). Un descripteur définit la façon dont un objet doit être empaqueté : par référence, par copie (en profondeur ou non), en

[6] l'auteur n'indique pas clairement ce qui se passe coté serveur : est-ce que le méta-composant `Accept` de l'objet serveur est modifié pour dépaqueter les messages acceptés, ou bien cette opération est-elle effectuée avant, par un objet intermédiaire ?

faisant migrer l'objet, en le dupliquant[7]... Ces descripteurs peuvent être associés aux objets (dans un nouveau type de méta-composant appelé `Marshaling`), ou bien aux paramètres des appels de méthode (via des « annotations », que l'on peut considérer comme des paramètres supplémentaires). Dans le dernier cas, ils sont donc mélangés au code fonctionnel de l'application.

La duplication d'un objet est gérée par un nouveau type de méta-composant, appelé `Replication`, et associé aux différentes copies de l'objet. Ce méta-composant, différent de `State`, interagit néanmoins avec lui afin d'intercepter les modifications de l'état de l'objet de base. Chaque modification de l'état d'une des copies est diffusée puis répercutée sur les autres copies, selon un protocole de cohérence qui dépend du méta-composant `Replication` choisi.

La migration est gérée, comme l'empaquetage des messages, par un mécanisme générique paramétré par des descripteurs. Ces descripteurs peuvent être associés aux objets (dans un nouveau type de méta-composant appelé `Migration`), ou bien aux paramètres des appels de méthode (via le système d'annotations utilisé pour l'empaquetage).

Discussion

La plate-forme Tj permet donc de placer au niveau méta le code lié à l'empaquetage, à la duplication et à la migration des objets. Ceci est essentiellement du à l'utilisation du langage CodA pour réaliser la plate-forme ainsi que les applications. Ce choix présente en effet plusieurs avantages :

- le niveau méta est décomposé en un nombre assez important de méta-espaces, ce qui permet de mieux séparer et de mieux composer les méta-composants que s'il n'y avait qu'un ou deux méta-espaces ;
- le nombre de méta-espaces est extensible (cf. ci-dessous), ce qui permet d'ajouter de nouveaux méta-espaces pour gérer les propriétés non-fonctionnelles nouvelles, non prévues au départ dans CodA (comme la duplication).

Mais le choix d'utiliser CodA de façon systématique, pour la plate-forme comme pour les applications, entraîne un surcoût à l'exécution sur tous les objets, alors que seuls certains d'entre eux utilisent réellement le niveau méta. Les performances obtenues sont donc probablement faibles (bien que McAffer ne donne pas de mesures de performances, il indique que sa plate-forme est adéquate pour *prototyper* des applications).

Le mécanisme qui permet d'ajouter de nouveaux méta-espaces est intéressant, mais il n'est malheureusement pas clair. Par exemple, comment fait-on pour que le nouveau méta-espace `Replication`, qui ne fait pas partie des sept méta-espaces par défaut, puisse intercepter les accès aux champs de l'objet de base ? [McA95b] indique simplement que le niveau méta est « manipulé » pour intercepter ces opérations, sans qu'il y ait besoin de modifier le méta-composant `State` (qui intercepte également ces opérations). D'après [McA93] et [McA95b], cette manipulation ne semble pas être faite à l'aide des mécanismes propres de CodA, mais plutôt en modifiant directement la mise en œuvre du langage en SmallTalk (ce qui ne serait pas une solution générique).

[7] la duplication est différente de la simple copie : les différentes copies d'un objet dupliqué sont maintenues en cohérence.

2.2.2 Singhai et al.

Singhai et al. proposent, dans [SSC97], d'utiliser la réflexivité pour construire une plate-forme modulaire et extensible. L'architecture réflexive qu'ils proposent, volontairement minimale, a été conçue pour permettre d'ajouter des contraintes temps-réel aux appels de méthodes à distance. Bien que conçue uniquement pour le temps-réel, cette architecture permet aussi d'ajouter des propriétés de tolérance aux fautes et d'équilibrage de charge.

Architecture

L'architecture proposée dans [SSC97] concerne essentiellement les liaisons entre un objet client et un objet serveur. Comme dans CORBA, ces liaisons sont constituées d'un talon coté client, et d'un squelette coté serveur. Cependant, au lieu d'être constitués d'un seul objet, comme c'est le cas en CORBA, ces talons et squelettes sont en fait composés de plusieurs objets.

Un talon est constitué d'un objet principal, dont le code est généré par le compilateur de talons, et de deux objets auxiliaires, qui fournissent chacun une partie des fonctions du talon (cf. figure 2.4). Le premier objet auxiliaire, de type `Marshaler`, prend en charge l'empaquetage des requêtes et le dépaquetage des réponses. Le deuxième objet auxiliaire, de type `Invoker`, prend en charge l'envoi des requêtes. Finalement, l'objet principal appelle les méthodes de ses objets auxiliaires dans l'ordre approprié. On peut considérer les deux objets auxiliaires comme deux méta-objets de l'objet principal, placés dans les deux méta-espaces de cet objet (similaires aux méta-espaces `Marshaling` et `Send` utilisés dans Tj).

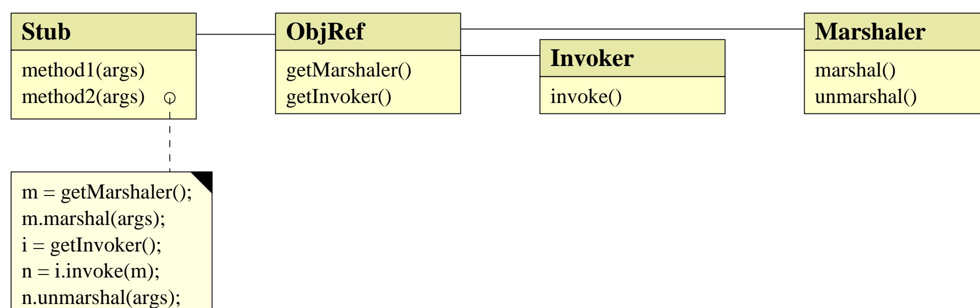


FIG. 2.4 – Composition des talons proposée par Singhai et al. (tirée de [SSC97])

Un squelette est également constitué d'un objet principal, et d'un objet auxiliaire de type `Dispatcher`[8] (cf. figure 2.5 page ci-contre). Cet objet auxiliaire est chargé de la politique d'ordonnancement des requêtes reçues.

L'intérêt de décomposer les talons et squelettes en plusieurs objets est que l'on peut changer les objets auxiliaires sans modifier les objets principaux, ni donc le compilateur

[8]curieusement, il ne semble pas y avoir d'objet auxiliaire correspondant à l'objet auxiliaire `Marshaler` utilisé coté client.

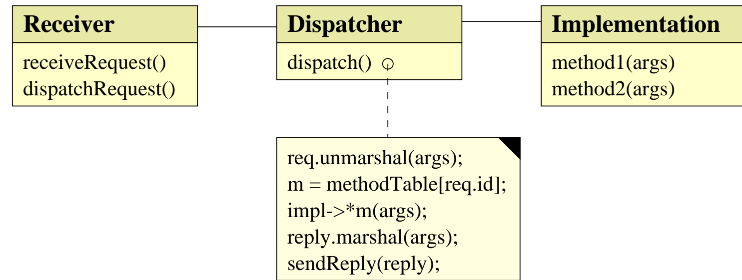


Fig. 2.5 – Composition des squelettes proposée par Singhai et al. (tirée de [SSC97])

qui les génère. Cela permet d'étendre les fonctions offertes par la plate-forme sans avoir à modifier et recompiler le code existant.

Applications

Singhai et al. montrent trois applications de leur architecture. La première application, qui a motivé la définition de l'architecture, consiste à réaliser des invocations de méthodes avec contraintes temps-réel. Il suffit pour cela de définir des sous-classes des classes `Marshaler`, `Invoker` et `Dispatcher` (cf. figure 2.6) :

- la classe `Marshaler` est étendue pour pouvoir ajouter une échéance dans les requêtes empaquetées ;
- la classe `Invoker` est étendue pour ajouter l'échéance appropriée dans chaque requête, avant de l'envoyer sur le réseau ;
- la classe `Dispatcher` est étendue pour tenir compte de l'échéance associée à chaque requête, et pour ordonnancer les requêtes en conséquence. Cet ordonnancement est d'ailleurs confié à un « sous-objet » auxiliaire, de type `Scheduler`.

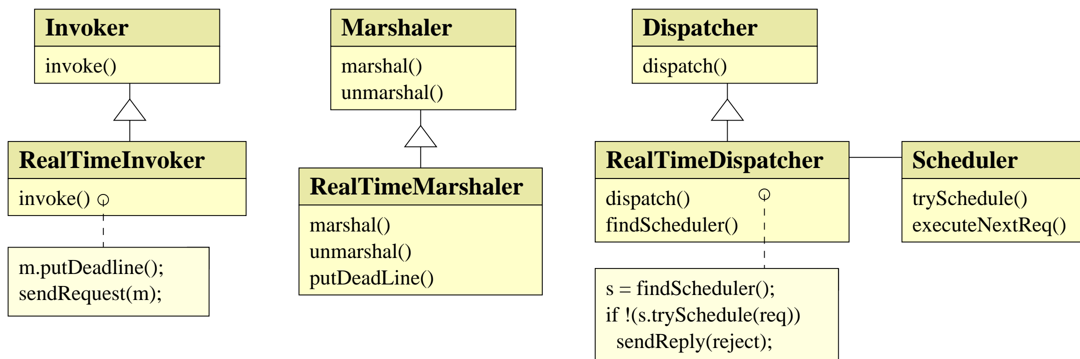


Fig. 2.6 – Implantation des contraintes temps-réel dans [SSC97]

Une deuxième application de cette architecture consiste à ajouter un algorithme d'« équilibrage de charge ». Pour cela, les auteurs utilisent une sous-classe `Invoker` qui

envoie chaque requête vers le serveur le moins chargé au moment de l'appel (le serveur en question est choisi parmi un groupe de serveurs équivalents).

Finalement, une troisième application consiste à ajouter un algorithme de « tolérance aux fautes ». Les auteurs utilisent pour cela une sous-classe de `Invoker`, qui cette fois diffuse chaque requête à plusieurs serveurs équivalents, pour minimiser les risques d'échec en cas de panne d'un des serveurs.

Discussion

Comme Tj, la plate-forme proposée par Singhai et al. utilise la réflexivité pour séparer au niveau méta le code lié à certaines propriétés non fonctionnelles, ici le temps réel, la tolérance aux fautes et l'équilibrage de charge. Mais, au lieu d'être utilisée de façon systématique, la réflexivité est utilisée de façon minimale, uniquement sur les talons et squelettes. Cette approche présente plusieurs avantages par rapport à l'utilisation d'un langage réflexif comme CodA :

- l'architecture peut être facilement mise en œuvre dans un langage quelconque, alors que Tj est fortement liée à CodA ;
- la réflexivité est implantée à l'aide des mécanismes usuels de la programmation orienté objet : il n'y a donc même pas besoin d'être familier avec les concepts de la réflexivité pour programmer ou étendre la plate-forme ;
- la restriction de l'utilisation de la réflexivité aux talons et squelettes n'impose pas de surcoûts sur les autres objets de la plate-forme et de l'application. De plus, le surcoût dans les talons et squelettes est extrêmement faible, surtout par rapport à la durée d'un appel de méthode à distance. Les performances sont donc pratiquement inchangées par rapport à une plate-forme non réflexive.

Mais cette approche a également des inconvénients. Par exemple, contrairement à Tj, il n'est pas possible d'ajouter de nouveaux méta-espaces sans modifier les classes `Stub` et `Receiver` et le générateur de code correspondant. Ce qui limite l'extensibilité de la plate-forme (elle est néanmoins relativement extensible : on pourrait probablement facilement ajouter des fonctions de surveillance (*monitoring*), de migration...). Le nombre de méta-espaces utilisé, bien plus petit que dans CodA, provoque également plus de « conflits », c'est à dire de situations où plusieurs propriétés doivent être mises en œuvre dans le même méta-espace (par exemple, la tolérance aux fautes et l'équilibrage de charge interviennent au niveau de la classe `Invoker`). Or les auteurs ne proposent pas de mécanisme pour composer plusieurs méta-objets dans un même méta-espace. En fait, ils ne semblent pas s'intéresser au problème de la composition de propriétés non-fonctionnelles.

2.2.3 Entreprise Java Beans

Le modèle *Enterprise Java Beans* (EJB) [EJB] ne résulte pas d'un travail de recherche à proprement parler. Nous le présentons néanmoins car l'architecture qu'il propose permet de séparer le code non-fonctionnel du code fonctionnel des applications. De plus, ce modèle a l'avantage d'être très répandu et très utilisé.

Architecture

Selon le modèle EJB, une application distribuée est constituée de *composants* (ou *Beans*), qui sont en fait essentiellement des objets (au sens où ils n'ont qu'un seul point d'entrée - cf. section 3.1.1 page 36). A l'exécution, ces composants sont encapsulés dans des *conteneurs*, eux-mêmes inclus dans des *serveurs* EJB. Conteneurs et serveurs font partie de la plate-forme middleware :

- un serveur est un environnement d'exécution pour ses conteneurs. Un serveur se présente généralement sous la forme d'un processus, et il fournit des services système pour ses conteneurs, comme par exemple un service de persistance, un moniteur transactionnel, ou un service d'authentification ;
- un conteneur encapsule un ou plusieurs composants, et gère les propriétés non-fonctionnelles de ces composants.

Plus précisément, un conteneur est constitué, au minimum, des objets suivants, accessibles à distance :

- l'usine à composants, ou *EJB Home*, permet de créer de nouveaux composants à l'intérieur du conteneur, de retrouver des composants existants, et de détruire des composants. Il y a une usine à composants par conteneur et par type de composant ;
- les objets d'interposition, ou *EJB Objects*, permettent d'accéder aux composants encapsulés à l'intérieur du conteneur. En effet, ces composants ne sont jamais accédés directement depuis l'extérieur, mais toujours en passant par un objet d'interposition (d'où son nom). Il y a un objet d'interposition par composant encapsulé (sauf éventuellement pour les composants dit « sans état », ou *stateless bean*).

Un objet d'interposition[9] a avant tout un rôle équivalent à celui d'un squelette dans CORBA : il dépaquette les messages d'invocation à distance reçus sur le réseau, invoque la méthode correspondante du composant encapsulé, empaquette le résultat de cet appel, puis le retourne dans un message de réponse. Cependant, et c'est là qu'il se différencie d'un squelette CORBA, un objet d'interposition peut effectuer certaines opérations juste avant et juste après avoir invoqué la méthode sur le composant. Cette possibilité permet de l'assimiler à un méta-objet du *Bean* associé.

Le modèle EJB prévoit que le code des objets d'interposition, ainsi que celui des usines à composants, soit généré automatiquement à partir d'un fichier de déploiement. Ce fichier de déploiement, écrit par l'utilisateur en fonction de ses besoins, avant de déployer son application, indique par exemple que pour la méthode `transfer` des composants de type `Account`, il faut être dans le cadre d'une transaction, alors que pour la méthode `getBalance` ce n'est pas nécessaire. Ce fichier est utilisé lors du déploiement par un générateur de code, pour produire la classe des objets d'interposition. Dans l'exemple précédent, le code produit pour la méthode `getBalance` de l'objet d'interposition se contenterait d'appeler la méthode `getBalance` sur le composant, alors que le code produit pour la méthode `transfer` ferait des traitements supplémentaires pour gérer les transactions.

[9] et le squelette associé si ces objets sont séparés.

La figure 2.7 résume l'architecture du modèle EJB (se référer à [EJB] pour plus de détails sur cette architecture, mais aussi sur les autres aspects du modèle EJB : interfaces de programmation, cycle de vie...).

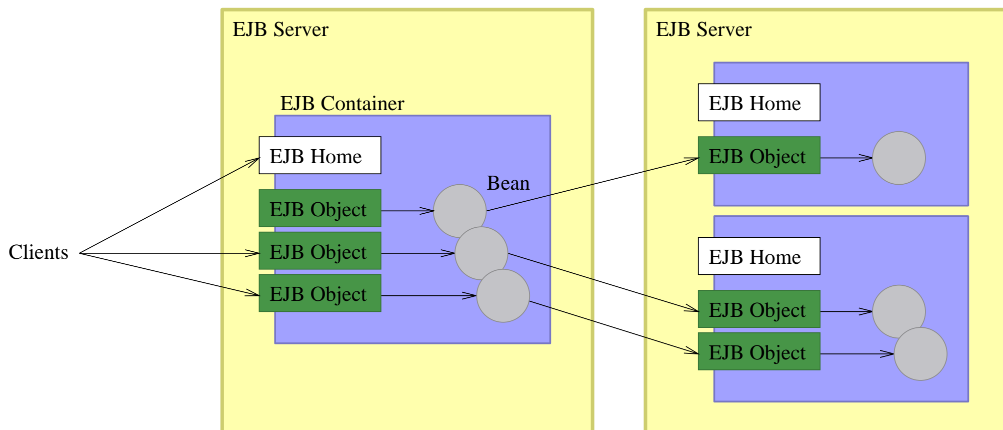


FIG. 2.7 – Architecture générale du modèle EJB

Applications

Le modèle EJB est actuellement utilisé pour séparer du code fonctionnel le code de trois propriétés non-fonctionnelles : persistance, transactions et protection. Comme nous l'avons dit, ce sont les objets d'interposition, assimilables à des méta-objets, qui permettent cette séparation. L'algorithme qu'ils utilisent pour traiter un appel de méthode à distance est approximativement le suivant :

- vérifier si l'appelant a bien le droit d'appeler cette méthode, le cas échéant ;
- démarrer une transaction si la méthode doit s'exécuter dans un contexte transactionnel et qu'aucune transaction n'est en cours ;
- charger le composant appelé en mémoire, s'il ne s'y trouve pas déjà (pour un composant persistant) ;
- appeler la méthode sur le composant ;
- enregistrer le composant sur support persistant (pour un composant persistant, et uniquement si la méthode appelée a modifié l'état de ce composant) ;
- terminer la transaction en cours si celle-ci a été démarrée par l'objet d'interposition lui-même.

Discussion

La plate-forme EJB a les mêmes avantages que celle proposée par Singhai et al., du fait qu'elle utilise la réflexivité de façon très limitée : les performances, notamment, sont quasiment identiques à celles des plates-formes non-réflexives. Cela permet aussi de réaliser la plate-forme avec un langage très répandu, ici Java. Ce qui permet, mais n'assure pas pour autant, une diffusion et une utilisation très large de la plate-forme.

Malheureusement, l'utilisation d'un générateur de code monolithique, qui produit des objets d'interposition (i.e. des méta-objets) performants mais eux aussi monolithiques, limite beaucoup l'extensibilité des plates-formes EJB (pour gérer une propriété non-fonctionnelle nouvelle, il faut réécrire le générateur de code). Une utilisation un peu moins limitée des techniques réflexives permettrait probablement de résoudre le problème. On pourrait par exemple utiliser les techniques de composition de méta-objets, c'est à dire remplacer les objets d'interposition par des listes d'objets plus simples, ne gérant qu'une seule propriété à la fois, et générés chacun par leur propre générateur de code.

2.2.4 FlexiNet

Comme Singhai et al., Hayton et al. proposent, dans [HHD98], d'utiliser la réflexivité pour construire une plate-forme modulaire et extensible, appelée FlexiNet. Ils proposent également, eux aussi, d'utiliser la réflexivité de façon limitée, uniquement dans les talons et squelettes. La différence principale avec Singhai et al. est qu'ils proposent de composer les méta-objets en listes plutôt qu'avec des méta-espaces.

Architecture

Comme dans Singhai et al., l'architecture proposée dans [HHD98] concerne essentiellement les liaisons client-serveur, qui sont constituées, là encore, d'un talon et d'un squelette, eux mêmes composés de plusieurs objets. La différence vient de la méthode de composition utilisée.

Un talon est composé, dans FlexiNet, d'une pile d'objets qui, à part le premier, peuvent être considérés comme des méta-objets (cf. figure 2.8 page suivante) :

- le talon à proprement parler transforme chaque appel de méthode effectué sur lui en une représentation concrète de cet appel, sous forme d'objets. Autrement dit, ce talon réifie les appels de méthodes qu'il reçoit. Il confie ensuite le traitement de ces appels réifiés aux objets suivants dans la pile, qui peuvent être considérés comme ses méta-objets, organisés en une liste ;
- les premiers méta-objets de cette liste disposent d'une représentation typée, facilement manipulable, de l'appel initial. Ils peuvent donc en profiter pour valider ou modifier les arguments de l'appel ou le nom de la méthode appelée (par exemple pour adapter l'interface utilisée par le client à celle offerte par le serveur) ;
- à partir d'un certain niveau, l'appel réifié est pris en charge par des méta-objets qui fournissent des protocoles de communication. L'appel réifié est sérialisé par un premier méta-objet, puis éventuellement compressé, chiffré... par les méta-objets suivants, avant d'être finalement envoyé sur le réseau par le dernier méta-objet de la liste.

Les squelettes sont également constitués d'une pile d'objets, qui peuvent également être considérés comme des méta-objets, organisés en une liste dont l'ordre est l'inverse de celui de la pile de protocoles (i.e. le premier méta-objet de la liste correspond au protocole de plus bas niveau) :

- les premiers méta-objets décompressent, déchiffrent..., le cas échéant, les messages reçus sur le réseau. Ces messages sont finalement désérialisés, et deviennent alors des appels de méthode réifiés facilement manipulables ;
- les méta-objets suivants peuvent effectuer des traitements de haut niveau sur ces appels réifiés : vérification des droits de l'appelant, lancement d'une transaction le cas échéant...
- le dernier méta-objet de la liste correspond à ce que l'on pourrait appeler le squelette proprement dit. Ce méta-objet se contente de réfléchir l'appel réifié vers le niveau de base, c'est à dire d'appeler la méthode appropriée sur l'objet serveur.

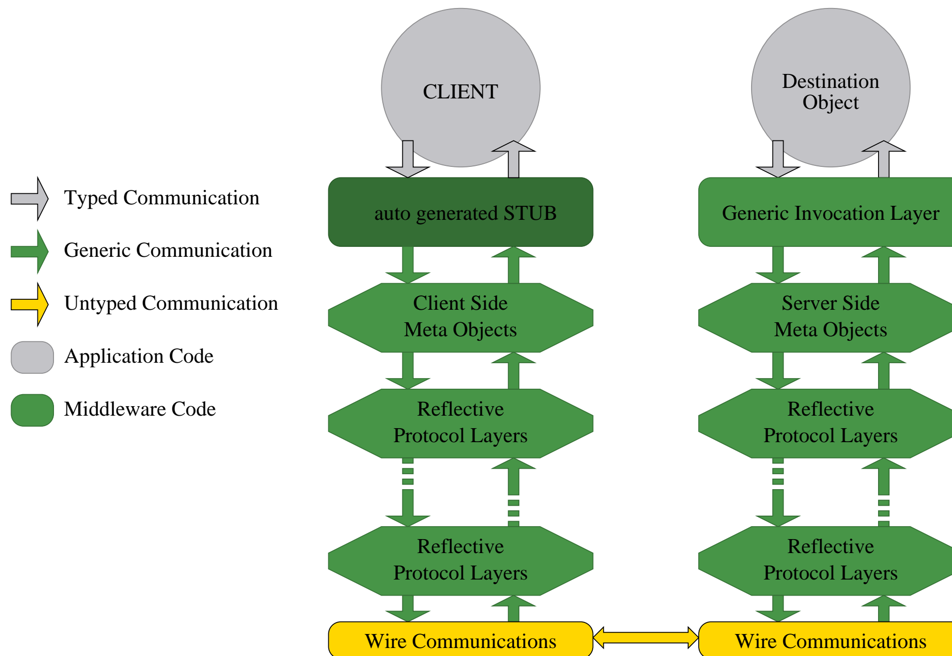


FIG. 2.8 – Architecture des liaisons dans FlexiNet (tirée de [HHD98])

Applications

Le système à objets mobiles *Mobile Object Workbench* [HBDH98], construit à l'aide de la plate-forme FlexiNet, montre que l'architecture de cette plate-forme permet de fournir un service de mobilité au niveau méta, et donc séparément du code fonctionnel. Ce système à objets mobiles est en effet mis en œuvre en insérant deux méta-objets, coté client et coté serveur, entre le talon et le squelette proprement dit et les méta-objets d'une liaison normale :

- coté client, le méta-objet inséré entre le talon et les méta-objets normaux envoie chaque message vers la dernière localisation connue de l'objet serveur. Si ce dernier ne se trouve pas à cet endroit, une erreur est retournée par le site destinataire. Le méta-objet contacte alors un serveur de localisation pour déterminer la position

courante de l'objet serveur, puis renvoie le message vers cette destination ;
- coté serveur, le méta-objet inséré entre le squelette et les méta-objets normaux vérifie que l'objet serveur est bien présent. Si ce n'est pas le cas, il renvoie un message d'erreur. Sinon, il verrouille l'objet serveur avant de confier le traitement de l'appel réifié au squelette, de façon à ce que cet objet ne migre pas pendant qu'une de ses méthodes est en cours d'exécution.

Discussion

Là encore, l'utilisation limitée de la réflexivité, i.e. dans les talons et les squelettes uniquement, offre l'avantage de pouvoir mettre en œuvre l'architecture dans le langage de son choix, et permet d'obtenir des performances quasiment identiques à celles des plates-formes non-réflexives (FlexiNet est aussi efficace que Java RMI et OrbixWeb, d'après [HHD98]).

Le fait d'utiliser des listes de méta-objets, au lieu d'un petit nombre de méta-espaces comme le proposent Singhai et al., offre même à la plate-forme FlexiNet des avantages supplémentaires. Tout d'abord, cela ne l'empêche pas de pouvoir séparer au niveau méta le code non-fonctionnel, comme le montre l'exemple de la migration. Mais cela lui permet, en plus, de séparer et de composer plus facilement le code de chaque propriété non-fonctionnelle. En effet, bien qu'il n'y ait pas eu, à notre connaissance, d'autres expérimentations que la migration, il nous semble évident que, puisqu'un objet d'interposition EJB peut gérer la persistance, la protection et les transactions, un squelette de FlexiNet doit pouvoir en faire autant, et même plus (en séparant chaque propriété dans un méta-objet dédié).

Malheureusement, l'architecture de FlexiNet est essentiellement limitée au modèle client-serveur (comme celle proposée par Singhai et al.). De plus, il n'y a pas de concept de composant ni de conteneur, permettant d'associer une propriété à plusieurs objets à la fois (il semble toutefois possible d'ajouter un tel concept au-dessus de FlexiNet, d'après [HBDH98]).

2.2.5 Blair et al.

Blair et al. proposent, comme Singhai et al. et Hayton et al., d'utiliser la réflexivité pour construire une plate-forme middleware modulaire et extensible. Mais l'architecture qu'ils proposent est très différente de celles présentées ci-dessus. La réflexivité y est en effet appliquée de façon plus systématique et plus « propre ».

Architecture

L'architecture proposée dans [BCRP98] est en fait plus un modèle objet réflexif qu'une architecture de plate-forme middleware. Cette architecture repose en effet sur un modèle objet, en l'occurrence ODP [ODP95b, ODP95a], et définit pour ce modèle un niveau méta composé de trois méta-espaces.

Avant de présenter l'architecture du niveau méta, rappelons brièvement les concepts du modèle ODP :

– un *objet* est une entité indépendante regroupant du code, des données, des activitées, des services et des interfaces d'accès à ces services. Un objet peut être composé de plusieurs objets internes ;
– une *interface* est un point d'accès à un objet. Les interfaces « calculatoires » (*computational interfaces*) possèdent un type (i.e. un ensemble de méthodes) et un sens (client ou serveur). Il existe aussi des interfaces *stream* et des interfaces de signalisation (communication par événements asynchrones) ;
– une *liaison* relie un nombre arbitraire d'interfaces d'objets différents. Une liaison *primitive* relie deux interfaces de même type et de sens complémentaire. Une liaison *composée* peut relier un nombre arbitraire d'interfaces, par l'intermédiaire d'un *objet de liaison* et de plusieurs interfaces primitives. Un objet de liaison est un objet comme les autres : il peut donc avoir plusieurs interfaces, et être composé d'objets internes. Il se distingue des objets normaux par son rôle, qui est de permettre aux objets normaux de communiquer entre eux.

Blair et al. proposent, dans [BCRP98][10], de compléter ce modèle en ajoutant à chaque objet un méta-espace appelé `composition` et, à chaque interface de chaque objet, deux méta-espaces qu'ils appellent `encapsulation` et `environment` (cf. figure 2.9 page suivante) :
– le méta-espace `composition` réifie la structure de l'objet de base (qui, rappelons-le, peut être composé de plusieurs objets internes) sous forme d'un graphe. Il permet de découvrir et de modifier la structure de l'objet de base ;
– le méta-espace `encapsulation` réifie la structure d'une interface. Il permet de découvrir les méthodes de cette interface, ainsi que son graphe d'héritage. Si l'implantation le permet, ce méta-espace peut également servir à modifier l'interface, par exemple en lui ajoutant des méthodes ;
– le méta-espace `environment` réifie « l'environnement d'exécution » d'une interface, qui décrit les différentes phases de traitement des messages arrivant sur cette interface : arrivée des messages, mise en file d'attente, distribution, ordonnancement... Ce méta-espace est à peu près équivalent à la réunion des sept méta-espaces de CodA.

Les auteurs proposent également d'utiliser le modèle ODP au niveau méta, c'est à dire que les méta-objets occupant les trois types de méta-espaces précédents sont en fait des objets ODP. On peut alors appliquer l'architecture récursivement, et définir ainsi une infinité de niveaux méta superposés. Pour garder un nombre de niveaux fini, chaque niveau méta est en fait instancié uniquement s'y on essaye d'y accéder.

Finalement, les auteurs proposent de compléter cette architecture par un ensemble de composants prédéfinis, mais extensible, permettant de « remplir » l'architecture abstraite précédente (aussi bien au niveau de base qu'au(x) niveau(s) méta). Cet ensemble contient des composants primitifs, qui offrent des protocoles, des micro-protocoles, des filtres, des tampons, des démultiplexeurs, des politiques d'ordonnancement..., ainsi que des composants composés, qui offrent des assemblages de composants prêts à l'emploi : objets de liaisons de types variés, méta-objets de type `environment` pré-configurés...

[10]ils ont par la suite proposés, dans [BCCD00], un quatrième méta-espace, nommé `resource`.

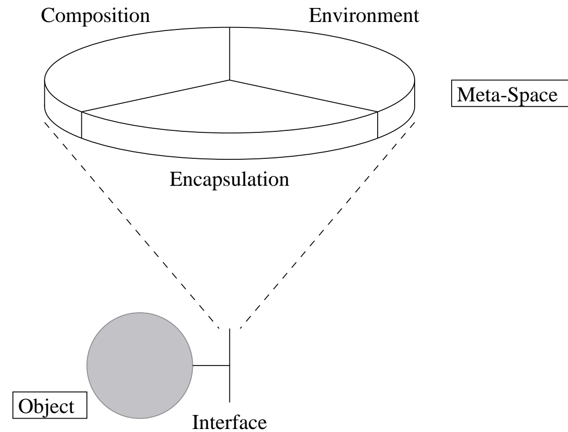


FIG. 2.9 – Architecture réflexive proposée par Blair et al. (tirée de [CBC98])

Applications

Les applications visées par Blair et al. concernent surtout l'auto-adaptabilité des applications, notamment multimédia, en fonction des variations du contexte d'exécution (puissance du processeur, débit, latence ou gigue du réseau...). Autrement dit, Blair et al. s'intéressent à une propriété non-fonctionnelle de « bas niveau », la performance. Celle-ci, contrairement aux propriétés de haut niveau, n'est programmée nulle part : elle « émerge » de la façon dont est réalisé tout le système, et en particulier la plate-forme middleware. La réflexivité n'est donc pas utilisée pour essayer de séparer le code de cette propriété non-fonctionnelle, mais pour permettre à l'application de modifier certains détails d'implantation à l'intérieur de la plate-forme middleware (choix des protocoles, de la politique de gestion des tampons, de l'ordonnanceur...), et qui ont une influence sur les performances de l'application.

[BCRP98] décrit une première expérimentation de l'architecture précédente : le but était de permettre à une application multimédia mobile de s'auto-adapter en modifiant la composition des objets de liaisons qu'elle utilise. Cette expérimentation était limitée au méta-espace `composition`.

Discussion

L'architecture proposée par Blair et al. semble très générique et très extensible, mais elle est difficile à évaluer pour le moment. Tout d'abord parce que leurs auteurs s'intéressent plus à l'auto-adaptabilité qu'à la séparation et à la composition de propriétés non-fonctionnelles de haut niveau. Ils n'ont donc pas expérimenté leur architecture par rapport à nos objectifs. On peut cependant penser, étant donné que le méta-espace `environment` est similaire à la réunion des sept méta-espaces de CodA[11], que cette

[11]et compte tenu du fait qu'il peut avoir un méta-méta-espace `composition` permettant de modifier sa composition

architecture permet effectivement de séparer au niveau méta le code des propriétés non-fonctionnelles de haut niveau.

L'architecture proposée utilise la réflexivité de façon plus poussée que dans les travaux précédents (à part Tj). On peut donc se demander si cela affecte ou non les performances. Malheureusement, le prototype actuel est réalisé en Python, et ne vise pas à atteindre de bonnes performances[12]. On ne peut donc pas trancher pour le moment.

2.2.6 Autres travaux

L'état de l'art précédent est loin d'être exhaustif. Ainsi, parmi les travaux non mentionnés, on peut citer [Led98, Led99], qui montre que l'on peut séparer le code non-fonctionnel du code fonctionnel en utilisant des *méta-classes* au lieu des méta-objets. On peut également citer [RGL98], qui propose un mécanisme pour l'adaptabilité dynamique : celui-ci est basé sur des *politiques d'adaptation* spécifiées par l'utilisateur, qui servent à reconfigurer automatiquement la plate-forme lors de l'occurence de certains événements. Tous les travaux précédents concernaient des plates-formes middleware complètes, mais il existe également des propositions basées sur des canevas logiciels (ou *frameworks*). Par exemple, [LK97] propose d'utiliser les concepts de la programmation orienté aspects pour programmer les applications distribuées. Il propose notamment de programmer les aspects de synchronisation et d'empaquetage/dépaquetage des données séparément, à l'aide de deux langages spécialisés. De même, [CM93] propose un compilateur « réflexif », appelé OpenC++, ainsi qu'un canevas logiciel construit avec ce compilateur et mettant en œuvre des algorithmes de communication de groupe au niveau méta.

2.3 Synthèse

Problèmes résolus

Par rapport à nos objectifs, les questions que l'on peut considérer comme résolues par les travaux présentés dans les sections 2.2.1 à 2.2.5 sont les suivantes :

- séparation du code fonctionnel et non-fonctionnel : il semble acquis qu'il faille utiliser la réflexivité pour tenter d'atteindre cet objectif. En effet, la méthode la plus fréquemment proposée pour essayer d'atteindre cet objectif consiste à utiliser une *architecture* réflexive (ce qui ne nécessite pas forcément d'utiliser un *langage* réflexif ; au contraire, cela peut même être nuisible en ce qui concerne les performances) ;
- séparation et composition du code de chaque propriété non-fonctionnelle : il semble acquis qu'il faille utiliser un mécanisme de composition au niveau méta. En effet, la plupart des auteurs proposent d'utiliser ce genre de mécanisme, généralement dans le but d'obtenir une architecture modulaire, mais qui semble également être un moyen pour essayer de séparer et de composer les propriétés

[12] un autre prototype, appelé OpenORB, est actuellement en cours de réalisation [BCCD00]. Il est basé sur COM et C++.

non fonctionnelles. Cependant, il n'y a pas de consensus sur le mécanisme de composition à utiliser : méta-espaces, listes de méta-objets...
– modularité et extensibilité de la plate-forme : cette propriété découle plus ou moins automatiquement des mécanismes précédents. Cependant, si on utilise la réflexivité de façon trop large on perd en performances (Tj), et si on l'utilise de façon trop limitée (EJB, Singhai et al.), on perd en modularité et en extensibilité. Il faut donc faire un compromis, comme souvent, entre les performances d'une part, et la modularité et l'extensibilité d'autre part.

Problèmes restant à résoudre

Les travaux précédents nous donnent des pistes pour essayer d'atteindre nos objectifs, mais aucun n'offre de solution complète pour les atteindre. De plus, ils laissent plusieurs autres problèmes en suspens :
– aucun ne propose de mécanisme pour protéger l'accès aux structures internes de la plate-forme rendues accessibles par l'introduction de la réflexivité ;
– le maintien de l'intégrité de la plate-forme et des applications, lors des opérations de reconfiguration dynamique, n'est pratiquement pas abordé (seul [PCB00] propose un mécanisme, les *component frameworks*) ;
– la plupart des plates-formes se limitent à un modèle client-serveur, et n'utilisent pas de notion de composant. Puisque nous souhaitons une plate-forme modulaire et extensible, il nous semble indispensable qu'une telle plate-forme ne soit pas limitée à un modèle de programmation particulier. Blair et al. ont cet objectif, mais il est malheureusement difficile d'évaluer leur proposition pour le moment, faute d'expérimentations ;
– de manière générale, les auteurs mettent l'accent sur l'architecture proposée, et délaissent les expérimentations « réalistes », avec des applications réelles ou plausibles. Notamment en ce qui concerne la séparation et la composition de propriétés non-fonctionnelles (ce qui est du, il est vrai, au fait que peu d'entre eux ont cet objectif).

Bilan

Le travail réalisé dans cette thèse s'appuie sur les solutions apportées par les travaux précédents (nous proposons en effet une architecture réflexive, ainsi qu'un nouveau mécanisme de composition pour le niveau méta), tout en essayant de remédier à certaines de leurs limitations. En effet, en plus des objectifs définis dans le chapitre précédent, nous avons également eu pour objectif de ne pas nous restreindre au modèle client-serveur, et de faire des expérimentations, basées sur des scénarios applicatifs plausibles, de l'architecture que nous proposons. Nous nous sommes également intéressé, dans une moindre mesure, au problème de la protection de la plate-forme (la solution que proposons pour ce problème, qui n'est malheureusement que partielle, est présentée dans l'annexe B page 125).

Chapitre 3

Proposition

Ce chapitre présente l'architecture de plate-forme middleware que nous proposons pour atteindre les objectifs présentés dans la section 1.4 page 11, ainsi que pour s'affranchir des limitations mises en évidence dans la section 2.3 page 32 concernant les travaux existants dans ce domaine. Cette architecture permet d'offrir au programmeur d'application un modèle de programmation très général, inspiré du modèle ODP, et présenté dans la première section. Les sections suivantes présentent l'architecture elle-même, qui est une sorte d'architecture EJB complétée par l'utilisation d'un mécanisme de composition original.

3.1 Modèle de programmation

Toute plate-forme middleware offre aux programmeurs un certain *modèle de programmation*, c'est à dire un ensemble de concepts qui permettent d'architecturer et de programmer des applications distribuées. Par exemple, les plates-formes CORBA offrent un concept d'objets distribués, interagissant selon un modèle de type client-serveur, et utilisant un mécanisme de passage de références d'objets bien défini. Les middlewares orientés message (*Message Oriented Middleware*, ou MOM) offrent quant à eux des concepts basés sur l'envoi de message asynchrone, des interactions de type publication-abonnement...

Nous avons choisi pour notre part un modèle très général, inspiré du modèle calculatoire d'ODP [ODP95b, ODP95a], qui englobe tous les autres. Ce choix découle du fait que la plupart des travaux existants concernant l'adaptation des propriétés non-fonctionnelles se sont placés dans le cadre simplificateur d'un modèle client-serveur (cf. section 2.3 page 32). Il était donc intéressant a priori d'étudier ce modèle plus général, non seulement parce que l'architecture résultante aurait un champ d'application plus vaste, mais aussi pour découvrir les problèmes nouveaux qui pouvaient se poser.

Après une présentation rapide des principaux concepts de notre modèle de programmation, cette section présente en détails le concept, nouveau à notre connaissance, de référence de connecteur : pourquoi nous l'avons défini, de quelle manière, et comment il s'utilise.

3.1.1 Composants, connecteurs et interfaces

Notre modèle de programmation utilise les concepts de *composants*, de *connecteurs* et d'*interfaces* pour structurer les applications. Ces concepts sont définis ci-dessous.

Définition 4 *Un* composant *encapsule du code et des données, comme un objet. Cependant, contrairement à un objet, un composant peut avoir plusieurs interfaces, qui peuvent être de types différents.*

Cette définition du terme « composant » est très minimaliste, et peut donc facilement être raffinée, si besoin est, pour correspondre à des définitions plus précises, comme celle de la norme CORBA 3.0 [CCM99]. Cette définition correspond en fait à la notion d'« objet » dans le modèle ODP. Mais nous n'utilisons pas le terme « objet » pour bien montrer qu'un composant n'est pas la même chose qu'un objet au sens des langages à objets, qui n'ont qu'une seule interface d'accès. En effet, un « objet » ODP peut être composé de plusieurs objets langage.

Définition 5 *Une* interface *est un point d'accès à un composant. Une interface a un* type*, défini par un ensemble de signatures de méthodes, ainsi qu'un sens : une interface* d'entrée *permet d'appeler des méthodes dans un composant, alors qu'une interface* de sortie *permet d'appeler des méthodes sur un connecteur.*

Il ne faut pas confondre cette notion avec la notion d'interface que l'on trouve dans certains langages, comme Java ou CORBA IDL. En effet, dans un langage, une interface désigne simplement un type, alors que, selon notre définition, qui correspond au concept de même nom dans ODP, ce terme doit être compris dans le sens « un point de communication qui se trouve *à l'interface* entre un composant et un connecteur ». Ainsi, un composant peut avoir plusieurs interfaces différentes mais de même type.

Définition 6 *Un* connecteur *est un composant qui relie deux interfaces ou plus, pas nécessairement du même type. Le rôle d'un connecteur est de permettre aux composants « normaux » de communiquer entre eux par l'intermédiaire de leurs interfaces. Un connecteur peut représenter n'importe quel type d'interaction entre composants : client-serveur, publication-abonnement...*

Ce concept de connecteur correspond à la notion d'objet de liaison dans ODP. Du fait que l'on a choisi d'utiliser le terme « composant » au lieu de « objet », nous avons également choisi d'utiliser le terme « connecteur » au lieu d'« objet de liaison », pour rester cohérent (les termes composants et connecteurs sont généralement employés de concert). La figure 3.1 page ci-contre illustre les définitions précédentes.

Remarques

- comme dans le modèle ODP, les interfaces d'un composant peuvent évoluer dynamiquement : certaines peuvent être créées, d'autres détruites. Cela n'empêche pas les composants d'avoir, si besoin est, un certain nombre d'interfaces permanentes, définies statiquement ;

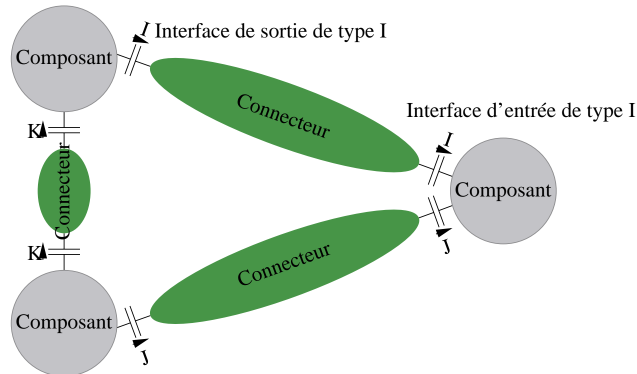


FIG. 3.1 – Les concepts du modèle de programmation

– le modèle ODP permet de considérer un graphe de composants et de connecteurs comme un composant ou un connecteur unique. Ce qui permet de définir des composants et connecteurs complexes par composition de composants et connecteurs plus simples. Rien n'empêche notre modèle d'offrir cette possibilité. Toutefois, associer des propriétés non-fonctionnelles à des composants composés, qui peuvent être distribués sur plusieurs sites, est un problème a priori bien plus complexe que d'associer ces propriétés à des composants simples (i.e. non composés) et non distribués. C'est pourquoi nous nous sommes limité dans ce travail à un modèle « plat », sans composants composés.

Exemples

Les exemples ci-dessous, également représentés sur la figure 3.2 page 39, montrent que le concept de connecteur peut représenter n'importe quel type d'interaction entre composants :

– une liaison client-serveur entre un client C et un serveur S peut être représentée par un connecteur reliant une interface de sortie de C à une interface d'entrée de même type de S. L'interface de sortie de C est alors utilisée pour appeler des méthodes sur le connecteur, et indirectement sur S. Symétriquement, l'interface d'entrée de S permet de répondre aux requêtes du client en appelant les méthodes du composant S ;
– si S offre une méthode `getBalance` alors que C attend une méthode `solde`, on peut les connecter malgré tout à l'aide d'un connecteur plus évolué, prenant en charge la traduction des noms de méthode. Un tel connecteur relie alors des interfaces de types différents ;
– une liaison de type publication-abonnement entre un certain nombre d'éditeurs et d'abonnés[1] peut être représentée par un connecteur reliant des interfaces de sortie des éditeurs à des interfaces d'entrée des abonnés. Les éditeurs (i.e. les

[1] qu'ils soient abonnés à des éditeurs désignés explicitement, ou bien implicitement par l'intermédiaire de canaux de diffusion - ou *topics*

serveurs d'information) utilisent leur interface de sortie pour publier des messages, et les abonnés (i.e. les clients) sont prévenus de l'arrivée de ces messages par leur interface d'entrée. La situation est donc symétrique du cas client-serveur, puisqu'ici les serveurs utilisent une interface de sortie au lieu d'une interface d'entrée, et vice et versa pour les clients ;
– un flux vidéo ou audio entre un serveur et un ou plusieurs clients peut être représenté par un connecteur similaire à celui du cas publication-abonnement : l'éditeur est remplacé par le serveur de flux vidéo, les abonnés par les clients, et les messages asynchrones par des paquets de données. Un tel connecteur peut prendre en charge le découpage du flux en paquets, ainsi que son réassemblage chez les clients, de façon à fournir à ces derniers une interface semblable à celle de la classe `java.io.InputStream` ;
– un connecteur peut également représenter des interactions basées sur des objets partagés. Les composants reliés à un tel connecteur communiquent en modifiant l'état d'un objet partagé, et en étant notifié automatiquement de toute modification effectuée par les autres composants. Un exemple d'application typique d'un tel connecteur est l'édition coopérative de documents.

3.1.2 Références d'interface

Afin de permettre au programmeur de désigner les entités de son application, des *références* sont indispensables. Le modèle CORBA définit par exemple des références d'objet. Mais ce concept est limité au cas des interactions client-serveur. C'est pourquoi le modèle ODP utilise un concept plus général : les *références d'interface*. Nous présentons ici ce concept, ainsi que ses limitations quand on souhaite adapter les propriétés non-fonctionnelles des applications. Pour résoudre ces problèmes, nous introduisons dans la section suivante la notion de *référence de connecteur*.

Comme son nom l'indique, une référence d'interface désigne une interface d'un composant. On peut voir ce concept comme une généralisation des références d'objet, puisque l'on peut considérer qu'une référence d'objet désigne l'unique interface de cet objet. Étant donné que les connecteurs peuvent être de type très différents, les références d'interfaces ne suffisent pas à elles seules pour construire de nouveaux connecteurs (alors qu'une référence d'objet CORBA suffit pour connecter un nouveau client à un serveur, car il n'y a qu'un seul type de « connecteur »). C'est la raison pour laquelle le modèle ODP définit également le concept *d'usine à liaisons* [DHTS98].

Une usine à liaisons permet de construire des objets de liaison (que nous appelons connecteurs), à partir d'un certain nombre de références d'interfaces. Selon ce modèle, il faut une usine à liaisons par type de connecteur possible. C'est faisable si le nombre de connecteurs possibles est raisonnable, mais cela devient difficile si ce nombre est très grand. Or c'est justement le cas quand on souhaite adapter les propriétés non fonctionnelles des applications. En effet, de nombreuses propriétés non-fonctionnelles requièrent des modifications de protocoles au niveau des connecteurs. Par exemple, les transactions requièrent la propagation d'un contexte transactionnel dans les messages d'invocation à distance, la protection requiert l'ajout d'informations sur l'identité de l'appelant dans

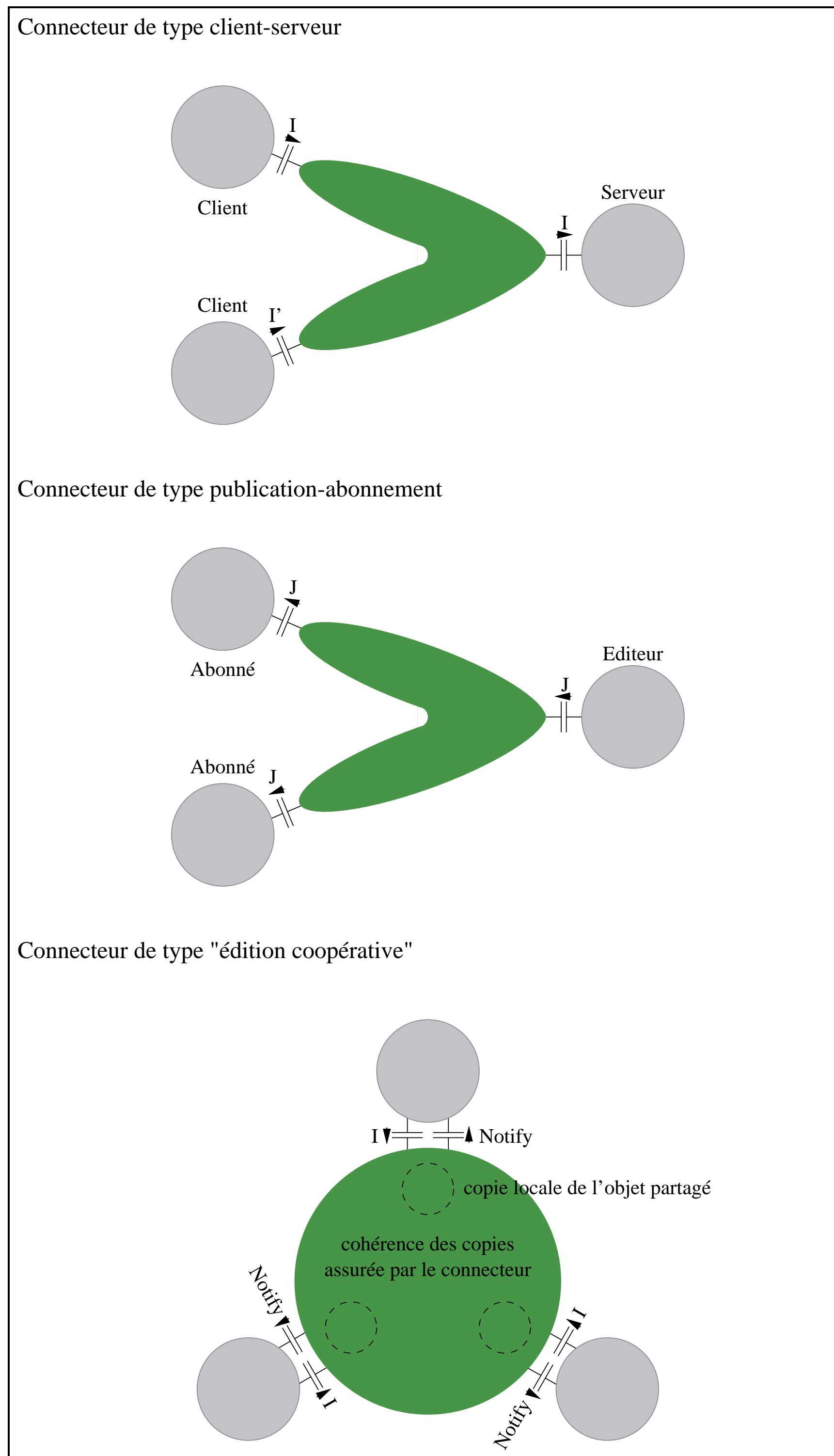


FIG. 3.2 – Exemples de connecteurs

ces mêmes messages... Comme en plus le nombre d'adaptations possible croit de façon exponentielle avec le nombre de propriétés non-fonctionnelles, on obtient un très grand nombre possible de connecteurs, qui nécessiterait donc un nombre équivalent d'usines à liaisons.

3.1.3 Références de connecteur

Pour résoudre ce problème, l'idée est d'utiliser en quelque sorte une usine à liaison unique mais générique, et de reporter les informations sur la façon de construire les connecteurs des usines à liaisons dans les références elles-mêmes. Cela nous conduit à la notion de *référence de connecteur*.

Comme son nom l'indique, une référence de connecteur désigne un connecteur. On peut voir ce concept comme une autre généralisation possible des références d'objet CORBA, puisque l'on peut considérer qu'une référence d'objet désigne le connecteur client-serveur permettant d'accéder à cet objet. Plus précisément, une référence de connecteur a deux rôles :

- elle désigne tout d'abord un connecteur, en désignant (implicitement ou explicitement) toutes les interfaces de ce connecteur. *Par exemple*[2], une référence de connecteur client-serveur désigne l'interface (d'entrée) de l'objet serveur et, implicitement, les interfaces clientes. Une référence de connecteur publication-abonnement désigne les interfaces (de sortie) des éditeurs et, implicitement, les interfaces des abonnés. Une référence de connecteur de type « édition coopérative » désigne une interface contenant la copie maître, qui elle même contient la liste des interfaces contenant les autres copies. Et ainsi de suite ;
- elle décrit ensuite comment créer de nouvelles interfaces pour ce connecteur, en fonction de leur type et de leur sens (entrée ou sortie). Par exemple, une référence de connecteur client-serveur décrit comment construire une interface de sortie pour un nouveau client. Une référence de connecteur publication-abonnement décrit comment construire une interface d'entrée pour un nouvel abonné. Une référence de connecteur de type « édition coopérative » décrit comment construire une interface pour un nouveau participant. Et ainsi de suite.

De plus, cette description de la façon de construire de nouvelles interfaces est effectuée dans un format générique. En effet, comme nous le verrons plus précisément dans la section 4.2.6 page 69, une référence contient, en gros, une description de la pile de protocoles à instancier pour créer une nouvelle interface. Cela permet d'utiliser une seule usine à liaisons, générique, pour construire n'importe quel type de connecteur.

Cependant le mécanisme précédent ne permet que d'agrandir des connecteurs existants, pas d'en créer de nouveaux. Pour résoudre ce problème sans réintroduire d'usines à liaisons spécifiques, nous décrivons la façon de créer les interfaces initiales des connecteurs dans un fichier de configuration associé à chaque composant, là aussi dans un format générique. Le fonctionnement exact de ce mécanisme, qui nécessite de connaître

[2] ces exemples ne sont pas du tout exhaustifs : les références de connecteur client-serveur, publication-abonnement... peuvent être implantés de multiples façons, et on peut également implanter des références pour d'autres types de connecteur.

précisément l'architecture que nous proposons, est décrit dans la section 4.3.3 page 75.

Remarques

- selon la définition ci-dessus, les interfaces créées à partir d'une référence de connecteur appartiennent au connecteur désigné par cette interface. Autrement dit, les connecteurs s'agrandissent dynamiquement au fur et à mesure que de nouveaux composants y sont attachés. Cela nous amène à préciser notre définition des connecteurs : un connecteur relie toutes les interfaces qui communiquent *entre elles* en utilisant les *mêmes protocoles*. Par exemple, dans le cas client-serveur, nous sommes amenés à considérer que toutes les interfaces clientes, ainsi que l'interface serveur, font partie d'un seul connecteur (alors que, classiquement, on considère qu'il y a un connecteur par client, et que ces connecteurs partagent la même interface serveur) ;
- les références de connecteur permettent de se passer des usines à liaisons, ce qui était le but recherché (plus précisément, il n'y a plus qu'une seule usine à liaisons, générique). En contrepartie, les références de connecteurs sont beaucoup plus volumineuses que de simples références d'interface, puisqu'une référence de connecteur contient une ou plusieurs références d'interfaces, ainsi qu'une ou plusieurs descriptions de « pile de protocoles » ;
- il est probablement possible d'utiliser à la fois des références d'interface, avec leurs usines à liaisons, et des références de connecteurs, afin de choisir les références les plus adaptées selon la situation. Mais nous n'avons pas exploré cette piste : nous nous sommes limités à des références de connecteurs uniquement.

3.1.4 Passage de références

Le programmeur ne manipule jamais les références de connecteur directement : il n'utilise en effet que des références d'objet langage (s'il utilise un langage orienté objet). Les références de connecteur sont donc manipulées indirectement, via des objets qui *représentent* les interfaces de ce composant, comme le montre la figure 3.3.

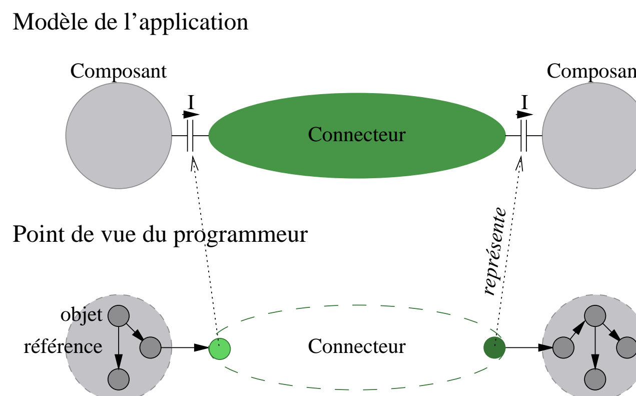


FIG. 3.3 – Représentation des interfaces par des objets langages

Par exemple, lorsqu'un serveur retourne un objet par référence[3] à un client, il transmet en fait une référence de connecteur au client, bien que cela soit complètement invisible pour le programmeur de l'application. Plus précisément, le serveur *exporte* l'objet, puis le client *importe* la référence de connecteur résultant de cette exportation. La figure 3.4 permet de définir le fonctionnement de ces opérations :

– supposons que le serveur retourne l'objet `s` en résultat d'un appel du client. Dans ce cas l'exportation consiste à créer le représentant `s1` s'il n'existe pas déjà, puis à retourner la référence `ref1` du connecteur auquel il appartient. L'importation de `ref1` consiste quant à elle à créer le représentant `t1` s'il n'existe pas déjà, et à retourner cet objet ;
– supposons maintenant que le serveur retourne le représentant `t2` en résultat d'un appel du client. Dans ce cas l'exportation consiste à retourner la référence `ref2` du connecteur auquel appartient ce représentant, alors que l'importation de `ref2` consiste à retourner l'objet `c`.

Exporter *un objet* consiste donc à trouver ou à construire une référence de connecteur « correspondant » à cet objet, alors qu'importer *une référence de connecteur* consiste à trouver ou à construire un objet « correspondant » à cette référence.

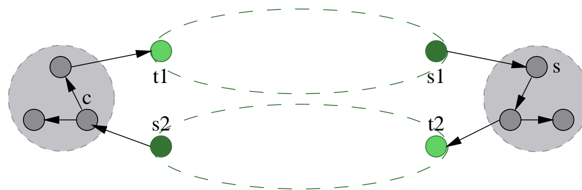


FIG. 3.4 – Importation et exportation des références de connecteurs

3.2 Architecture de haut niveau

Le modèle de programmation que nous souhaitons offrir aux applications étant défini, nous pouvons maintenant présenter l'architecture de plate-forme middleware que nous proposons pour exécuter les applications respectant ce modèle, tout en permettant d'adapter leurs propriétés non-fonctionnelles. Nous commençons pour cela, dans cette section, par présenter une vue de haut niveau de cette architecture. La section suivante présente le modèle de composition qui permet de la raffiner afin de pouvoir atteindre nos objectifs.

L'architecture de haut niveau que nous proposons est inspirée du modèle EJB : elle est en effet basée sur des serveurs, des conteneurs, ainsi que des talons et des squelettes. Ce choix est motivé par le fait que le modèle EJB permet de séparer le code non-fonctionnel du code fonctionnel (grâce à des objets d'interposition - cf. section 2.2.3 page 24). Les sections suivantes définissent plus précisément les éléments de notre architecture.

[3]les objets qui implantent l'interface `JavaPodInterface` sont passés par référence (cf. section 4.2.2 page 66). Les autres sont passés par copie.

3.2.1 Conteneurs

Comme dans le modèle EJB, un conteneur encapsule un composant et gère ses propriétés non-fonctionnelles, notamment grâce à des objets d'interposition, qui ne sont autres que des squelettes étendus (cf. section 2.2.3 page 24), et que nous appelons donc tout simplement des squelettes.

La différence principale entre notre concept de conteneur et celui des EJB est que, pour nous, un conteneur ne contient qu'un et un seul composant et que, par conséquent, il ne contient pas d'usine à composants (*EJB Home*). Nous avons choisi cette définition car elle est plus générale que celle des EJB. En effet, notre modèle de composant et de conteneur peut simuler celui des EJB (en plaçant plusieurs *beans* à l'intérieur d'un composant - cf. figure 3.5), mais l'inverse n'est pas vrai (un *bean* ne peut pas simuler un composant ayant plusieurs interfaces d'accès).

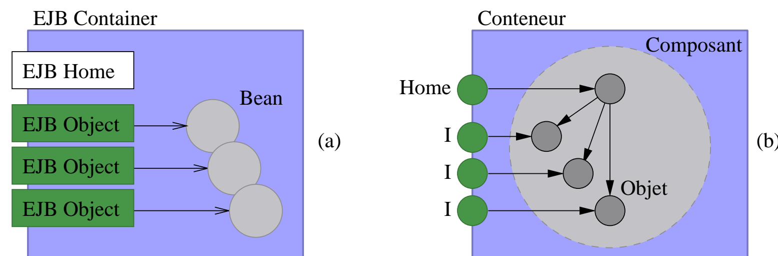


FIG. 3.5 – Simulation des conteneurs EJB (a) avec nos propres conteneurs (b)

Simuler le modèle EJB avec nos composants et conteneurs ne revient toutefois pas au même en ce qui concerne les propriétés non-fonctionnelles. En effet, dans le modèle EJB, les propriétés non-fonctionnelles sont associées à chaque *bean*, alors que dans notre modèle elles sont associées au composant entier. Par exemple, pour la persistance, cela signifie que chaque *bean* peut être enregistré dans le support persistant indépendamment des autres alors que, si on plaçait ces *beans* dans un de nos composants (cf. figure 3.5), ils seraient tous enregistrés en même temps.

3.2.2 Talons, squelettes et portes

Un talon représente une interface de sortie d'un composant, alors qu'un squelette représente une interface d'entrée. Un talon ou un squelette est également appelé une *porte*.

Du fait de leur nature, qui est d'être à l'interface, ou à l'intersection, entre un composant et un connecteur, nous considérons que les talons et les squelettes appartiennent à la fois à un conteneur et à un connecteur (comme le suggère la figure 3.6 page suivante). Vus comme éléments d'un conteneur, ils apparaissent comme des passages obligés qui permettent la communication entre le composant encapsulé et son environnement. Vus comme éléments d'un connecteur, ils font apparaître un connecteur comme un objet fragmenté, constitué de talons et squelettes situés sur des serveurs différents.

Comme nous l'avons déjà indiqué pour les squelettes, talons et squelettes ont un rôle plus étendu que dans CORBA car, comme les objets d'interposition dans le modèle EJB, ils permettent d'associer des propriétés non-fonctionnelles aux composants. De plus, ils ne sont pas limités à des connecteurs client-serveurs : même s'ils appartiennent à des connecteurs de type « flot multimédia » ou « objet partagé », nous continuons à appeler ces objets talons et squelettes, bien qu'ils n'aient même plus dans ce cas la fonction d'empaquetage et de dépaquetage des requêtes.

3.2.3 Serveurs

Un serveur, dans notre architecture, est similaire à un serveur EJB : il offre un environnement d'exécution pour les conteneurs, des services systèmes... Il peut également offrir des fonctions d'administration : déploiement de composants, surveillance de l'activité du serveur... La figure 3.6 résume l'architecture de haut niveau que nous proposons.

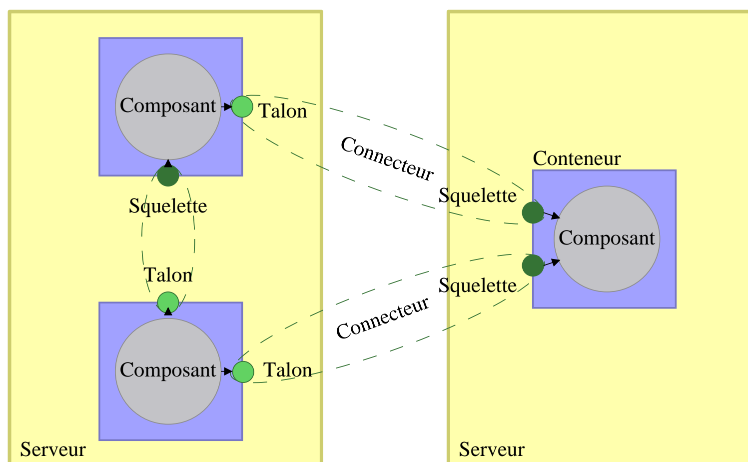


FIG. 3.6 – Résumé de l'architecture de haut niveau

3.3 Modèle de composition

L'architecture précédente permet a priori, comme avec le modèle EJB, de séparer le code non-fonctionnel des applications. Cependant, si on se contentait, comme les plates-formes EJB actuelles, de générer des talons, des squelettes et des conteneurs monolithiques, à l'aide d'un générateur de code lui aussi monolithique, on ne pourrait pas atteindre nos deux autres objectifs (modularité et extensibilité, et séparation et composition du code de chaque propriété non-fonctionnelle). C'est pourquoi nous introduisons dans cette section un modèle de composition, que l'on utilise ensuite pour raffiner l'architecture présentée ci-dessus.

La première section montre à l'aide d'un exemple les besoins qu'il faut être capable de satisfaire pour pouvoir séparer et composer « assez facilement » des propriétés non-fonctionnelles. La section suivante montre que les mécanismes existants ne sont pas entièrement satisfaisants pour résoudre le problème, ce qui nous conduit dans la troisième section à la définition d'un nouveau mécanisme de composition.

3.3.1 Les besoins

Selon nous, pour pouvoir séparer et composer « facilement » des « briques » fournissant des propriétés non-fonctionnelles, il faut un mécanisme de composition capable de *simuler* les mécanismes de l'héritage de classes (surcharge de méthodes, ajout de nouvelles méthodes...) *sans* leurs inconvénients. Cette section explique pourquoi.

Pour cela, considérons un talon (imaginaire) d'un connecteur client-serveur minimal, et supposons que ce talon ait les méthodes suivantes :

- la méthode `invoke` implante un mécanisme d'appel de méthode à distance. Elle prend en argument un nom de méthode, des arguments, et retourne le résultat de l'exécution à distance de cette méthode ;
- les méthodes `marshal` et `unmarshal` permettent d'empaqueter et de dépaqueter les arguments d'un appel de méthode à distance. Elles sont utilisées par la méthode `invoke`, qui leur délègue une partie de ses fonctions ;
- les méthodes de l'interface applicative (comme par exemple `getBalance` pour un compte bancaire) sont ajoutées automatiquement par un générateur de code. Elles se contentent d'appeler la méthode `invoke` avec leur nom et leurs arguments en paramètres.

Il est maintenant facile de voir pourquoi les mécanismes de l'héritage de classes sont naturels pour adapter ce talon, afin de prendre en compte diverses propriétés non-fonctionnelles :

- si on veut enregistrer une trace de l'activité du client, un moyen simple et naturel est de **surcharger** la méthode `invoke`, de façon à ce qu'elle imprime un message avant et/ou après chaque appel de méthode à distance. De même, si on veut garder les résultats des appels déjà effectués dans un cache, pour optimiser les performances, un moyen simple consiste là encore à surcharger la méthode `invoke`. Si on veut protéger le serveur, alors il faut envoyer au serveur l'identité du client dans chaque message. Pour cela, un moyen simple est de surcharger la méthode `marshal`. De même si on veut spécialiser l'algorithme d'empaquetage, ou si on veut ajouter dans chaque message des informations pour le temps-réel ;
- les méthodes `marshal` et `unmarshal` sont spécifiques aux connecteurs de type client-serveur. Par exemple, elles sont inutiles pour un connecteur de type flot de données. En effet, un talon de type « flot de données » fonctionne généralement en recevant des paquets envoyés automatiquement d'un serveur, en les réassemblant dans un tampon au fur et à mesure, et en utilisant ce tampon pour fournir les données au client. Par conséquent, il n'y a pas besoin d'appels de méthode à distance, ni donc de fonction d'empaquetage et de dépaquetage. Il serait donc naturel d'**ajouter** les méthodes `marshal` et `unmarshal` à un talon *générique*,

uniquement quand elles sont nécessaires.

Mais ces exemples montrent aussi les limites de l'héritage de classes. En effet, avec ce mécanisme, il faut une classe par adaptation possible. Rien qu'avec les 5 adaptations citées ci-dessus pour le talon client-serveur, il faudrait donc $2^5 = 32$ classes différentes. De plus, la classe d'un objet ne pouvant pas être modifiée dynamiquement, il serait impossible d'activer et de désactiver dynamiquement certaines propriétés non-fonctionnelles, comme la surveillance.

En résumé, pour pouvoir séparer et composer « assez facilement » des propriétés non-fonctionnelles, il faut, selon nous, un mécanisme de composition *dynamique* permettant de *simuler* la surcharge et l'ajout de méthodes, et ne provoquant pas d'explosion combinatoire, contrairement à l'héritage de classes.

3.3.2 Limites des mécanismes existants

De nombreux mécanismes existants permettent a priori de résoudre le problème précédent : réflexivité, délégation... Cette section montre, pour chacun de ces mécanismes, comment le diagramme de classes de la figure 3.7 pourrait être simulé, et explique pourquoi aucune ces solutions ne nous satisfait vraiment.

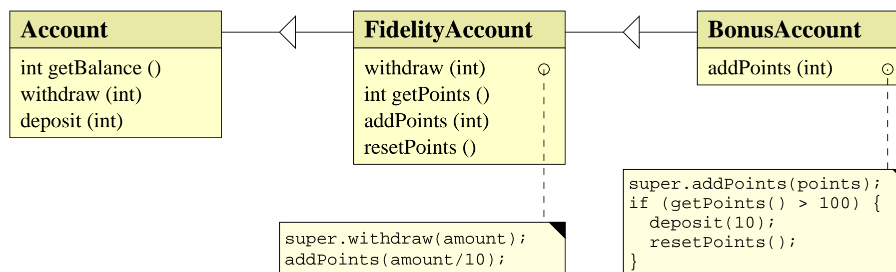


FIG. 3.7 – Une hiérarchie de classes très simple. La classe `Account` représente un compte bancaire extrêmement simplifié. La sous-classe `FidelityAccount` gère des points de fidélité, qui sont incrémentés à chaque retrait. Lorsque 100 points ont été accumulés, la sous-classe `BonusAccount` crédite le compte de 10 unités

Réflexivité

Les plates-formes de recherche présentées dans le chapitre 2 offrent des mécanismes de composition qui permettent de simuler certains mécanismes de l'héritage de classes, et notamment la surcharge de méthodes. De plus, ces mécanismes semblent basés sur la réflexivité. Il est donc naturel de se demander si on peut réellement satisfaire les besoins mis en évidence dans la section précédente à l'aide de la réflexivité. Cette section montre que ce n'est pas le cas.

Il convient tout d'abord de faire la distinction entre les mécanismes réellement basés sur la réflexivité et ceux qui ne le sont pas. On s'aperçoit alors que tous les mécanismes de composition de « méta-objets », que ce soit avec des méta-espaces, des listes ou

des graphes arbitraires, ne sont en fait que des techniques de composition d'*objets*, et plus précisément des techniques de composition par délégation. Le seul mécanisme réellement réflexif est celui qui consiste à réifier, modifier puis réfléchir une opération du niveau de base.

La question posée au début de cette section se résume donc à savoir si les mécanismes de l'héritage peuvent être simulés en réifiant des opérations de base. En théorie oui. Par exemple, la hiérarchie de la figure 3.7 page ci-contre peut être simulée avec un objet de base et deux méta-objets, comme le montre la figure 3.8 : le méta-objet « surcharge » la méthode `withdraw` de l'objet de base, grâce à ses capacités d'interception, et introduit de nouvelles méthodes, qui peuvent à leur tour être « surchargées » par un méta-méta-objet. De plus, ce mécanisme de composition peut être utilisé dynamiquement, et permet d'éviter les problèmes d'explosion combinatoire.

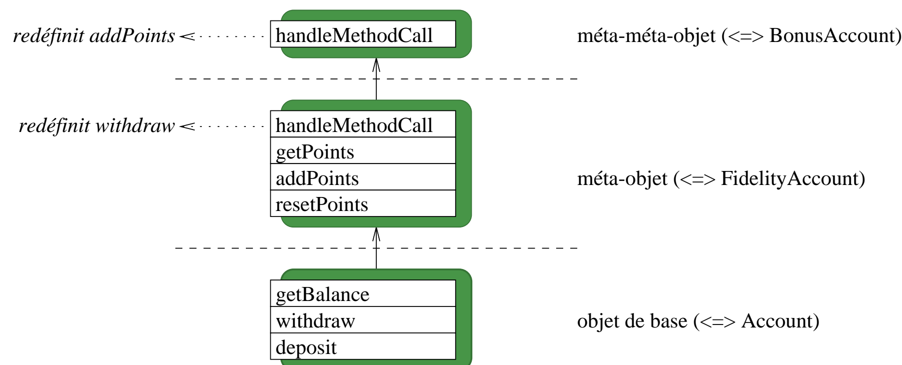


FIG. 3.8 – Simulation de l'héritage en utilisant une tour de méta-objets

Malheureusement ce mécanisme est difficilement utilisable en pratique, à cause des inconvénients suivants :

– les méta-objets ne sont pas fait pour surcharger des méthodes spécifiques d'un objet de base, mais au contraire pour pouvoir s'appliquer à n'importe quel type d'objet de base. De plus, pour le programmeur, une telle utilisation détournée peut être la source de nombreuses erreurs de typage non détectables à la compilation. En effet, il ne faut pas se tromper, dans les méthodes `handleMethodCall`, sur le type des arguments des méthodes de base que l'on veut surcharger ;
– surcharger plusieurs fois une même méthode, par exemple `withdraw`, à l'aide d'une tour de méta-objets ne serait vraiment pas pratique : le méta-méta-objet devrait surcharger la méthode `handleMethodCall` du méta-objet afin de pouvoir surcharger, indirectement, la méthode `withdraw` de l'objet de base ! Il faudrait donc compléter ce modèle avec un mécanisme de composition de méta-objets intra-niveau, le plus simple étant une composition en liste, comme dans Dalang [WS98]. Mais on passerait alors d'une structure linéaire à une structure en arbre, plus complexe à manipuler.
– les méthodes « ajoutées » à l'objet de base, comme la méthode `getPoints`, ne sont pas facilement accessibles par les clients de cet objet, qui doivent chercher

dans quel méta-objet ces méthodes sont implantées, puis appeler ces méthodes directement sur ces méta-objets (le niveau méta n'est donc plus transparent pour le niveau de base).
– un tel mécanisme serait peu efficace, car les opérations de réification sont en général assez coûteuses en temps.

Délégation

La délégation, dans les langages orientés-objets classiques, consiste à confier tout ou partie du comportement d'un objet à un ou plusieurs objets délégués. C'est une technique de composition très largement utilisée, y compris au niveau méta : en effet, comme nous l'avons déjà dit, toutes les techniques de composition de méta-objets sont en fait des variantes de composition d'objets par délégation. Il est donc naturel de se demander si on peut simuler les mécanismes de l'héritage avec cette technique. Là encore, la réponse est oui en théorie, mais non en pratique.

Comme le montre la figure 3.9, on peut simuler la hiérarchie de classes de la figure 3.7 page 46 en utilisant la délégation. L'idée consiste à séparer l'implantion de l'objet `Account` dans un objet délégué `AccountImpl`. On peut alors intercaler entre les deux un ou plusieurs objets (comme `FidelityAccount`) pour intercepter et surcharger certaines méthodes. De plus, ce principe peut être appliqué récursivement. Par exemple, l'implantation de l'objet `FidelityAccount` est séparée dans un objet délégué `FidelityAccountImpl`, ce qui permet d'intercaler entre les deux un objet `BonusAccount` pour surcharger la méthode `addPoints`. De plus, ce mécanisme peut être utilisé dynamiquement, et permet d'éviter les problèmes d'explosion combinatoire.

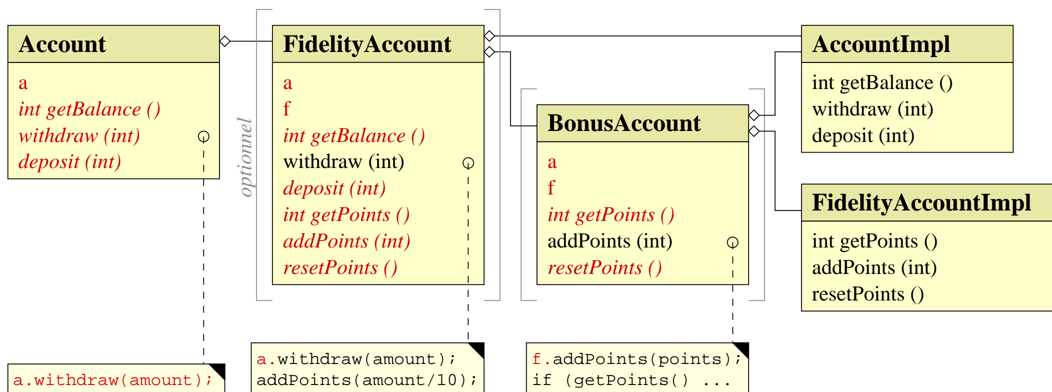


FIG. 3.9 – Simulation de l'héritage en utilisant la délégation

Malheureusement ce mécanisme a également des inconvénients, qui le rendent difficilement utilisable en pratique :
– pour surcharger *une* méthode un objet intercalé doit implanter *toutes* les méthodes fournies par le délégué par défaut correspondant. Par exemple, bien que l'objet `FidelityAccount` ne surcharge que la méthode `withdraw`, il doit quand même

implanter les méthodes `getBalance` et `deposit` (en déléguant le traitement à `AccountImpl`). De même pour l'objet `BonusAccount` et la méthode `addPoints`. D'où des indirections superflues qui dégradent les performances ;
– comme pour la solution basée sur la réflexivité, les méthodes « ajoutées » à l'objet de base ne sont pas facilement accessibles par les clients de cet objet. Par exemple, pour appeler la méthode `getPoints` sur un objet de type `Account`, il faut tout d'abord déterminer dans quel délégué cette méthode est implantée, puis appeler la méthode directement sur ce délégué.
– comme pour la solution basée sur la réflexivité, on obtient une structure en arbre, voire en graphe, assez complexe à manipuler.

Délégation « étendue »

Certains langages, comme Self [US87], utilisent la délégation entre objets à un niveau plus fondamental que dans les langages classiques. En effet, dans ces langages, lorsqu'un objet reçoit un appel de méthode alors qu'il n'implante pas lui-même cette méthode (c'est possible car ces langages ne sont pas typés statiquement), il peut *déléguer* le traitement de cette méthode à un autre objet. Ce mécanisme de délégation « étendu » permet de simuler facilement les mécanismes de l'héritage.

Par exemple, la figure 3.10 montre comment la hiérarchie de classes de la figure 3.7 page 46 peut être simulée en Self[4]. Lorsqu'un objet client appelle une méthode sur un objet, et si cette méthode n'est pas implantée par l'objet lui-même, l'appel est implicitement délégué à l'objet père, désigné par le *slot* `parent*`. Grâce à ce mécanisme, un objet peut surcharger des méthodes fournies par son objet père. Il peut également lui ajouter des méthodes. De plus, ce mécanisme est complètement dynamique, et évite les problèmes d'explosion combinatoire de l'héritage de classes.

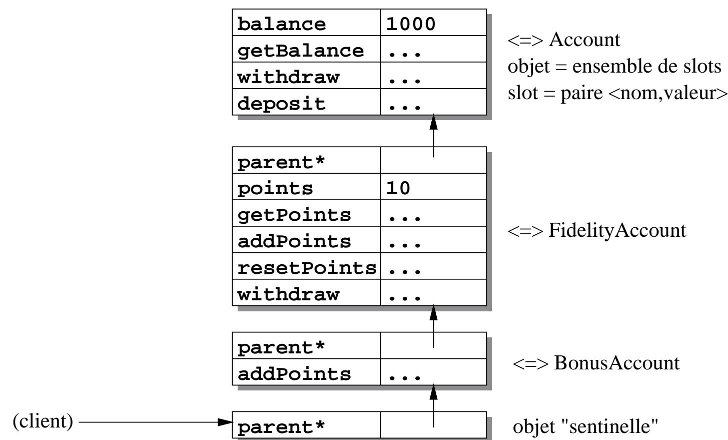


FIG. 3.10 – Simulation de l'héritage en utilisant Self

[4]en pratique, les slots contenant les méthodes seraient reportés dans des objets séparés appelés *traits*, afin de pouvoir être partagés. Cette séparation n'est pas représentée ici, pour simplifier la figure.

Mais ce mécanisme requiert un langage non typé statiquement. Selon notre point de vue, c'est un inconvénient : en effet, ces langages ne permettent pas de détecter autant d'erreurs à la compilation que les langages statiquement typés. Un autre inconvénient est que si on veut pouvoir modifier dynamiquement la composition d'un objet composé, il faut utiliser un objet « sentinelle » pour référencer cet objet (par exemple, si le compte bancaire de la figure 3.10 page précédente était référencé par l'intermédiaire de l'objet `b`, ces références deviendraient invalides si `b` était supprimé dynamiquement). Ces inconvénients sont toutefois minimes comparés à ceux des solutions précédentes, et c'est pourquoi le mécanisme de composition que nous proposons dans la section 3.3.3 est très similaire à celui-ci.

3.3.3 Un nouveau mécanisme

Les mécanismes de composition existants n'étant pas complètement satisfaisants pour simuler l'héritage de classes, comme le montre la section précédente, nous proposons dans cette section un nouveau mécanisme, conçu directement dans ce but.

Ce nouveau mécanisme permet de définir des *objets composés*. Comme son nom l'indique, un objet composé est un objet : son état interne est, par définition, la réunion des états internes de ses membres. De même, ses méthodes sont, par définition, la réunion des méthodes de ses membres. Pour simuler la sémantique de l'héritage de classe, la sémantique de ces méthodes est définie de la façon suivante :

- tout d'abord, les membres d'un objet composé sont totalement ordonnés, c'est à dire placés dans une liste. Le membre en tête de liste est appelé un *objet extensible*, alors que les autres sont appelés des *extensions*[5]. L'objet extensible ne peut être remplacé ni supprimé dynamiquement, alors que les extensions peuvent être ajoutées, remplacées, ou supprimées dynamiquement. L'objet extensible joue le rôle d'une classe de base, et les extensions ceux des sous-classes de la classe de base ;
- un appel à une méthode `m` sur un objet composé est exécuté par le *dernier* membre de la liste dans lequel cette méthode est définie. Si `m` n'est définie dans aucun membre, l'appel échoue ;
- dans les extensions, une forme spéciale d'appel de méthode, notée `dsuper.`*m* par analogie avec la notation classique `super.`*m*, permet d'appeler la version originale d'une méthode surchargée dynamiquement par l'extension courante. Ainsi, un appel de ce type à une méthode `m` depuis une extension `e` est exécuté par le *dernier* membre de l'objet composé dans lequel cette méthode est définie, et qui se situe *avant* l'extension `e`. Si un tel membre n'existe pas, l'appel échoue.

La figure 3.11 page ci-contre montre comment simuler la hiérarchie de classes de la figure 3.7 page 46 à l'aide d'un objet composé : l'objet de base correspond à la classe `Account`, la première extension à la classe `FidelityAccount`, et la seconde à la classe `BonusAccount`. Grâce à la sémantique des objets composés, la méthode `withdraw` de la première extension surcharge celle de l'objet de base. De même, la méthode `addPoints` de la seconde extension surcharge celle de la première extension.

[5]cette distinction n'est pas vraiment fondamentale, et pourrait être supprimée.

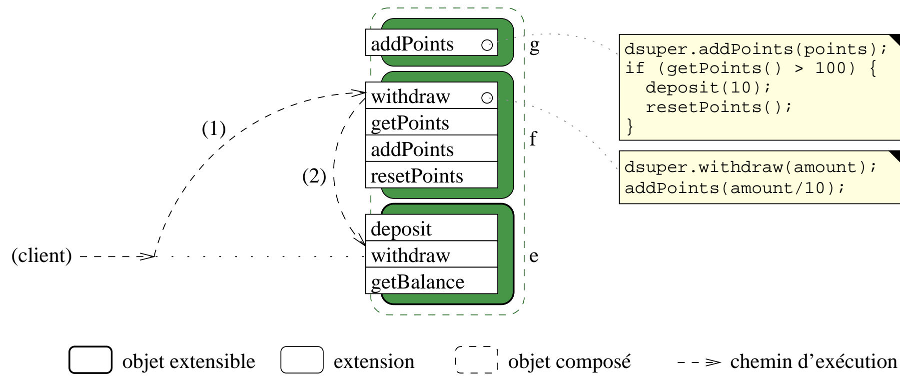


FIG. 3.11 – Un exemple d'objet composé

Comme nous le souhaitions, ce mécanisme permet de simuler la surcharge et l'ajout de méthodes, il peut être utilisé dynamiquement, et il évite les problèmes d'explosion combinatoire. De plus, il utilise une structure en liste au lieu d'une structure en arbre ou en graphe et, comme nous le verrons dans la section 4.1 page 55, il peut être mis en œuvre au sein d'un langage statiquement typé, sans utiliser d'opérations de réification, sans introduire d'indirections superflues, et sans avoir besoin d'introduire d'objets « sentinelles ». Finalement, on peut lui associer un mécanisme de contrôle d'accès très simple (cf. annexe B page 125), alors que ce serait probablement bien plus difficile avec les mécanismes précédents.

Méthodes inversées

Ce mécanisme à cependant un inconvénient, qui est une contrepartie de sa simplicité : comme l'explique cette section, il faut parfois décomposer une extension en fragments pour insérer chaque fragment à un endroit différent dans un objet composé. Pour éviter ce problème particulier, nous avons étendu légèrement le mécanisme défini ci-dessus.

Afin de simuler une pile de protocoles à l'aide d'une liste d'extensions, par exemple dans un talon ou un squelette, on souhaiterait pouvoir représenter chaque élément de la pile de protocoles à l'aide d'une extension ayant deux méthodes :

- la méthode `send` encapsule les données transmises par la couche supérieure, et transmet le résultat à la couche inférieure en appelant `dsuper.send` ;
- la méthode `recv` décapsule les données en provenance de la couche inférieure, et *devrait* les transmettre à la méthode `recv` de la couche supérieure.

Mais, si on place plusieurs extensions $e_1, ..., e_n$ de ce type dans un objet composé (cf. figure 3.12 page suivante), alors un appel à `recv` sera exécuté dans le même ordre qu'un appel à `send` (c'est à dire par e_n, puis par e_{n-1} ... jusqu'à e_1), alors qu'il devrait normalement être exécuté dans l'ordre inverse.

Une façon de résoudre le problème est de découper chaque extension en deux, et de placer les extensions de réception dans l'ordre inverse des extensions d'émission. Mais

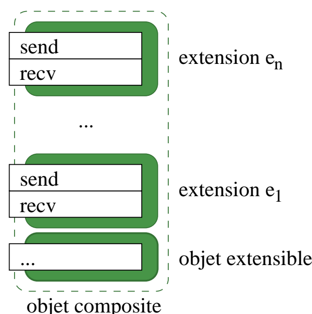


FIG. 3.12 – Simulation d'une pile de protocoles avec des extensions

ce n'est pas très pratique. Nous avons donc préféré introduire le concept de méthode *inversée*. Une méthode inversée est comme une méthode normale, sauf en ce qui concerne la sémantique du modèle de composition :

- ainsi, un appel à une méthode inversée `m` sur un objet composé est exécuté par le *premier* membre de la liste dans lequel cette méthode est définie ;
- de même, un appel à un méthode inversée `m`, en utilisant une forme d'appel spéciale notée `dsub.m`, est exécuté par le *premier* membre de l'objet composé dans lequel cette méthode est définie, et qui se situe *après* le membre appelant.

Ce concept permet de résoudre très simplement le problème des piles de protocoles : il suffit que les méthodes `recv` soient inversées.

Remarques diverses

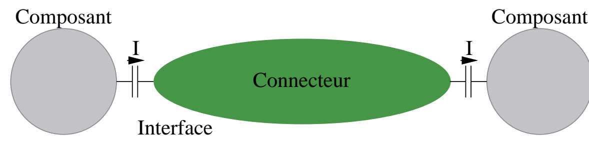
- les membres d'un objet composé ne peuvent pas être eux-mêmes des objets composés. Cette restriction permet de simplifier la mise en œuvre, sans restreindre les possibilités du modèle (remplacer un membre composé par la liste de ses membres permet de se ramener à un modèle non hiérarchique, sans changer la sémantique de l'objet composé hiérarchique initial) ;
- pour des raisons de cohérence, un *thread* ne doit pas être capable de modifier les membres d'un objet composé pendant qu'un *autre thread* exécute une méthode de cet objet composé. Cette règle n'empêche pas, et c'est voulu, une méthode d'un objet composé de modifier l'objet lui-même ;
- une méthode ne peut pas être inversée pour un membre et normale pour d'autres : soit elle l'est pour tous, soit pour aucun.

3.4 Résumé de notre proposition

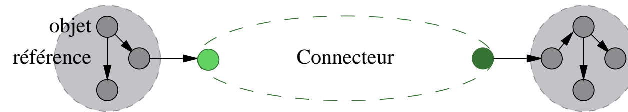
La figure 3.13 page suivante résume l'architecture proposée, ainsi que les correspondances entre le modèle de programmation, la vision du programmeur, l'architecture de haut niveau de la plate-forme, et enfin l'architecture complète.

Cette architecture permet d'offrir aux applications un modèle de programmation

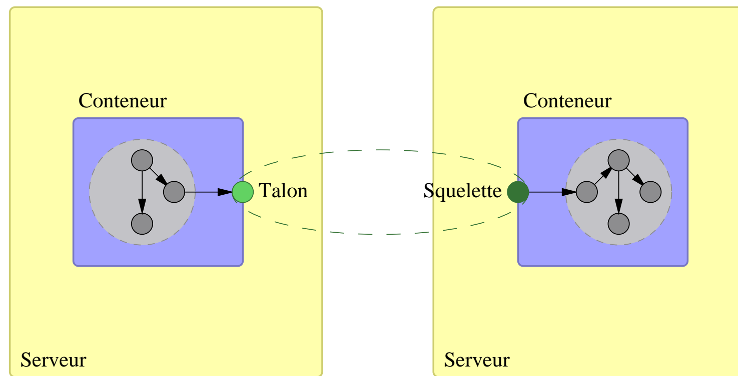
Modèle de l'application



Point de vue du programmeur



Architecture de haut niveau



Architecture détaillée

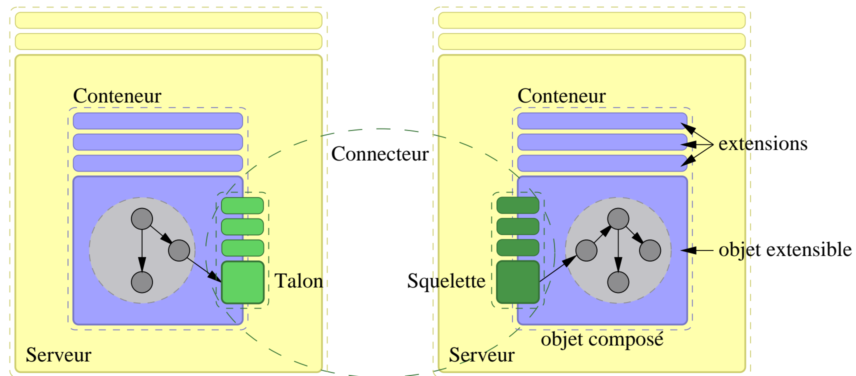


FIG. 3.13 – Résumé de notre proposition

inspiré du modèle ODP. La différence principale est que nous utilisons des références de connecteur à la place des références d'interface, car ces dernières ne sont pas très pratiques quand on souhaite adapter les propriétés non-fonctionnelles des applications. L'architecture elle-même est inspirée du modèle EJB : elle est basée en effet sur des serveurs, des conteneurs, des talons et des squelettes. Elle va cependant plus loin que le modèle EJB, car *tous* ces éléments d'architecture sont en fait des objets composés, construits à l'aide d'un mécanisme de composition nouveau, inspiré de l'héritage de classes, mais transposé aux objets.

Chapitre 4

Mise en œuvre

L'architecture présentée dans le chapitre précédent a été conçue de façon à ce qu'une plate-forme réalisée selon ce modèle soit *capable* d'atteindre nos objectifs. Mais cela ne signifie pas que toute plate-forme mise en œuvre selon ces principes atteigne automatiquement nos objectifs : encore faut-il utiliser cette architecture « correctement ». La plate-forme JavaPod, présentée dans ce chapitre, est justement une tentative dans ce sens. L'architecture détaillée de cette plate-forme est un facteur au moins aussi important que son architecture générale (i.e. celle présentée dans le chapitre précédent) en ce qui concerne ses possibilités et ses limites vis-à-vis de nos objectifs et est donc, à ce titre, partie intégrante de la proposition faite dans cette thèse pour tenter d'atteindre nos objectifs.

Ce chapitre présente donc la plate-forme JavaPod, en se basant sur une présentation du rôle des classes et des méthodes principales du code de cette plate-forme. A l'issue de ce chapitre, notre contribution étant alors complètement présentée, nous l'évalueront dans les deux chapitres suivants, consacrés aux expériences avec l'application BAGHERA et à leurs résultats.

4.1 Le langage ejava

Le langage ejava est une extension de Java qui permet de programmer facilement des objets composés, tels que définis dans la section 3.3.3 page 50. Il a été utilisé pour programmer la plate-forme JavaPod, et c'est pourquoi nous le présentons ici, avant de présenter la plate-forme elle-même.

4.1.1 Motivation

On peut parfaitement se passer d'un nouveau langage pour programmer des objets composés[1]. Mais il faut alors simuler ces objets dans un langage classique, à l'aide de *design patterns*. Par exemple, une façon simple de faire, mais pas très efficace ni

[1] c'est d'ailleurs ce que l'on a fait dans un premier temps, afin de pouvoir expérimenter assez rapidement.

très pratique pour le programmeur, consiste à réifier les appels de méthodes sur les objets composés, comme le montre la figure 4.1 : chaque membre contient un lien vers le membre suivant, ainsi qu'une méthode chargée d'exécuter les appels de méthodes réifiés (en essayant, comme en Self, de trouver la méthode demandée dans le membre courant puis, à défaut, dans les membres suivants).

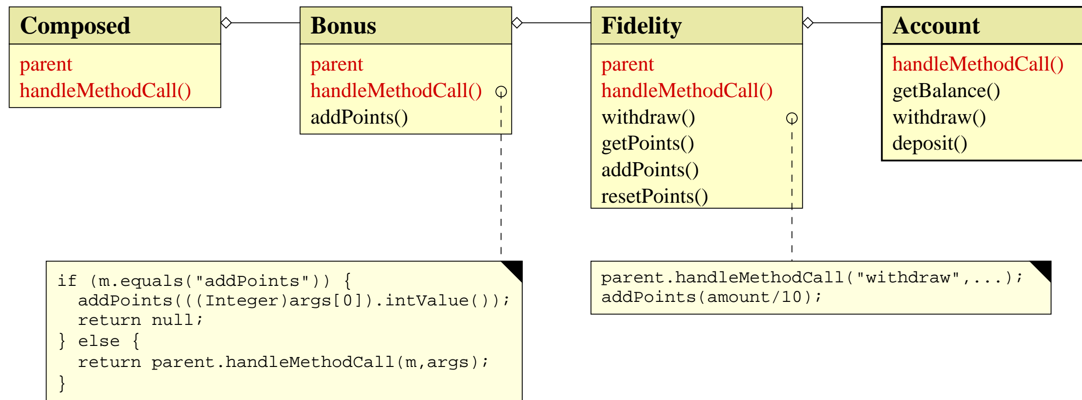


Fig. 4.1 – Simulation d'un objet composé dans un langage classique

Simuler les objets composés dans un langage classique présente l'avantage, pour le programmeur, de ne pas avoir à apprendre un nouveau langage. Mais cela ne compense pas les inconvénients de cette approche : non seulement l'utilisation de *design patterns* complique beaucoup la programmation mais, en plus, il n'est pas possible de changer ces *design patterns* sans avoir à modifier tout le code basé sur eux. A l'inverse, l'utilisation d'un nouveau langage permet de reporter la stratégie de mise en œuvre dans le compilateur, que l'on peut modifier indépendamment du code écrit dans ce nouveau langage. De plus, l'apprentissage d'un nouveau langage est assez simple, surtout si c'est une extension d'un langage existant déjà connu. C'est pourquoi nous avons choisi de définir le langage ejava.

4.1.2 Choix de conception

Nous avons choisi de définir notre langage comme une extension de Java. Ce choix élimine la possibilité de pouvoir faire apparaître les objets composés comme des objets Java normaux, à cause de la nature fortement typée de Java.

En effet, si les objets composés étaient comme des objets normaux, alors il faudrait définir le type de ces objets, qui dépend des membres qui les composent. La définition du type statique de ces objets est faisable : on peut le définir comme étant celui de leur objet extensible, invariable. Par contre, le type dynamique doit obligatoirement refléter l'ensemble des méthodes de l'objet, c'est à dire qu'il devrait être la « réunion » des types de ses membres. Mais, même en se limitant aux interfaces, c'est à dire en définissant le type dynamique d'un objet composé comme étant la réunion des interfaces de ses

membres, on se heurte au problème suivant : si `o` est un objet composé contenant un membre `p` implantant l'interface `I`, alors il devrait être légal d'écrire `I i = (I)o`. Mais si on retire ensuite `p` de `o`, le type dynamique de `i`, égal à celui de `o` puisque `i == o`, ne serait plus un sous-type du type statique de `i`, ce qui violerait la propriété la plus importante du système de typage. Plusieurs solutions sont possibles pour contourner ce problème :

- faire apparaître les objets composés comme des objets normaux, mais en interdisant la modification dynamique *directe* des objets composés. Cela obligerait à simuler ces modifications en changeant l'objet composé complet par un autre, ce qui n'est pas très pratique ;
- faire apparaître les objets composés comme des objets spéciaux, non typés statiquement, en faisant attention qu'on ne puisse pas les « mélanger » avec les objets normaux. Cela implique des modifications assez importantes et peu élégantes sur le système de typage de Java ;
- ne pas faire apparaître les objets composés en tant qu'objets concrets.

Nous avons choisi la dernière solution : les objets composés ne peuvent plus être référencés, stockés dans des variables ni passés en argument d'une méthode, ce qui élimine tous les problèmes liés au typage. Mais cela élimine également toute possibilité d'utiliser ces objets de façon *directe*. Il faut donc introduire des moyens *indirects* de les utiliser, le plus simple étant d'utiliser les membres de ces objets. Les conséquences de ce choix sont les suivantes :

- pour pouvoir référencer de façon indirecte un objet composé via une référence à l'un de ses membres, et ce sans ambiguïté, on est obligé d'imposer la contrainte suivante : *un objet ne peut appartenir à plusieurs objets composés à la fois* ;
- pour pouvoir appeler une méthode d'un objet composé de façon indirecte, via un appel de méthode sur l'un de ses membres, on doit imposer la sémantique suivante pour ces méthodes : *un appel de méthode sur un membre d'un objet composé a la même sémantique qu'un appel sur l'objet composé lui-même.*

Pour référencer un objet composé, le plus pratique est de référencer son objet extensible. En effet, celui-ci étant invariable, on est sûr que la référence désignera toujours le même objet composé. Quoi qu'il en soit, le deuxième point ci-dessus assure que quelque soit le membre d'un même objet composé utilisé pour appeler une méthode donnée, le résultat sera le même.

Nous avons vu dans la section 2.3 page 32 qu'il n'était pas nécessaire d'utiliser un langage réflexif pour réaliser une plate-forme réflexive, et que cela pouvait même être nuisible pour les performances. Et pourtant nous proposons dans cette section de réaliser la plate-forme JavaPod avec un langage qui, sans être véritablement réflexif, entraîne un surcoût similaire à celui d'un langage réflexif. Pour limiter ce surcoût au maximum, nous avons choisi de ne pas rendre tous les objets extensibles par défaut, c'est à dire que *seuls les objets programmés explicitement pour cela peuvent recevoir des extensions ou servir eux-mêmes d'extensions.* Le surcoût lié à l'extensibilité n'est ainsi payé que par les objets qui ont réellement besoin d'être extensibles, et qui sont en général peu nombreux par rapport au nombre total d'objets.

4.1.3 Définition succincte

Cette section introduit les principales caractéristiques de ejava en montrant comment programmer l'objet composé de la figure 3.11 page 51 à l'aide de ce langage. Une définition plus précise et plus complète est donnée dans l'annexe A page 117.

Caractéristiques principales

Les caractéristiques principales du langage ejava sont les suivantes :
- tout d'abord, comme son nom l'indique, ejava est une extension de Java. En particulier, tout programme Java, sans aucune exception, est aussi un programme ejava ayant, de plus, exactement la même sémantique ;
- la syntaxe de Java est étendue par l'introduction de quelques nouveaux mots-clefs comme `extensible`, `extension`, `dsuper`... De nouvelles formes d'appel de méthode sont également introduites ;
- les classes, les méthodes et les expressions utilisant cette nouvelle syntaxe ont une sémantique étendue par rapport à Java. Par contre, le code qui n'utilise pas cette nouvelle syntaxe garde exactement la même sémantique qu'en Java (sauf les appels de méthodes sur les objets composés) ;
- finalement, de nouvelles méthodes ont été ajoutées dans certaines classes de base, comme `Object` et `Class`, pour gérer les extensions d'un objet composé, et pour compléter les capacités d'introspection de Java.

Programmation d'un objet extensible

L'objet extensible de la figure 3.11 page 51 peut se programmer en ejava comme indiqué sur la figure 4.2.

```
public extensible interface Account {
  int getBalance ();
  void withdraw (int amount);
  void deposit (int amount);
}
```

```
public extensible class AccountImpl implements Account {
  private int balance;
  public extensible int getBalance () { return balance; }
  public extensible void withdraw (int amount) { balance += amount; }
  public extensible void deposit (int amount) { balance -= amount; }
}
```

FIG. 4.2 – Programmation de la classe `Account` en ejava

La seule différence par rapport à Java est l'utilisation du mot-clef `extensible` :
- dans l'en-tête de la classe `AccountImpl`, il signifie que les instances de cette classe peuvent recevoir des extensions dynamiquement ;
- devant les méthodes, il signifie que la sémantique de ces méthodes peut être modifiée dynamiquement si on attache à une instance de cette classe une extension

contenant une ou plusieurs méthodes extensibles de même nom et de même signature. A contrario, les méthodes héritées de `Object` (comme `toString`) n'étant pas déclarées comme `extensible`, *et n'étant pas redéfinies comme telles dans la classe* `AccountImpl`, elles ne peuvent être modifiées dynamiquement par l'ajout d'extensions ;
- dans l'en-tête de l'interface `Account`, il signifie que toutes les méthodes de cette interface sont extensibles : c'est un raccourci qui évite d'avoir à mettre le mot-clef `extensible` devant chaque méthode. Une classe qui implante cette interface est obligée de déclarer ces méthodes extensibles, sous peine d'une erreur à la compilation.

Programmation d'une extension

L'extension qui permet de surcharger la méthode `invoke`, tout en ajoutant des méthodes d'empaquetage, peut se programmer comme indiqué sur la figure 4.3.

```
public extensible interface Fidelity {
  int getPoints ();
  void addPoints (int points);
  void resetPoints ();
}
public extension class FidelityImpl implements Fidelity dextends Account {
  private int points;
  public extensible void withdraw (int amount) {
    dsuper.withdraw(amount);
    addPoints(amount/10);
  }
  public extensible int getPoints () { return points; }
  public extensible void addPoints (int points) { this.points += points}
  public extensible void resetPoints () { points = 0; }
}
```

FIG. 4.3 – Programmation de la classe `Fidelity` en ejava

Cet exemple illustre de nouvelles constructions du langage ejava, qui sont spécifiques aux classes d'extensions :
- le mot-clef `extension` dans l'en-tête de la classe `FidelityImpl` signifie que les instances de cette classe peuvent servir d'extensions.
- la notation `dsuper` est utilisée pour appeler la version « originale » de la méthode `withdraw`, surchargée par cette extension. Nous utilisons une notation différente de `super`, qui garde son sens habituel, afin d'éviter les confusions au cas où la super-classe contiendrait également une méthode `withdraw`.
- la clause `dextends`, qui contient une liste d'*interfaces*, permet de spécifier que les extensions de type `Fidelity` ne peuvent être ajoutées dans un objet composé `o` que si, pour chaque interface de cette clause, il existe dans `o` un membre situé avant l'extension qui implante cette interface. Ceci est vérifié à l'exécution. Mais cette clause sert également à des vérifications statiques : la notation `dsuper` ne

peut en effet être utilisée que pour appeler des méthodes définies dans l'une des interfaces de cette clause. Remarquez que cette clause ne réintroduit pas de phénomène d'explosion combinatoire.

Utilisation des objets extensibles et des extensions

La figure 4.4 montre comment utiliser les classes précédentes :

- l'instantiation de ces classes, lignes 1 et 4, se fait tout à fait normalement ;
- la ligne 5 permet d'ajouter une extension à l'objet extensible, grâce à la méthode `setExtensions`, définie dans la classe `Object`, et qui prend en argument une liste ordonnée d'extensions. Une méthode `getExtensions`, elle aussi définie dans la classe `Object`, permet de faire l'inverse ;
- ligne 6, l'appel à `withdraw` provoque un appel à `addPoints`, alors que ce n'était pas le cas ligne 3. Ceci est du au fait que l'extension `Fidelity` a été ajoutée entre temps, et montre que les extensions permettent bien de surcharger des méthodes dynamiquement. Remarquez que si on avait utilisé `f` à la place de `a` pour cet appel, on aurait obtenu exactement le même résultat ;
- la ligne 7 montre comment appeler une méthode sur un objet composé alors que cette méthode n'est pas définie dans le membre utilisé. On indique pour cela, avant le nom de la méthode, le nom d'une classe ou d'une interface dans lequel cette méthode est définie. Cette syntaxe permet tout d'abord de distinguer ces appels des appels normaux, et permet ainsi de relaxer, pour ces appels seulement, la contrainte Java selon laquelle on ne peut appeler une méthode que si elle est définie dans le type statique de l'objet appelé. Cette syntaxe permet également au compilateur de vérifier si le type des arguments correspond bien au type des paramètres formels de la méthode.

```
1  AccountImpl a = new AccountImpl();
2  a.deposit(1000);
3  a.withdraw(100);
4  FidelityImpl f = new FidelityImpl();
5  a.setExtensions(new Object[] {f});
6  a.withdraw(100);                          // équivalent à f.withdraw(100);
7  System.out.println(a.Fidelity::getPoints()); // imprime "10" (et non pas "20")
```

FIG. 4.4 – Exemple d'utilisation des classes `Account` et `Fidelity`

L'exemple de la figure 3.11 page 51 ne contenant pas de méthodes inversées, nous n'en avons pas parlé dans cette section. En fait, ces méthodes sont déclarées tout simplement à l'aide d'un nouveau mot-clef, le mot-clef `reversed`. Consultez l'annexe A page 117 pour plus de précisions à ce sujet.

4.1.4 Mise en œuvre

Le langage ejava a été mis en œuvre en modifiant le compilateur KOPI [KOP], un compilateur Java écrit en Java et disponible sous licence GNU. Le compilateur

ejava ainsi obtenu produit du *bytecode* Java qui peut s'exécuter sur n'importe quelle machine virtuelle Java conforme aux spécifications *Java 2 Platform Standard Edition* (il y a en fait quelques problèmes de portabilité - cf. section A.3 page 120). Nous donnons dans cette section quelques indications sur ce compilateur. Une description complète prendrait trop de place, et ne serait de toute façon pas pertinente : ce que nous proposons est avant tout une architecture, pas une mise en œuvre.

Processus de compilation

Le compilateur ejava [EJA] produit directement du *bytecode*, et non pas des sources Java qu'il faudrait ensuite compiler avec un compilateur Java. La raison principale de ce choix est qu'une compilation directe permet de produire des programmes plus efficaces qu'avec une compilation en deux phases. Une autre raison est que le processus de compilation lui-même est plus rapide en utilisant une phase au lieu de deux. Enfin, la dernière raison est qu'une compilation directe permet de conserver des informations de débogage, comme les numéros de lignes, qui seraient perdues avec un processus en deux temps. Ce choix n'a malheureusement pas que des avantages. Par exemple, une compilation en deux phases est plus modulaire, car on peut facilement changer le compilateur Java utilisé pour la deuxième phase sans changer le compilateur ejava lui-même.

Traduction des méthodes extensibles

L'idée de base pour simuler la surcharge dynamique des méthodes extensibles est de *rediriger* les appels à ces méthodes vers le membre approprié de l'objet composé. Ainsi, quelque soit le membre utilisé pour appeler une méthode extensible, l'appel est toujours redirigé vers le même membre, tel que défini par la sémantique de notre modèle de composition.

Cette redirection est réalisée en traduisant chaque méthode extensible par deux méthodes : le code original de chaque méthode extensible `m` est reporté dans une *méthode auxiliaire* `$m$`[2] de même signature, et remplacé par un code de redirection. La figure 4.5 page suivante illustre ce principe.

Une méthode `m` peut être surchargée dynamiquement par une extension de n'importe quel type, pourvu qu'elle contienne une méthode `m` de même signature. Par conséquent, le type le plus général possible permettant de stocker le membre `$mHandler$` vers lequel l'appel doit être redirigé est une interface, notée `Im`, ne contenant que la méthode auxiliaire `$m$`. Le principe de traduction ci-dessus *requiert* donc de générer une *interface auxiliaire* pour chaque *type* de méthode extensible (le type d'une méthode est constitué de son nom et de sa signature).

Cette méthode de traduction a été choisie car elle est plus efficace à l'exécution que celle indiquée dans la figure 4.1 page 56. Elle permet en effet d'éviter les opérations de réification des appels de méthode extensibles, qui sont assez coûteuses. En terme de

[2] tous les noms générés par le compilateur sont, par convention, de la forme $...$, pour réduire les risque de conflits avec les noms définis par l'utilisateur.

```
public class C implements Im {
  private Im $mHandler$;
  public void m (int i, int j) {
    $mHandler$.$m$(i,j); // redirection de l'appel
  }
  public void $m$ (int i, int j) {
    // code original de m
  }
}
```

FIG. 4.5 – Principe de base pour la traduction des méthodes extensibles

volume de code généré, par contre, elle peut être moins efficace, étant donné qu'il faut générer une interface auxiliaire par type de méthode extensible.

Remarques

- le principe de traduction ci-dessus requiert en fait de générer une *et une seule* interface auxiliaire par type de méthode extensible. De plus, les interfaces auxiliaires correspondant à deux méthodes différentes doivent évidemment avoir des noms différents. Autrement dit, le nom d'une interface auxiliaire doit dépendre de façon biunivoque du type de la méthode auxiliaire correspondante. Pour cela, le nom d'une interface auxiliaire est défini grâce à des conventions qui permettent d'encoder ce type en un identificateur Java. Ainsi, dans l'exemple ci-dessus, le nom de l'interface auxiliaire n'est en fait pas `Im`, mais `mElIIErV`, dérivé de `m(II)V` en utilisant un caractère d'échappement pour encoder les caractères tels que « `(` » et « `)` », lui-même dérivé de `void m (int, int)`. Nous utiliserons néanmoins dans la suite des noms du type `I`*m* pour les interfaces auxiliaires, pour simplifier ;
- ces conventions de nommage étant par nécessité biunivoques, on peut reconstituer le nom et la signature d'une méthode extensible à partir du nom d'une interface auxiliaire. Par conséquent, on peut reconstituer une interface auxiliaire complète à partir de son seul nom. Cette propriété permet de ne pas générer de fichiers `.class` pour les interfaces auxiliaires lors de la compilation. A la place, on utilise un `ClassLoader` étendu qui permet de reconstituer à la volée le *bytecode* de ces interfaces. Nous utilisons pour cela le paquetage `asm` [ASM], une librairie Java de notre cru qui permet de générer assez facilement du *bytecode* Java, et ce de façon bien plus rapide qu'avec les librairies existantes du même genre, comme par exemple `jas` [JCE].

Traduction de `dsuper`

Selon notre modèle de composition, un appel de méthode du type `dsuper.m` doit être exécuté par le dernier membre de l'objet composé dans lequel `m` est définie, et qui se situe avant l'extension courante. Autrement dit, compte tenu des principes de traduction déjà exposés, un appel de ce type doit être traduit par un appel à la méthode auxiliaire `$m$` sur le membre en question. L'expression `dsuper.m(i,j)` est donc traduite

par les instructions[3] de la figure 4.6, où `$mSuperHandler$` est un champ de type `Im` ajouté à la classe traduite, et qui contient une référence vers le membre mentionné précédemment. Ce membre peut être nul si l'extension courante ne fait pas partie d'un objet composé, auquel cas l'appel échoue en lançant une exception.

```
if ($mSuperHandler$ == null) {
  throw new UndefinedMethodException("dsuper.m");
} else {
  $mSuperHandler$.$m$(i,j);
}
```

FIG. 4.6 – Traduction de l'expression `dsuper.m(i,j)`

Traduction des expressions du type *o.C*::*m*(...)

Les appels de méthode du type `o.C::m(...)`, dits *étendus*, doivent être traduits par un appel à une méthode auxiliaire sur le membre approprié de l'objet composé dont `o` fait partie. Le problème est de déterminer ce membre.

La méthode utilisée pour les appels normaux (i.e. non étendus), qui consistait à stocker une référence vers ce membre dans un champ de `o`, n'est pas utilisable dans le cas des appels étendus, puisqu'ils peuvent faire référence à des méthodes définies bien après que la classe de `o` ait été compilée. La seule solution est d'ajouter dans `o` une référence vers une structure initialisée *à l'exécution*, et permettant de trouver le membre vers lequel un appel de méthode doit être redirigé, en fonction du nom de cette méthode. Une solution simple est d'utiliser pour cette structure une `Hashtable`. Mais ce n'est pas très efficace.

Nous avons donc utilisé la solution suivante. Considérons un objet extensible de classe `C` avec une méthode extensible `m`, et muni d'une extension de classe `D` redéfinissant la méthode `m` et ajoutant la méthode `n`. La « table de méthodes » correspondant à cet objet composé est alors représentée par un objet dont la classe, dite *composée*, est définie figure 4.7 page suivante.

Pour traduire un appel étendu du type `c.D::n(b)`, il suffit alors d'utiliser l'expression `((In)c.$composition$).$n$(b)`, si on suppose que `$composition$` est un champ de `c`, de type `ComposedObject`, et contenant une référence vers une instance de la classe `CD`.

Il est ici indispensable de générer les classes composées de façon dynamique, en fonction des besoins. C'est à dire que ces classes doivent être générées lorsque l'on modifie les extensions d'un objet composé, si la classe composée nécessaire pour représenter la table de méthode associée n'existe pas encore. En effet, si on voulait générer ces classes statiquement, il faudrait prévoir toutes les combinaisons possibles, et on retomberait sur un problème d'explosion combinatoire.

[3]une instruction n'étant pas utilisable au sein d'une expression en Java, cette règle de traduction n'est utilisable que si on produit directement du *bytecode* Java, où les notions d'expressions et d'instructions n'existent plus.

```
public class CD extends ComposedObject implements Im, In {
  private C c;
  private D d;
  public CD (C c, D d) {
    this.c = c;
    this.d = d;
  }
  public void $m$ (int i, int j) {
    d.$m$(i,j);
  }
  public void $n$ (byte b) {
    d.$n$(b);
  }
}
```

FIG. 4.7 – Représentation des tables de méthodes

Comme pour les interfaces auxiliaires, les noms des classes composées sont en fait plus compliqués que ceux utilisés dans les exemples. Ils permettent de retrouver à partir d'un nom la liste des classes à composer, et ce sans ambiguïté. A partir de là, il est facile de générer automatiquement le code complet de la classe composée correspondante (nous utilisons pour cela le paquetage `asm`, comme pour les interfaces auxiliaires).

Synthèse

La figure 4.8 page ci-contre montre une version Java *simplifiée* du *bytecode* produit par le compilateur ejava pour les classes `AccountImpl` et `FidelityImpl` (cf. figures 4.2 page 58 et 4.3 page 59), ainsi que l'interface auxiliaire et la classe composée correspondantes, générées elles de façon dynamique. On peut facilement y reconnaître l'utilisation des règles de traduction précédentes. Seul le code de redirection des méthodes extensibles est un peu plus sophistiqué que ce qui a été présenté ci-dessus :

- il utilise en effet la « table de méthodes » de l'objet composé et non pas un champ ajouté à la classe traduite, pour éviter les redondances ;
- il est de plus synchronisé par un algorithme de type lecteurs-rédacteurs, qui permet d'assurer que la composition de l'objet ne peut pas être modifiée pendant qu'un « lecteur » utilise une méthode de cet objet, tout en autorisant plusieurs « lecteurs » à utiliser l'objet composé simultanément. Cet algorithme est mis en œuvre en utilisant une variante de l'algorithme de Peterson, c'est à dire qu'il est basé sur une boucle d'attente active. Dans le cas où les modifications sont peu fréquentes, cette méthode est en effet plus efficace que d'utiliser les blocs `synchronized` de Java.

4.2 Le noyau JavaPod

Selon l'architecture présentée dans le chapitre 3 page 35, les talons, squelettes, conteneurs et serveurs sont en fait des objets composés, c'est à dire constitués d'un objet

```
public class AccountImpl implements Account, IgetBalance, Iwithdraw, Ideposit {
  private int balance;
  private ComposedObject $composition$;
  public void withdraw (int amount) {
    Synchro.getReaderLock();
    try {
      return ((Iwithdraw)$composition$).$withdraw$();
    } finally {
      Synchro.releaseReaderLock();
    }
  }
  public void $withdraw$ (int amount) { balance -= amount; }
  // ...
}

public class FidelityImpl implements Iwithdraw, IgetPoints, IaddPoints, ... {
  private int points;
  private ComposedObject $composition$;
  private Iwithdraw $withdrawSuperHandler$;
  public void withdraw (int amount) { // meme code que dans la classe AccountImpl }
  public void $withdraw$ (int amount) {
    if ($withdrawSuperHandler$ == null) {
      throw new UndefinedMethodException("dsuper.withdraw");
    } else {
      $withdrawSuperHandler$.$withdraw$(amount);
    }
    addPoints(amount/10);
  }
  // ...
}

public interface Iwithdraw {
  void $withdraw$ ();
}

public class AccountImplFidelityImpl extends ComposedObject implements ... {
  private AccountImpl a;
  private FidelityImpl f;
  public AccountImplFidelityImpl (AccountImpl a, FidelityImpl f) {
    this.a = a;
    this.f = f;
  }
  public int $getBalance$ () { return a.$invoke$(); }
  public void $withdraw$ (int amount) { f.$withdraw$(); }
  public void $deposit$ (int amount) { a.$withdraw$(); }
  // ...
}
```

Fig. 4.8 – Traduction des classes `AccountImpl` et `FidelityImpl` en Java

extensible et de plusieurs extensions. La plate-forme JavaPod, programmée en ejava en suivant ces principes architecturaux, est donc composée d'un noyau, qui regroupe le code des quatre types d'objets extensibles précédents, et d'une série d'extensions de ce noyau.

Cette section présente le noyau de la plate-forme JavaPod, à travers une présentation du rôle de ses classes et de ses méthodes principales. Celles-ci se contentent en fait de définir un *framework* pour les extensions du noyau : elles ne fournissent aucune fonctionnalité, aucun protocole, ni aucun service système. Toutes ces fonctionnalités sont en effet fournies par les extensions du noyau, qui sont présentées dans les sections suivantes.

4.2.1 Organisation générale

Les classes et interfaces du noyau JavaPod se divisent en trois catégories :

- la première catégorie est constituée de classes destinées au programmeur d'applications. Elle contient la classe `Component` et l'interface `JavaPodInterface` ;
- la deuxième catégorie, la plus importante, définit les classes extensibles `StubImpl`, `SkeletonImpl`, `ContainerImpl` et `ServerImpl`, spécifiées par les interfaces `Stub`, `Skeleton`, `Container` et `Server`, et qui correspondent aux concepts de talon, squelette, conteneur et serveur ;
- la troisième catégorie est constituée de classes utilitaires, parmi lesquelles on trouve la classe `Extension`, qui est la super-classe de toutes les extensions du noyau JavaPod, ainsi que la classe `ExtensionSet`, qui représente un ensemble non ordonné d'objets de type `Extension`.

Les sections suivantes présentent plus en détails les classes et interfaces des deux premières catégories.

4.2.2 Les classes `Component` et `JavaPodInterface`

Les composants applicatifs qui s'exécutent sur la plate-forme JavaPod peuvent être des graphes d'objets Java totalement arbitraires, même non connexes, aux deux exceptions suivantes près :

- un et un seul objet dans ce graphe doit être une instance de la classe `Component`, ou d'une sous-classe de cette classe. Cet objet sert d'interface entre le composant et son conteneur. Par exemple, si le composant doit fournir des *upcalls* au conteneur, alors ces *upcalls* doivent être implantés dans cet objet ;
- les types des interfaces d'accès aux composants doivent tous être des interfaces Java héritant de l'interface `JavaPodInterface`. Cette interface joue un rôle similaire à l'interface `Remote` dans JavaRMI.

Les classes `Component` et `JavaPodInterface` sont des classes Java et non pas ejava. Plus généralement, les composants applicatifs peuvent être programmés entièrement en Java : l'utilisation de ejava n'est requise que pour la programmation de la plate-forme JavaPod elle-même.

4.2.3 La classe `ServerImpl`

La classe `ServerImpl` est pratiquement vide. Ses méthodes extensibles sont les suivantes :

- `Container getContainer (Component c)` : retourne le conteneur encapsulant le composant `c` ;
- `Container createContainer (Component c, ExtensionSet exts)` : construit un conteneur pour le composant `c`, lui associe les extensions `exts`, et retourne le résultat ;
- `Extension[] sort (ExtensionSet exts)` : trie les extensions `exts` de telle sorte qu'une fois composées dans cet ordre, elles aient la sémantique « souhaitée ». La sémantique d'un objet composé dépend en effet de l'ordre de ses extensions.

La méthode `sort` ne fait rien par défaut, car elle ne peut pas et ne doit pas connaître les extensions du noyau, et donc à fortiori comment les ordonner.

4.2.4 La classe `ContainerImpl`

La classe extensible `ContainerImpl` correspond au concept de conteneur. La plupart de ses méthodes, extensibles, sont dédiées à la gestion des talons et des squelettes du conteneur[4] :

- `Stub createStub (String itf, ExtensionSet exts)` : crée un talon pour un *nouveau* connecteur. Plus précisément, cette méthode crée un talon extensible implantant l'interface applicative `itf`, lui ajoute les extensions `exts`, et retourne le résultat ;
- `Stub attachStub (Reference ref, String itf)` : cette méthode crée un talon pour un *connecteur existant*, désigné par sa référence. Plus précisément, cette méthode crée un talon extensible implantant l'interface applicative `itf`, lui ajoute les extensions retournées par `ref.getStubExtensions(itf)` (cf. section 4.2.6 page 69), et retourne le résultat (sauf si un talon identique existe déjà, auquel cas ce dernier est retourné directement) ;
- `void addStub (Stub stub)` : ajoute un talon dans la liste des talons du conteneur. Cette méthode est appelée notamment par les deux méthodes précédentes ;
- `Enumeration getStubs ()` : retourne la liste des talons du conteneur.

Les méthodes `createSkeleton`, `attachSkeleton`, `addSkeleton` et `getSkeletons` sont des méthodes similaires pour les squelettes. La classe `Container` contient enfin deux méthodes extensibles pour importer et exporter des références de connecteurs. La sémantique de ces méthodes découle directement de la définition de ces opérations donnée dans la section 3.1.4 page 41 :

- `Object importReference (Reference ref, String itf)` : retourne un objet implantant l'interface `itf` et « correspondant » au connecteur désigné par `ref`. Plus précisément, si le conteneur contient un talon de type `itf` appartenant au connecteur `ref`, alors ce talon est retourné. Sinon, si le conteneur contient un squelette de type `itf` appartenant au connecteur `ref`, alors l'objet applicatif

[4] certains arguments non essentiels ont été omis pour simplifier la présentation de certaines méthodes.

attaché à ce squelette est retourné. Dans les autres cas, cette méthode retourne `attachStub(ref,itf)`;
- `Reference exportReference (Object o, String itf)` : retourne la référence d'un connecteur « correspondant » à l'interface `itf` de l'objet `o`. Plus précisément, si `o` est un talon, alors la référence du connecteur auquel il appartient est retournée. Sinon, si le conteneur contient un squelette d'interface `itf` et correspondant à l'objet `o`, alors la référence du connecteur auquel il appartient est retournée. Dans les autres cas, un squelette est créé avec la méthode `createSkeleton(o,itf,null)`, puis la référence du connecteur ainsi créé est retournée.

4.2.5 Les classes `StubImpl` et `SkeletonImpl`

Le noyau JavaPod inclut un compilateur de talons, dont le rôle est de compiler une interface en une classe de talon implantant cette interface. Ce compilateur fonctionne de façon assez classique, en générant une sous-classe de la classe de talon générique `StubImpl` (voir ci-dessous). La différence principale par rapport aux compilateurs de talons « classiques » est que, comme dans FlexiNet [HHD98], ce compilateur est utilisé de façon dynamique : les classes de talons sont générées à la volée (là encore grâce au paquetage `asm`), quand il faut instancier un talon pour une interface donnée et que la classe nécessaire n'a pas encore été générée. Le compilateur de talons devient ainsi complètement transparent pour le programmeur d'applications. Comme dans JavaRMI 1.2, il n'y a pas de compilateur de squelettes, car les possibilités d'introspection de Java permettent de s'en passer.

Les classes `StubImpl` et `SkeletonImpl` contiennent les trois méthodes extensibles suivantes, dont l'implantation par défaut[5] consiste à retourner un champ initialisé par le constructeur :
- `String getInterface ()` : retourne le nom de l'interface applicative implantée par cette *porte*, c'est à dire par ce talon ou ce squelette;
- `Container getContainer ()` : retourne le conteneur auquel appartient cette porte;
- `Reference getReference ()` : retourne la référence du connecteur auquel appartient cette porte.

La classe `StubImpl` contient en plus la méthode extensible `Object invoke (Method m, Object[] args)`, dont le rôle est de retourner le résultat de l'appel réifié passé en argument. Par défaut, cette méthode lance une exception `"Not implemented"`, puisque la sémantique précise de cette méthode dépend du type du connecteur auquel appartient le talon. Le compilateur de talons implante chaque méthode de l'interface applicative du talon en utilisant la méthode `invoke`. Par exemple, la méthode applicative `int getBalance ()` est implantée par le code de la figure 4.9 page ci-contre.

Outre les trois méthodes ci-dessus, la classe `SkeletonImpl` contient la méthode extensible `Object getObject ()`, dont le rôle est de retourner l'objet applicatif *associé* au squelette (dans un connecteur client-serveur, par exemple, c'est l'objet qui implante réellement les fonctions du serveur). Enfin, la classe `SkeletonImpl` contient également

[5] c'est à dire tant qu'elle n'est pas surchargée par des extensions.

```
public class AccountStub extends StubImpl implements Account {
  private static Method GET_BALANCE = ... // objet representant la methode getBalance
  public int getBalance () {
    return ((Integer)invoke(GET_BALANCE,null)).intValue();
  }
  // ...
}
```

Fig. 4.9 – Exemple de code généré par le compilateur de talons JavaPod

la méthode `invoke`, dont l'implantation par défaut consiste ici à appeler la méthode passée en paramètre sur l'objet associé au squelette.

4.2.6 L'interface `Reference`

L'interface `Reference` correspond au concept de référence de connecteur. Selon la définition de la section 3.1.3 page 40, une référence de connecteur désigne un connecteur particulier, et décrit la façon de construire de nouvelles interfaces (au sens « point d'accès ») pour le connecteur désigné, selon leur type et leur sens. Or les interfaces d'un connecteur sont représentées, dans notre architecture, par des talons et des squelettes composés d'un objet extensible et de plusieurs extensions. Autrement dit, pour décrire la façon de construire une nouvelle interface, il suffit d'indiquer les extensions à assembler pour obtenir un talon ou un squelette représentant cette interface. L'interface `Reference` définit donc les deux méthodes suivantes :

– `ExtensionSet getStubExtensions (String itf)` : retourne les extensions à assembler pour obtenir un talon représentant une interface de sortie de type `itf` pour le connecteur désigné par cette référence ;
– `ExtensionSet getSkeletonExtensions (String itf)` : retourne les extensions à assembler pour obtenir un squelette représentant une interface d'entrée de type `itf` pour le connecteur désigné par cette référence.

Ces deux méthodes permettent non seulement de décrire la façon de construire de nouvelles interfaces, mais aussi de désigner un connecteur particulier. En effet, elles ne retournent pas des classes mais des instances d'extensions, qui peuvent donc contenir des données d'identification. Par exemple, pour un connecteur client-serveur, la méthode `getStubExtensions` retournerait des extensions implantant un talon client-serveur, *initialisées avec l'adresse du squelette serveur.*

Notez que puisque l'interface `Reference` est une interface, les références de connecteur peuvent être implantées de n'importe quelle façon, la seule contrainte étant de fournir les deux méthodes ci-dessus.

4.3 Les extensions du noyau JavaPod

Comme nous l'avons dit au début de ce chapitre, l'organisation des extensions du noyau, et notamment leur granularité et leurs interfaces, est un facteur au moins aussi

important que l'architecture définie par le noyau en ce qui concerne les capacités de notre plate-forme à atteindre nos objectifs. C'est pourquoi nous présentons dans cette section les extensions du noyau JavaPod, qui fournissent toutes les fonctionnalités de la plate-forme JavaPod elle-même.

4.3.1 Organisation générale

Les extensions du noyau JavaPod sont réparties dans plusieurs paquetages, eux mêmes organisés de façon hiérarchique :

- le paquetage `javapodx.management` contient les paquetages suivants :
  - le paquetage `admin` permet de faire exécuter tout type de tâche à des conteneurs ou à des serveurs distants,
  - le paquetage `binding` permet de compléter le modèle de composants avec une notion de *ports*, qui sont des interfaces (au sens « point d'accès ») fournies ou requises, nommées, et définies statiquement (un port est donc similaire à une *facette* d'un composant CORBA [CCM99]),
  - le paquetage `config` permet principalement de résoudre le problème de la *création* des connecteurs (cf. section 3.1.3 page 40),
  - le paquetage `console` permet d'afficher des messages d'information ou d'erreur,
  - le paquetage `deploy` permet de déployer des serveurs et des composants grâce à des scripts de configuration et de déploiement ;
- le paquetage `javapodx.protocol` fournit des protocoles de communication :
  - le paquetage `stp`, pour *server transport protocol*, spécifie un protocole d'envoi de messages entre serveurs. Une implantation par défaut de cette spécification est fournie par le sous-paquetage `stp.basic`,
  - le paquetage `gtp`, pour *gate transport protocol*, spécifie un protocole d'envoi de messages entre portes, c'est à dire entre talons et squelettes. Il contient les sous-paquetages suivants :
    - `basic` fournit une implantation de base de cette spécification,
    - `mobile` fournit une mise en œuvre utilisable entre composants mobiles,
    - `encrypted` fournit une variante avec encodage, qui ne peut fonctionner qu'au dessus d'un protocole `gtp` existant, comme `basic` ou `mobile` ;
- le paquetage `javapodx.connector` fournit différents types de connecteur :
  - le paquetage `rpc` spécifie des extensions pour connecteurs de type client-serveur. Une implantation par défaut est fournie par le sous-paquetage `rpc.basic`,
  - le paquetage `stream` spécifie des extensions pour connecteurs de type « flot de données ». Le sous-paquetage `stream.basic` fournit une mise en œuvre de cette spécification ;
- le paquetage `javapodx.service` fournit des services système implantant diverses propriétés non-fonctionnelles :
  - le paquetage `naming` définit des extensions pour optimiser l'import et l'export des références dans le cas d'un composant de type « serveur de noms »,
  - le paquetage `monitoring` définit des extensions pour présenter graphiquement, sous forme d'un arbre, le contenu d'un serveur,

- le paquetage `persistence` définit des extensions pour rendre des composants persistants,
- le paquetage `protection.acl` définit des extensions pour protéger des composants avec des listes de contrôle d'accès,
- le paquetage `protection.hsc` définit des extensions pour protéger des composants avec des capacités cachées (ou *hidden software capabilites* [HI97]),
- le paquetage `disconnection` définit des extensions pour pouvoir utiliser des composants en mode déconnecté,
- le paquetage `replication.cache` définit des extensions pour dupliquer des composants en utilisant une technique basée sur des caches [HL98],
- le paquetage `replication.abcast` définit des extensions pour dupliquer des composants en exécutant chaque opération de modification sur chaque copie, et qui utilise pour cela un protocole de diffusion de type ABCAST.

L'ensemble de ces extensions représente environ 11000 lignes de code, sans les commentaires. Par comparaison, le noyau JavaPod représente environ 1000 lignes de code.

Le reste de cette section présente les extensions qui permettent de réaliser les connecteurs de type client-serveur, ainsi que les extensions qui permettent de configurer et de déployer les composants. Le chapitre suivant présente les extensions réalisées dans le cadre des expériences avec l'application BAGHERA, et qui concernent la persistance, la protection, et le mode déconnecté. Les autres extensions, concernant notamment les connecteurs de type « flot de données », la mobilité et la duplication, ne seront pas présentées.

4.3.2 Extensions pour les connecteurs client-serveur

Les extensions qui permettent de réaliser des connecteurs de type client-serveur sont organisées en *couches*, à la façon d'une pile de protocoles. Les extensions des couches basses sont génériques et peuvent être réutilisées pour d'autres types de connecteurs. Les extensions de la couche haute sont par contre spécifiques aux connecteurs de type client-serveur.

Nous présentons ici les interfaces et le principe de fonctionnement de ces extensions sans nous préoccuper de la façon dont elles sont instanciées et déployées : ce problème est en effet examiné dans la section suivante.

Présentation

Les connecteurs de type client-serveur sont mis en œuvre à l'aide des couches `stp`, `gtp` et `rpc`[6]. Les deux premières permettent d'envoyer des messages entre serveurs et entre portes. Elles sont utilisées par la troisième pour envoyer des messages d'invocation de méthode à distance.

Chaque couche est implantée en suivant le principe bien connu consistant à séparer l'interface et l'implantation. Ainsi, on peut remplacer une implantation d'une couche

[6] il y a en réalité une couche supplémentaire, mais qui n'est pas indispensable pour la suite et que nous omettons donc pour simplifier.

par une autre sans modifier les autres couches, ou bien composer plusieurs implantations d'une même interface dans une seule couche.

La couche `stp`

La couche `stp`, pour *server transport protocol*, est spécifiée par une interface contenant les trois méthodes extensibles suivantes :

- `STPAddress getSTPAddress ()` : retourne l'adresse du serveur local ;
- `PDU stpSend (STPAddress dst, PDU data, boolean sync)` : envoie les données `data` au serveur d'adresse `dst`. Si `sync` est vrai, alors l'appel est bloquant et retourne la réponse du serveur distant. Sinon, l'appel est non bloquant et retourne `null` ;
- `PDU stpRecv (PDU data, boolean sync)` : traite les données `data` en provenance d'un serveur distant. `sync` indique si l'émetteur attend une réponse.

`PDU` et `STPAddress` sont des interfaces *vides*. Cela signifie d'une part qu'on peut envoyer par ce protocole tout type de données, et d'autre part que le format des adresses des serveurs est laissé au libre choix de l'implantation.

Le paquetage `javapodx.protocol.stp.basic` fournit une mise en œuvre de ce protocole qui utilise des *sockets* TCP/IP ainsi que le mécanisme de sérialisation de Java. Plus précisément, les méthodes précédentes sont implantées dans une extension de serveur[7] de la façon suivante :

- `STPAddress getSTPAddress ()` : retourne l'adresse du serveur local, constituée d'une adresse IP et d'un numéro de port TCP ;
- `PDU stpSend (STPAddress dst, PDU data, boolean sync)` : encapsule `data` et `sync` dans un objet qui est sérialisé puis envoyé dans une *socket* TCP vers le serveur `dst`, puis attend et retourne la réponse le cas échéant ;
- `PDU stpRecv (PDU data, boolean sync)` : cette méthode *inversée* transmet les données reçues à la couche supérieure, en appelant `dsub.stpRecv(data,sync)`.

La couche `gtp`

La couche `gtp`, pour *gate transport protocol*, est spécifiée par une interface contenant les méthodes extensibles `getGTPAddress`, `gtpSend` et `gtpRecv`, dont les rôles sont similaires à ceux des méthodes de la couche `stp`, si ce n'est qu'il s'agit cette fois de transporter des messages entre talons et squelettes.

Le paquetage `javapodx.protocol.gtp.basic` fournit une mise en œuvre de ce protocole basée sur la couche `stp`. Cette implantation par défaut utilise des adresses hiérarchiques : l'adresse d'une porte est constituée du numéro de cette porte dans son conteneur, du numéro de ce conteneur dans son serveur, et enfin de l'adresse `stp` de ce serveur. Elle utilise les trois types d'extensions suivants :

- une extension de serveur contient une table de hachage permettant de retrouver un conteneur d'après son numéro. Cette extension contient notamment les deux méthodes extensibles suivantes :

[7] c'est à dire destinée à être associée à un objet extensible de type `Server`

  - `Container gtpGetContainer (GTPAddress dst)` : retourne le conteneur contenant la porte d'adresse `dst`, trouvé en consultant la table précédente,
  - `PDU stpRecv (PDU data, boolean sync)` : « surcharge » la méthode correspondante de la couche `stp`. Cette méthode décode les données `data`, qui contiennent une adresse `gtp` et un message destiné à la couche supérieure. Elle utilise ensuite la méthode précédente pour retrouver le conteneur destinataire de ce message. Enfin, grâce à une table contenue dans ce conteneur, elle trouve la porte destinatrice, puis appelle sa méthode `gtpRecv` avec le message encapsulé ;
- une extension de conteneur contient le numéro du conteneur, ainsi qu'une table de hachage permettant de retrouver une porte de ce conteneur d'après son numéro. Elle ajoute des méthodes au conteneur permettant de consulter cette table, et surcharge certaines méthodes du conteneur, comme `addStub` et `addSkeleton`, pour attribuer automatiquement un numéro aux nouvelles portes, et pour mettre à jour la table précédente ;
- enfin, une extension de porte contient le numéro de cette porte, et lui ajoute les trois méthodes `getGTPAddress`, `gtpSend` et `gtpRecv` :
  - la méthode `getGTPAddress` retourne une adresse constituée du numéro de cette porte, du numéro de son conteneur et de l'adresse `stp` du serveur,
  - la méthode `gtpSend` encapsule les données passées en argument dans un objet qui est envoyé vers le serveur contenant la porte destinatrice grâce à la couche `stp`,
  - la méthode *inversée* `gtpRecv` se contente de transmettre les données reçues à la couche supérieure, en appelant `dsub.gtpRecv`.

La couche `rpc`

La couche `rpc`, pour *remote procedure call*, fournit des extensions permettant de construire des connecteurs client-serveur. Plus précisément, elle fournit une classe d'extension de talon, pour le coté client, et une classe d'extension de squelette, pour le coté serveur.

L'extension de talon surcharge la méthode `invoke` de la classe extensible `Stub` de façon à ce qu'elle implante l'algorithme suivant :
- les arguments de l'appel passés par référence (c'est à dire ceux dont le type hérite de `JavaPodInterface`) sont exportés en utilisant la méthode `exportReference` du conteneur ;
- un message d'invocation à distance est construit et envoyé au squelette en utilisant la couche `gtp` (il n'y a pas d'étape d'empaquetage et de dépaquetage des arguments, car cela est fait automatiquement au niveau de la couche `stp`, grâce au mécanisme de sérialisation) ;
- le message de réponse est décodé, le résultat qu'il contient est importé s'il s'agit d'une référence de connecteur, puis il est retourné à l'appelant.

L'extension de squelette « surcharge » quant à elle la méthode `gtpRecv` de la couche inférieure, de façon à traiter les messages d'invocation en provenance des clients de la

façon suivante :

- les arguments qui sont des références de connecteurs sont importées en utilisant la méthode `importReference` du conteneur ;
- la méthode dont le nom est contenu dans le message d'invocation est appelée sur l'objet associé au squelette en appelant la méthode extensible `invoke` ;
- le résultat est exporté si besoin, avant d'être inclus dans un message de réponse retourné à l'appelant.

Résumé

La figure 4.10 montre où se placent les extensions précédentes dans le cas de deux composants reliés par un connecteur client-serveur et situés dans deux serveurs différents (les talons et squelettes ne sont pas représentés à cheval sur les conteneurs, comme dans la figure 3.13 page 53, pour gagner un peu de place).

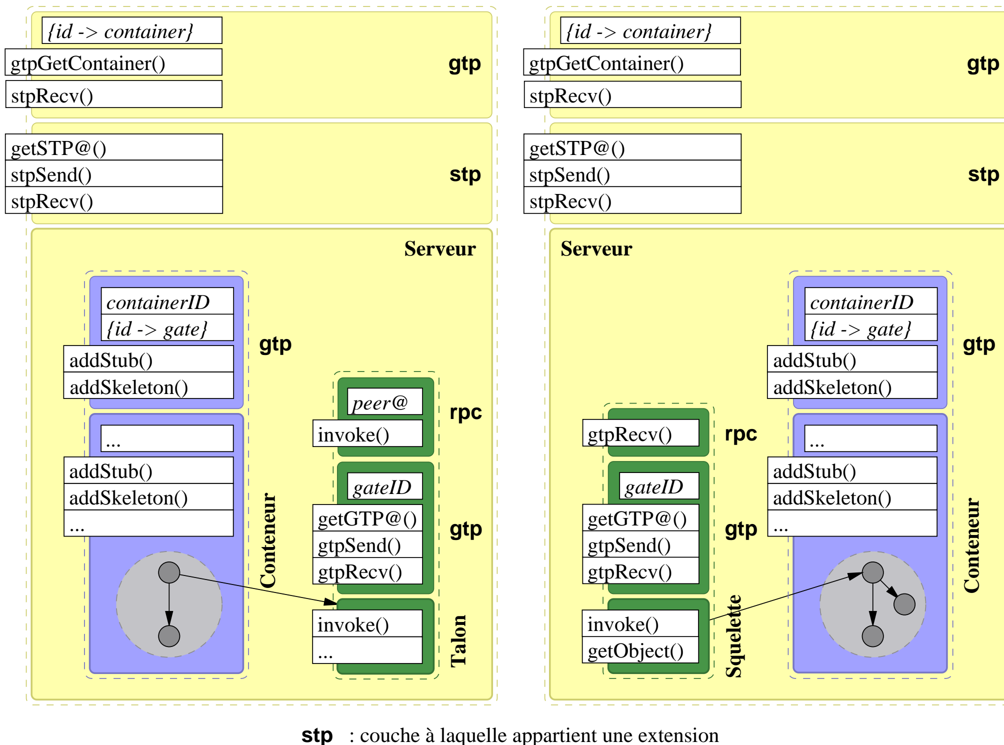


FIG. 4.10 – Résumé des extensions pour les connecteurs client-serveur.

4.3.3 Extensions pour la configuration et le déploiement

Cette section présente les extensions fournies par les paquetages `binding`, `config` et `deploy`, qui permettent de déployer des composants et de créer des connecteurs entre

eux. La section suivante montre comment on peut les utiliser à l'aide d'un exemple concret.

Le paquetage `binding`

Ce paquetage définit avant tout des conventions de programmation qui permettent au programmeur d'indiquer les *ports* d'entrée et de sortie de ses composants, un port étant une interface (au sens point d'accès) nommée. Ces conventions de programmation sont similaires dans leur principe à celles utilisées dans les *Java Beans* [JBS] :

- une méthode *Type* `get`*XXX*`InputPort ()`, dans une sous-classe de `Component`, indique que ce composant possède un port d'entrée nommé *XXX* et de type *Type*. De plus, cette méthode retourne l'objet interne du composant relié à ce port ;
- une méthode *Type* `get`*XXX*`OutputPort ()`, accompagnée d'une méthode `void set`*XXX*`OutputPort (`*Type* port), dans une sous-classe de `Component`, indique que ce composant possède un port de sortie nommé *XXX* et de type *Type*. De plus, ces méthodes permettent de modifier le talon connecté à ce port.

Ce paquetage définit ensuite une extension qui ajoute à un conteneur des méthodes extensibles pour gérer les ports du composant encapsulé :

- `getInputPorts` et `getOutputPorts` retournent les noms et les types des ports du composant ;
- `getInputPort` et `getOutputPort (String name)` retournent la porte correspondant au port `name` ;
- `createInputPort` et `createOutputPort (String name)` créent et attachent un connecteur sur le port `name` ;
- `attachInputPort` et `attachOutputPort (String name, Reference ref)` créent une nouvelle interface pour le connecteur `ref` et l'attachent sur le port `name`.

Ces méthodes utilisent les capacités d'introspection de Java pour découvrir et utiliser les méthodes du composant qui respectent les conventions de programmation ci-dessus. Elles utilisent également les méthodes `createStub`, `createSkeleton`, `attachStub` et `attachSkeleton` du conteneur.

Le paquetage `config`

La construction d'un connecteur se déroule généralement de la façon suivante[8] :

- création d'un talon ou d'un squelette *initial* ;
- obtention de la référence `ref` de cet « embryon » de connecteur ;
- construction de nouvelles interfaces grâce aux instructions codées dans `ref`.

Par exemple, un connecteur client-serveur est créé en commençant par un squelette initial, agrandi au fur et à mesure par l'ajout de talons clients. Le paquetage `config` offre des extensions qui permettent de configurer automatiquement les talons et les squelettes initiaux créés par un composant (et donc les connecteurs créés par ce composant), et ce

[8] une autre méthode consiste à créer la référence du connecteur *ex nihilo*, puis de construire toutes les interfaces du connecteur à l'aide de cette référence.

en utilisant des informations fournies dans un fichier de configuration séparé du code du composant.

Les informations fournies dans ce fichier de configuration indiquent, pour chaque type de talon et de squelette initial que le composant peut créer, les extensions à utiliser pour cette porte initiale, ainsi que les extensions à utiliser pour les portes qui permettront d'agrandir le connecteur (en fonction du sens et du type de ces portes). La figure 4.11 montre un exemple d'un tel fichier de configuration.

```
{ skel I                         // utiliser les extensions suivantes pour construire...
  { root ext1 ext2 ... }         // ...un squelette initial de type I
  { stub I ext3 ... }            // ...ses futurs talons de type I
}
{ stub J
  { root ext4 ext5 ... }         // ...un talon initial de type J
  { stub Subscribe ext6 ... }    // ...ses futurs talons de type Subscribe
  { skel J ext7 ext8 ... }       // ...ses futurs squelettes de type J
}
```

Fig. 4.11 – Exemple de fichier de configuration

Les talons et les squelettes initiaux sont créés avec les méthodes `createStub` et `createSkeleton` de la classe `Container` (cf. section 4.2.4 page 67). Ces deux méthodes ont un paramètre qui spécifie les extensions à utiliser pour construire le talon ou le squelette. Mais les clients de ces méthodes, comme par exemple `exportReference` ou `createInputPort`, utilisent toujours la valeur `null` pour ce paramètre. Autrement dit, toutes les portes initiales *devraient* normalement n'avoir aucune extension. En fait il n'en est rien, et ce grâce à une extension du paquetage `config` qui surcharge les méthodes `createStub` et `createSkeleton` du conteneur par des méthodes exécutant l'algorithme suivant :

- détermination des extensions à utiliser pour construire une porte initiale du type demandé, grâce au fichier de configuration précédent ;
- création de ces extensions *et d'une extension supplémentaire pour surcharger la méthode* `getReference` (voir ci-dessous) ;
- création de la porte initiale avec les extensions ainsi obtenues, en appelant la méthode `dsuper.createStub` ou `dsuper.createSkeleton`.

Tant qu'elle n'est pas surchargée, la méthode extensible `getReference` (cf. section 4.2.5 page 68) retourne la référence utilisée lors de la création de la porte. Pour une porte créée pour agrandir un connecteur existant, cette référence est celle du connecteur, et la méthode `getReference` n'a pas besoin d'être surchargée. Par contre, pour une porte initiale, cette référence est *nulle*. La méthode `getReference` *doit* donc être surchargée dans ce cas. L'extension mentionnée ci-dessus la surcharge de façon à retourner une référence contenant les informations concernant la construction des portes supplémentaires, issues du fichier de configuration.

En résumé, lorsqu'un composant crée une porte dite initiale, par exemple lors de l'exportation d'un nouvel objet, cette opération est interceptée par une extension du paquetage `config`, qui crée et associe automatiquement à cette nouvelle porte les exten-

sions spécifiées dans le fichier de configuration du composant. Cette extension associe également à cette porte initiale, de façon implicite, une extension qui permet d'obtenir la référence de l'embryon de connecteur, référence qui permet à son tour d'agrandir ce connecteur.

Le paquetage `deploy`

Comme son nom l'indique le paquetage `deploy` permet de déployer des applications, c'est à dire qu'il permet de créer des serveurs de composants, d'instancier des composants et leurs conteneurs dans ces serveurs, puis de relier ces composants par des connecteurs. Les fonctions offertes par ce paquetage sont très rudimentaires, mais rien n'empêche de les compléter (par exemple pour pouvoir déployer à distance une application sur plusieurs sites, au lieu d'avoir besoin de déployer « à la main » les composants sur leurs sites respectifs).

Le paquetage `deploy` ne définit pas d'extensions au sens strict, mais utilise les extensions des paquetages `admin`, `config` et `deploy` pour exécuter des *scripts de déploiement*. Le plus simple pour présenter ce paquetage est d'utiliser un exemple concret, ce qui est fait dans la section 4.4.

4.4 Exemple d'utilisation

Cette section montre comment on peut programmer et déployer une application client-serveur extrêmement simple à l'aide de la plate-forme JavaPod. La plupart des classes et des extensions présentées dans ce chapitre se retrouvent dans cet exemple, ce qui permet de mieux comprendre leur fonctionnement.

4.4.1 Programmation

L'application considérée dans cet exemple est constituée d'un composant serveur qui permet d'afficher des messages, et d'un composant client qui utilise le serveur pour faire afficher à distance le message traditionnel `"Hello world!"`.

Pour programmer cette application, il faut commencer par définir l'interface du serveur, que l'on appelle `Hello` (cf. figure 4.12 page suivante). Les seules contraintes à ce niveau sont que cette interface doit étendre `JavaPodInterface`, et que ses méthodes doivent déclarer l'exception `JavaPodException` (ces classes, définies dans le noyau JavaPod, sont similaires aux classes `Remote` et `RemoteException` de Java RMI).

Il faut ensuite programmer le serveur. Pour cela, il faut définir une sous-classe de la classe `Component` (cf. section 4.2.2 page 66) implantant l'interface `Hello`. Il faut ensuite programmer la méthode `hello`, mais aussi une méthode `Hello getServerInputPort ()` qui permet d'indiquer que le composant serveur à un port d'entrée nommé `Server` et de type `Hello` (qui est d'ailleurs relié à l'unique objet qui constitue le composant).

De même, pour programmer le client, il faut définir une sous-classe de la classe `Component` avec deux méthodes indiquant que ce composant a un port de sortie, nommé

ici `HelloServer`, de type `Hello`. On place ensuite le code effectif du client dans la méthode `run` de l'interface `Runnable`.

```
public interface Hello extends JavaPodInterface {
  void hello (String msg) throws JavaPodException;
}

public class ServerImpl extends Component implements Hello {
  public Hello getServerInputPort () { return this; }
  public void hello (String msg) { System.out.println(msg); }
}

public class ClientImpl extends Component implements Runnable {
  private Hello server;
  public Hello getHelloServerOutputPort () { return server; }
  public void setHelloServerOutputPort (Hello server) { this.server = server; }
  public void run () {
    try {
      server.hello("Hello world");
    } catch (Exception e) {
      System.out.println(e.toString());
    }
  }
}
```

FIG. 4.12 – Programmation d'une application `HelloWorld`

4.4.2 Déploiement

Le script de la figure 4.13 page suivante permet de déployer l'application précédente. Il fonctionne de la façon suivante.

Les trois premiers blocs sont des définitions de macro, un peu comme dans un *makefile*, sauf qu'ici la valeur d'une macro n'est pas une chaîne de caractères mais un arbre défini avec une notation à la Lisp.

Plus précisément, le premier bloc définit la macro `CONNECTOR`, qui spécifie comment construire le connecteur entre le client et le serveur (cf. figure 4.11 page 76). En l'occurrence, on spécifie qu'il faut assembler les extensions des couches `gtp` et `rpc`. Autrement dit, le client et le serveur doivent être reliés par un connecteur de type client-serveur.

Le second bloc définit les extensions à associer aux conteneurs qui vont recevoir les composants. On y retrouve notamment l'extension du paquetage `binding` qui permet de gérer les ports des composants, l'extension du paquetage `config` qui permet de configurer automatiquement les portes initiales des composants, et l'extension de conteneur de la couche `gtp`. Le nom de l'extension de configuration est suivi d'une référence à la macro `CONNECTOR` : cela signifie que pour construire cette extension, il faut passer à son constructeur la valeur de cette macro, autrement dit le « fichier de configuration » du composant.

Le troisième bloc définit quant à lui les extensions à associer aux serveurs. On

```
CONNECTOR = {
  skel javapod.JavaPodInterface
  { root
      javapodx.protocol.gtp.basic.BasicGTPGatex
      javapodx.connector.rpc.basic.BasicImportExportGatex
      javapodx.connector.rpc.basic.BasicServerGatex
  }
  { stub javapod.JavaPodInterface
      javapodx.protocol.gtp.basic.BasicGTPGatex
      javapodx.connector.rpc.basic.BasicImportExportGatex
      {javapodx.connector.rpc.basic.BasicClientGatex "getGTPAddress"}
  }
}

COMPONENT = {
  javapodx.management.admin.basic.BasicAdminContainerx
  javapodx.management.binding.basic.BasicPortManagerContainerx
  {javapodx.management.config.basic.BasicConfigurationContainerx $CONNECTOR}
  javapodx.protocol.gtp.basic.BasicGTPContainerx
}

SERVER = {
  javapodx.management.console.basic.BasicConsoleServerx
  {javapodx.management.admin.basic.BasicAdminServerx $COMPONENT}
  {javapodx.protocol.stp.basic.BasicSTPServerx $PORT}
  javapodx.protocol.gtp.basic.BasicGTPServerx
}

#ifdef DEPLOY_SERVER
ERR := init($SERVER,$NULL)
S   := $THIS->createComponent(examples.hello.ServerImpl,$COMPONENT,$NULL)
REF := $S->createPort(INPUT,Server,$NULL,$NULL)
ERR := bindReference($REF,"server.ref")
ERR := stop()
#endif

#ifdef DEPLOY_CLIENT
ERR := init($SERVER,$NULL)
C   := $THIS->createComponent(examples.hello.ClientImpl,$COMPONENT,$NULL)
REF := lookupReference("server.ref")
ERR := $C->createPort(OUTPUT,HelloServer,$REF,$NULL)
ERR := $C->run()
ERR := stop()
#endif
```

Fig. 4.13 – Script de déploiement

y retrouve notamment l'extension de la couche `stp`, et l'extension de serveur de la couche `gtp`. L'extension de la couche `stp` prend en paramètre un numéro de port TCP, passé dans la macro `PORT` définie dans la ligne de commande lors du lancement de l'interpréteur de script.

Les deux blocs suivants contiennent les instructions de déploiement proprement dites. Si la macro `DEPLOY_SERVER` est définie dans la ligne de commande lors du lancement de l'interpréteur, alors le composant serveur est déployé :

- la commande `init` crée un serveur et lui associe les extensions spécifiées dans la macro `SERVER` ;
- la commande `createComponent` crée un composant de type `ServerImpl` et lui associe les extensions spécifiées dans la macro `COMPONENT` ;
- la commande `createPort` exporte l'objet du composant relié au port d'entré nommé `Server`, et retourne la référence de l'embryon de connecteur ainsi obtenu ;
- la commande `bindReference` stocke cette référence dans le fichier `server.ref` ;
- la commande `stop` arrête l'interpréteur de script (mais pas le serveur qu'il a créé).

De même, si la macro `DEPLOY_CLIENT` est définie dans la ligne de commande lors du lancement de l'interpréteur, alors le composant client est déployé :

- la commande `createComponent` crée un composant de type `ClientImpl` et lui associe les extensions spécifiées dans la macro `COMPONENT` ;
- la commande `lookupReference` lit la référence stockée dans le fichier nommé `server.ref` et retourne cette référence ;
- la commande `createPort` crée un nouveau talon pour le connecteur désigné par `REF` et le connecte sur le port de sortie nommé `HelloServer` ;
- la commande `run` lance le composant client en appelant sa méthode `run`.

Le déploiement proprement dit s'effectue à l'aide des deux commandes suivantes :

- `java` ... .`ServerLauncher` `PORT=1234 DEPLOY_SERVER=true deploy.script` ;
- `java` ... .`ServerLauncher` `PORT=1235 DEPLOY_CLIENT=true deploy.script`.

Ces commandes lancent l'interpréteur de script `ServerLauncher` avec le script contenu dans le fichier `deploy.script`, et avec certaines valeurs initiales pour les macros `PORT`, `DEPLOY_SERVER` et `DEPLOY_CLIENT`.

Remarques

- les exemples du même type pour CORBA ou Java RMI incluent directement dans le code du serveur des instructions pour enregistrer le composant dans un serveur de noms, et dans le code du client des instructions pour interroger ce serveur de noms. Il serait possible de faire de même avec la plate-forme JavaPod, mais nous avons volontairement reporté ce code lié au serveur de noms dans le script de déploiement, car il s'agit en fait de code non-fonctionnel qui n'a rien à faire dans le code de l'application ;
- il est facile, simplement en modifiant le script de déploiement précédent, et sans modifier le code source de l'application, de rendre le serveur mobile, protégé ou bien répliqué.

Chapitre 5

Expérimentations avec BAGHERA

L'architecture présentée dans le chapitre 3, ainsi que l'architecture plus détaillée mise en œuvre dans la plate-forme JavaPod, présentée dans le chapitre 4, ont été conçues pour permettre d'atteindre nos objectifs qui, rappelons le, peuvent se résumer ainsi : séparation du code fonctionnel et non-fonctionnel, séparation et composition du code de chaque propriété non-fonctionnelle, modularité et extensibilité de la plate-forme elle-même.

La démarche suivie pour définir cette architecture a consisté à analyser les conséquences de nos objectifs, et elle nous assure qu'une architecture de ce type est plus ou moins *nécessaire* pour les atteindre. En revanche, elle ne nous assure pas que cette architecture est *suffisante*. Pour vérifier que notre proposition permet bien d'atteindre nos objectifs, la seule solution est d'essayer de l'utiliser pour résoudre des cas concrets. C'est ce que nous avons fait en programmant l'application BAGHERA et les trois propriétés non-fonctionnelles présentées dans le chapitre 1 : persistance, protection et mode déconnecté. Ce chapitre et le suivant présentent ces expériences et leurs résultats.

5.1 Implantation de BAGHERA

L'implantation de la version de base de l'application BAGHERA, c'est à dire de son code fonctionnel uniquement, ne présente pas de difficultés ni d'intérêt pour ce que nous voulons montrer ici. Cette section se contente donc de présenter les interfaces fonctionnelles des différents composants, certaines de leurs méthodes étant mentionnées dans le reste de ce chapitre. La figure 5.1 page suivante présente les interfaces des composants `VirtualSchool`, `Mailbox`, `ElectronicCase`, et `ExerciseRepository` (cf. figure 1.1 page 7).

Remarques

- chaque méthode déclare l'exception `JavaPodException`, bien que cela n'apparaisse pas sur la figure, pour simplifier ;
- la classe `Mail` contient trois champs : le nom de l'émetteur, le nom du destinataire,

```
public interface VirtualSchool extends JavaPodInterface {
  Vector getStudents ();
  void addStudent (String name, String professor);
  void removeStudent (String name);
  String getProfessor (String student);
  ElectronicCase getElectronicCase (String student);
  Vector getProfessors ();
  void addProfessor (String name);
  void removeProfessor (String name);
  Vector getStudents (String professor);
  Mailbox getMailbox (String professor);
  void transferStudent (String student, String professor);
  ExerciseRepository getExerciseRepository ();
}

public interface Mailbox extends JavaPodInterface {
  String getOwner ();
  Vector getMails ();
  void addMail (Mail mail);
  void removeMail (int index);
  void removeMails ();
  void addListener (MailListener listener);
}

public interface ElectronicCase extends Mailbox {
  Vector getExerciseList ();
  void addExercise (String exerciseName);
  void removeExercise (String exerciseName);
  Solution getSolution (String exerciseName);
  void proposeSolution (String exerciseName, AnnotatedFigure solution);
  void confirmSolution (String exerciseName);
  void proposeCorrection (String exerciseName, AnnotatedFigure correction);
  void confirmCorrection (String exerciseName);
}

public interface ExerciseRepository extends JavaPodInterface {
  Vector getExerciseList ();
  AnnotatedFigure getExerciseProposition (String name);
  AnnotatedFigure getExerciseProof (String name);
  void addExercise (String name, Exercise exercise);
  void removeExcercise (String name);
}
```

FIG. 5.1 – Interfaces applicatives de l'application BAGHERA

et le corps du message. L'interface `MailListener` définit la méthode `mailReceived (Mail mail)`, qui permet d'être informé de l'arrivée de nouveaux messages ;
- la classe `AnnotatedFigure` contient une figure géométrique et un texte associé, qui peut être l'énoncé d'un exercice, une solution ou une correction ;
- la classe `Solution` contient la solution proposée par un élève à un exercice, la correction de cette solution par son professeur, et un indicateur de l'état d'avancement de cette solution : pas encore confirmée par l'élève, confirmée mais pas encore corrigée par le professeur, ou bien confirmée et corrigée.

Comme nous l'avons montré dans le chapitre 1 page 5, les besoins des utilisateurs de l'application BAGHERA peuvent conduire au besoin de pouvoir associer à cette application des propriétés de persistance, de protection et d'utilisation en mode déconnecté. Le reste de ce chapitre est consacré à la mise en œuvre de ces trois propriétés non-fonctionnelles avec la plate-forme JavaPod : nous présentons, pour chaque propriété, l'algorithme utilisé, sa mise en œuvre, et son application à BAGHERA.

5.2 Persistance

5.2.1 Algorithme

Un composant persistant est un composant dont la durée de vie est supérieure à celle du serveur qui le contient, c'est à dire que son état n'est pas perdu après un arrêt volontaire ou involontaire de ce serveur. Pour assurer cette propriété, nous avons choisi l'algorithme le plus simple qui soit, c'est à dire celui qui consiste à enregistrer sur disque, dans un fichier, le composant persistant après chaque modification de ce composant en mémoire.

Plus précisément, après chaque appel de méthode à distance sur un composant persistant, on consulte une table issue d'un fichier de configuration fourni par le programmeur, afin de savoir si la méthode appelée *peut* avoir modifié l'état du composant. Si c'est le cas, alors on enregistre dans un fichier le composant, mais aussi son conteneur et les talons et squelettes de ce conteneur. En effet, les talons et les squelettes, qui représentent les interfaces du composant, et les extensions du conteneur et des talons et des squelettes, qui fournissent les propriétés non-fonctionnelles du composant et qui contiennent notamment des adresses (cf. section 4.3.2 page 72) et des informations de configuration (cf. section 4.3.3 page 75), sont aussi importantes à enregistrer que le composant lui-même.

Un inconvénient de cet algorithme est qu'il peut poser des problèmes de cohérence, c'est à dire que l'état enregistré sur disque peut ne pas correspondre à un état distribué causalement cohérent. Un autre inconvénient de cet algorithme est son inefficacité, du au fait que l'on utilise des fichiers au lieu d'une base de données, mais surtout parce que l'on enregistre le conteneur et tous ses talons et squelettes en même temps que le composant lui-même. Nous avons néanmoins choisi cet algorithme car nous avons privilégié par dessus tout la simplicité de mise en œuvre (naturellement rien n'empêche a priori de fournir une implantation plus efficace du service de persistance).

5.2.2 Mise en œuvre

L'algorithme précédent est implanté à l'aide de trois types d'extensions : une extension de serveur, une de conteneur, et une autre de squelette.

L'extension de serveur

L'extension de serveur ajoute deux méthodes extensibles au serveur pour sauvegarder un composant sur disque et charger un composant en mémoire. L'implantation par défaut de ces méthodes utilise la sérialisation de Java.

Cette extension de serveur surcharge également la méthode `gtpGetComponent` (cf. section 4.3.2 page 72). En effet, après le redémarrage d'un serveur, les composants persistants sont présents sur disque mais pas en mémoire. Par conséquent, lorsqu'un message arrive pour l'un de ces composants, la méthode `gtpGetComponent` par défaut échoue. Pour éviter cela, l'extension de serveur surcharge cette méthode de façon à charger automatiquement en mémoire, à la volée, les composants persistants qui ne s'y trouvent pas encore.

L'extension de conteneur

L'extension de conteneur d'un composant persistant contient une table qui indique, pour chaque méthode d'accès possible au composant, si elle peut ou non modifier l'état du composant. Cette table est initialisée lors de la création de cette extension en utilisant un « fichier de configuration » fourni par le programmeur. En pratique, comme dans le cas du paquetage `config`, ce « fichier de configuration » est directement intégré dans le script de déploiement, comme le montre la figure 5.2.

```
...
MODES = "
interface Account {
  int getBalance ()           : read;  // ne modifie pas le composant
  void withdraw  (int amount) : write; // peut le modifier si amount != 0
  void deposit (int amount)   : write; // idem
}"
COMPONENT = {
  ...
  {javapodx.service.persistence.basic.BasicPersistentContainerx $MODES}
}
...
```

FIG. 5.2 – Spécification du type des méthodes d'accès à un composant persistant

L'extension de squelette

L'extension de squelette surcharge la méthode `invoke` afin d'intercepter les appels de méthode à distance à destination d'un composant persistant. Cette méthode surchargée appelle tout d'abord `dsuper.invoke`, afin d'effectuer l'appel normalement. Ensuite,

si le résultat n'est pas une exception, elle utilise la table stockée dans l'extension de conteneur précédente pour déterminer si la méthode qui vient d'être exécutée peut avoir modifié l'état du composant. Si c'est le cas, elle utilise l'extension de serveur précédente pour enregistrer le composant et son conteneur dans un fichier. La figure 5.3 résume les méthodes principales des trois types d'extensions précédents.

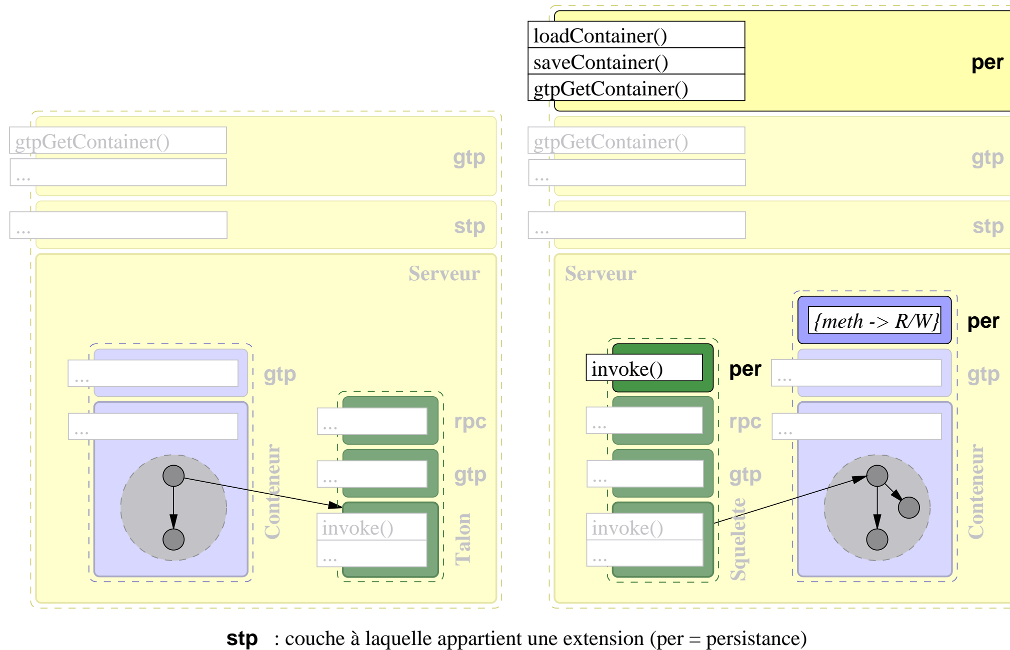


FIG. 5.3 – Résumé des extensions utilisées pour implanter la persistance

5.2.3 Application à BAGHERA

L'association de la propriété de persistance aux composants de type `VirtualSchool`, `Mailbox`, `ElectronicCase`, et `ExerciseRepository` de l'application BAGHERA est très simple, puisqu'il suffit de modifier le script de déploiement pour :

- ajouter la spécification du type (`read` ou `write`) de chaque méthode d'accès à ces composants ;
- déclarer l'extension de conteneur précédente dans la liste des extensions des conteneurs de ces composants ;
- déclarer l'extension de serveur précédente dans la liste des extensions du serveur contenant ces composants.

En ce qui concerne le premier point, les méthodes de type `write` sont tout simplement celles dont le nom ne commence pas par `get` (sauf la méthode `addListener`).

5.3 Protection

5.3.1 Algorithme

La protection d'une application consiste tout d'abord à spécifier une *politique de protection* puis, lors de l'exécution, à faire en sorte que cette politique soit respectée. Nous avons choisi de spécifier la politique de protection en utilisant des *listes de contrôle d'accès*, ce qui consiste à définir, pour chaque méthode d'accès à chaque composant de l'application, d'une part la liste des composants ayant le droit d'exécuter cette méthode, d'autre part le mode d'exécution de cette méthode :

- la liste des composants autorisés à appeler une méthode donnée, qui est justement appelée une liste de contrôle d'accès, est en fait la liste des *noms* des composants et des *groupes* de composants autorisés. On associe en effet à chaque composant un nom, et on réunit en général les composants ayant les mêmes droits par groupes (mais ce n'est pas obligatoire) ;
- le mode d'exécution d'une méthode est soit *normal*, soit *privilégié*. Une méthode normale s'exécute avec les droits de l'appelant, alors qu'une méthode privilégiée s'exécute avec les droits de l'appelé.

L'algorithme utilisé pour s'assurer que la politique de protection est bien respectée fonctionne de la façon suivante :

- on stocke dans chaque conteneur le nom de son composant, un mot de passe censé être connu de ce seul conteneur, et les listes de contrôle d'accès concernant les méthodes fournies par le composant ;
- on ajoute dans chaque message d'invocation de méthode à distance le nom et le mot de passe du composant appelant[1], sous forme chiffrée pour éviter que le mot de passe du composant ne soit intercepté sur le réseau ;
- *avant* d'invoquer une méthode sur un composant, on effectue les opérations suivantes :
  - on extrait du message d'invocation le nom et le mot de passe du composant appelant,
  - on interroge le composant authentificateur (voir ci-dessous) afin de s'assurer que le nom et le mot de passe reçus sont bien valides, ce qui assure que le message vient bien de là où il le prétend. On obtient en retour la liste des groupes auquel appartient le composant appelant,
  - on vérifie, à l'aide des listes de contrôle d'accès du composant appelé, que l'appelant à bien le droit d'exécuter la méthode demandée. Si ce n'est pas le cas, on retourne une exception sans exécuter cette méthode.

Le composant *authentificateur* stocke dans une table les noms des différents composants de l'application et, pour chaque nom, le mot de passe associé et la liste des groupes auquel appartient le composant. Le composant authentificateur offre une interface qui permet d'ajouter et d'enlever des noms dans cette table, et qui permet surtout d'authentifier les messages (en vérifiant que le nom et le mot de passe qu'ils contiennent

[1]sauf dans le cas d'un appel effectué pendant l'exécution d'une méthode non privilégiée, auquel cas on utilise le nom et le mot de passe du composant ayant appelé cette méthode non privilégiée

figurent bien dans la table précédente). Ce composant doit naturellement être lui-même protégé. Nous utilisons pour cela l'algorithme précédent, la seule différence étant l'étape d'interrogation du composant authentificateur, qui se fait ici de façon locale, et non plus en utilisant un appel de méthode à distance.

5.3.2 Mise en œuvre

L'algorithme précédent est implanté à l'aide de sept classes d'extensions, réparties en deux couches. La couche basse est une couche permettant de chiffrer les messages envoyés par la couche `gtp`. Elle n'est pas spécifique à l'algorithme précédent et peut être réutilisée dans d'autres contextes. La couche haute implante l'algorithme de protection proprement dit.

Extensions pour la cryptographie

Les deux classes d'extensions de cette couche implantent l'interface de la couche `gtp` en ajoutant des fonctions de chiffrement à une couche `gtp` inférieure, qui peut être quelconque (par exemple, il peut s'agir de la couche `gtp` standard, ou bien de celle pour composants mobiles).

La première classe d'extension, destinée aux conteneurs, stocke un couple de clefs RSA. La seconde, destinée aux talons et aux squelettes, surcharge les méthodes de l'interface `gtp` de la façon suivante :

- `getGTPAddress` ajoute à l'adresse retournée par `dsuper.getGTPAddress` la clef publique contenue dans l'extension de conteneur précédente ;
- `gtpSend` chiffre le message à envoyer avec la clef publique du destinataire, contenue dans son adresse, avant d'envoyer le message ainsi obtenu en appelant la méthode `dsuper.gtpSend` ;
- `gtpRecv` déchiffre le message reçu avec la clef privée du receveur avant de transmettre le message déchiffré à la couche supérieure en appelant `dsub.gtpRecv`.

Extensions pour la protection

L'algorithme de protection proprement dit est implanté par les cinq classes d'extensions suivantes :

- une extension de conteneur stocke le nom du composant, son mot de passe, ses listes de contrôle d'accès, et enfin une référence de connecteur permettant d'accéder au composant authentificateur. Cette extension ajoute au conteneur des méthodes extensibles permettant de consulter ces données, et d'authentifier un nom et un mot de passe en effectuant un appel de méthode à distance vers le composant authentificateur. Les listes de contrôle d'accès sont initialisés lors de la création de cette extension en utilisant un « fichier de configuration » fourni par le programmeur. En pratique, ce fichier est directement intégré dans le script de déploiement, comme le montre la figure 5.4 page suivante ;
- une variante de l'extension précédente est utilisée pour le conteneur du composant authentificateur. Cette variante interroge directement le composant encapsulé

pour authentifier un nom et un mot de passe, au lieu de faire un appel de méthode à distance ;
- une extension de talon, placée au dessus de l'extension de cryptographie, surcharge la méthode `gtpSend` de façon à ajouter dans chaque message envoyé le nom et le mot de passe stockés dans l'extension de conteneur (sauf si le composant est en train d'exécuter une méthode non privilégiée, auquel cas on utilise le nom et le mot de passe du composant ayant appelé cette méthode non privilégiée) ;
- une extension de squelette, symétrique et placée elle aussi au-dessus de l'extension de cryptographie, surcharge la méthode `gtpRecv` de façon à extraire de chaque message reçu le nom et le mot de passe qu'il contient ;
- enfin, une autre extension de squelette surcharge la méthode `invoke` de façon à vérifier que l'appel de méthode est bien autorisé. Elle utilise les méthodes de l'extension de conteneur pour authentifier l'appelant, vérifie ses droits à l'aide des listes de contrôle d'accès elles aussi stockées dans l'extension de conteneur, et enfin effectue l'appel proprement dit, le cas échéant, en appelant `dsuper.invoke`.

```
...
// Seul les composants de John et ceux qui appartiennent au groupe BANK
// ont accès aux deux premières méthodes. Par contre, tous les composants
// appartiennent au groupe ALL et peuvent donc appeler la méthode deposit.
ACLS = "
grant Account {
  int getBalance ()          : John, [BANK];
  void withdraw (int amount) : John, [BANK];
  void deposit (int amount)  : [ALL];
}"
COMPONENT = {
  ...
  {javapodx.service.protection.acl.AclContainerx $ACLS}
}
...
```

FIG. 5.4 – Spécification des listes de contrôle d'accès d'un composant protégé

La figure 5.5 page ci-contre résume les méthodes principales des classes d'extensions utilisées pour implanter la protection.

5.3.3 Application à BAGHERA

Pour protéger les composants de l'application BAGHERA, il faut définir une politique de protection, puis faire en sorte qu'elle soit respectée en associant les extensions précédentes aux conteneurs, talons et squelettes appropriés. Mais ce n'est pas suffisant comme le verrons ci-dessous.

Politique de protection

Pour protéger l'accès aux composants `VirtualSchool`, `Mailbox`, `ElectronicCase`, et `ExerciseRepository`, nous utilisons les noms et les groupes suivants :

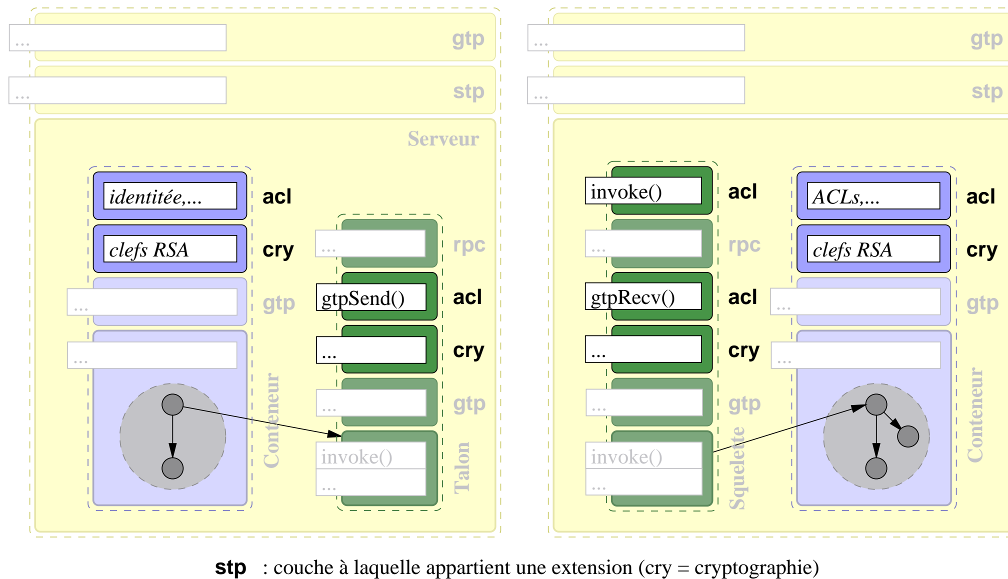


FIG. 5.5 – Résumé des extensions utilisées pour implanter la protection

- le nom et le mot de passe d'un composant `StudentAgent`, `ProfessorAgent` ou `AdminAgent` sont le nom et le mot de passe de la personne qui les utilisent. De plus, par définition, les composants `StudentAgent` appartiennent au groupe STUDENT, les composants `ProfessorAgent` au groupe PROFESSOR, et les composants `AdminAgent` au groupe ADMIN ;
- les noms et les mots de passe des autres composants de l'application BAGHERA sont arbitraires, mais tous ces composants appartiennent par définition au groupe APP (pour application) ;
- le groupe ALL contient tous les composants.

On peut alors définir la politique de protection comme le montre la figure 5.6 page 91, qui s'interprète de la façon suivante :

- seuls les composants du groupe APP peuvent ajouter ou enlever des utilisateurs et des groupes dans le composant authentificateur. Par contre, n'importe quel composant peut modifier son mot de passe, ou demander l'authentification d'un utilisateur (un objet `Identity` contient un nom et un mot de passe) ;
- la plupart des méthodes du composant `VirtualSchool` sont accessibles à tous, sauf celles qui modifient la liste des élèves et des professeurs, réservées aux administrateurs de l'application. Certaines méthodes sont privilégiées car elles ajoutent ou suppriment des utilisateurs dans le composant authentificateur, et ont donc besoin de s'exécuter avec les droits du groupe APP (cf. la section suivante) ;
- la plupart des méthodes d'un composant `ElectronicCase` ne sont accessibles que par le propriétaire de ce composant. Certaines le sont aussi par les professeurs, et deux seulement sont accessibles à tous ;
- les méthodes permettant de modifier la base d'exercices sont réservées aux profes-

seurs. Les autres sont accessibles à tous, sauf la méthode `getExerciseProof` (afin qu'un élève ne puisse pas récupérer la solution d'un exercice dans cette base).

Ajout de nouveaux utilisateurs

Même si on associait à chaque conteneur, talon et squelette les extensions permettant d'implanter la politique de protection précédente, l'application résultante ne fonctionnerait pas, à cause du problème suivant.

Lorsqu'un administrateur enregistre un nouvel élève, avec la méthode `addStudent`, il faut choisir un mot de passe initial pour cet élève, et il faut également l'enregistrer dans le composant authentificateur (sinon les messages envoyés par le composant `StudentAgent` de cet élève seraient systématiquement rejetés lors de l'étape d'authentification). Or le code fonctionnel de la méthode `addStudent` se contente de créer un composant `ElectronicCase` et d'enregistrer l'élève dans une table du composant `VirtualSchool`.

Nous avons résolu ce problème de la façon suivante. En temps normal, la méthode `addStudent` crée le cartable électronique de l'élève à l'aide d'une méthode appelée `createComponent`, fournie par une sous-classe de `Component`. Cette méthode appelle à son tour une méthode extensible `createComponent` du conteneur, qui créé effectivement le composant demandé[2]. Nous avons donc réalisé une extension, *spécifique à l'application* BAGHERA, utilisée uniquement pour le conteneur du composant `VirtualSchool`, et qui surcharge la méthode extensible `createComponent` de la façon suivante :

- examen des extensions du conteneur du composant à créer, afin de remplacer dans les listes de contrôle d'accès de ce composant le nom générique `owner` par le vrai nom de l'élève, propriétaire du futur composant ;
- tirage au sort d'un mot de passe, enregistrement du nom de l'élève et de ce mot de passe dans le composant authentificateur, et inclusion de ce nouvel utilisateur dans les groupes STUDENT et ALL ;
- création du cartable électronique, en appelant `dsuper.createComponent` ;
- affichage d'une boîte de dialogue pour informer l'administrateur du mot de passe initial choisi pour l'élève.

Le problème de l'enregistrement d'un nouveau professeur est résolue de la même manière. De même pour le problème de l'annulation de l'enregistrement d'un élève ou d'un professeur.

Protection de la méthode `addMail`

La méthode *addMail*, utilisable par tous les composants, permet d'envoyer un message à un élève ou à un professeur. Elle permet même, en principe, d'envoyer un message en se faisant passer pour un autre, puisque c'est l'émetteur du message qui crée l'objet

[2]les extensions associées au conteneur de ce nouveau composant ne sont pas spécifiées dans le code fonctionnel mais dans un « fichier de configuration » séparé, associé au composant créateur. C'est l'extension de conteneur du paquetage `config` qui permet d'associer ces extensions aux nouveaux composants automatiquement, de la même manière que pour les extensions des nouveaux talons et squelettes (cf. section 4.3.3 page 75).

```
grant Authenticator {
  void addUser (Identity user)                         : [APP];
  void removeUser (String user)                        : [APP];
  void setGroups (String user, String[] groups)        : [APP];
  void changePassword (Identity user, Identity newPasswd) : [ALL];
  String[] authenticate (Identity user)                : [ALL];
}

grant VirtualSchool {
  Vector getStudents ()                                : [ALL];
  void addStudent (String name, String professor)      : [ADMIN],<privileged>;
  void removeStudent (String name)                     : [ADMIN],<privileged>;
  String getProfessor (String student)                 : [ALL];
  ElectronicCase getElectronicCase (String student)    : [ALL];
  Vector getProfessors ()                              : [ALL];
  void addProfessor (String name)                      : [ADMIN],<privileged>;
  void removeProfessor (String name)                   : [ADMIN],<privileged>;
  Vector getStudents (String professor)                : [ALL];
  Mailbox getMailbox (String professor)                : [ALL];
  void transferStudent (String student, String professor) : [ADMIN];
  ExerciseRepository getExerciseRepository ()          : [ALL];
}

grant ElectronicCase {
  String getOwner ()                                   : [ALL];
  Vector getMails ()                                   : owner;
  void addMail (Mail mail)                             : [ALL];
  void removeMail (int index)                          : owner;
  void removeMails ()                                  : owner;
  void addListener (MailListener listener)             : owner;
  Vector getExerciseList ()                            : owner, [PROF];
  void addExercise (String name)                       : [PROF];
  void removeExercise (String name)                    : owner;
  Solution getSolution (String name)                   : owner, [PROF];
  void proposeSolution (String name, AnnotatedFigure s)  : owner;
  void confirmSolution (String name)                   : owner;
  void proposeCorrection (String name, AnnotatedFigure c) : [PROF];
  void confirmCorrection (String name)                 : [PROF];
}

grant ExerciseRepository {
  Vector getExerciseList ()                            : [ALL];
  AnnotatedFigure getExerciseProposition (String name)   : [ALL];
  AnnotatedFigure getExerciseProof (String name)       : [PROF];
  void addExercise (String name, Exercise exercise)    : [PROF];
  void removeExcercise (String name)                   : [PROF];
}
```

FIG. 5.6 – Extraits de la politique de protection de l'application BAGHERA

`Mail` correspondant, et qui peut donc initialiser le champ `sender` du message avec un faux nom.

Pour empêcher les utilisateurs de pouvoir faire cela, il faut rejeter tout appel à la méthode `addMail` si le champ `sender` du message *passé en argument* est différent de l'identité de l'appelant, qui peut être authentifiée. Malheureusement le modèle utilisé pour spécifier les politiques de protection, qui ignore totalement les arguments des méthodes appelées pour décider d'accepter ou non un appel, a une puissance insuffisante pour exprimer cette contrainte.

Pour résoudre le problème nous avons là encore réalisé une extension spécifique à l'application BAGHERA pour *compléter* le modèle de protection générique. Cette extension surcharge la méthode `invoke` des squelettes permettant d'accéder aux composants `Mailbox` et `ElectronicCase` de la façon suivante :

- si la méthode appelée n'est pas `addMail`, appeler `dsuper.invoke` et retourner le résultat ;
- sinon, comparer le champ `sender` du message passé en argument avec le nom du composant appelant, fourni par les extensions de protection génériques ;
- si ces noms sont différents, rejeter l'appel en lançant une exception. Sinon, l'accepter en appelant `dsuper.invoke`.

5.4 Mode déconnecté

5.4.1 Algorithme

Pour qu'un utilisateur puisse continuer à travailler en mode déconnecté avec une application distribuée, l'idée de base est la suivante :

- avant la déconnexion[3], on réalise une copie complète ou partielle des composants de l'application dont l'utilisateur aura besoin pour travailler en mode déconnecté ;
- après la reconnexion, on fusionne les copies précédentes avec les composants originaux, sachant que les deux peuvent avoir évolué indépendamment pendant le mode déconnecté.

Comme pour la persistance et la protection, la gestion du mode déconnecté utilise un algorithme générique, spécialisé pour chaque application par des données de configuration fournies par l'utilisateur. Par exemple, l'algorithme de protection est spécialisé en fonction de chaque application par la donnée d'une politique de protection. De même, l'algorithme de gestion du mode déconnecté est spécialisé par les données suivantes (spécifiées d'une manière qui sera indiquée dans la section suivante) :

- pour chaque composant situé sur un serveur qui peut être déconnecté des autres, l'utilisateur (plus précisément celui qui configure et qui déploie l'application) doit indiquer la liste des composants distants indispensables au fonctionnement de ce composant, et qui doivent donc être dupliqués localement lors de la déconnexion ;
- pour chaque composant susceptible d'être dupliqué/fusionné pour les besoins des

[3] supposée volontaire : nous ne traitons pas ici des déconnexions dues aux pannes transitoires de réseau.

composants précédents, l'utilisateur doit indiquer ce qui doit être recopié, et comment la fusion doit s'effectuer.

L'algorithme pour déconnecter un serveur consiste alors à passer en revue tous les composants du serveur et, pour chaque composant, à parcourir la liste de ses composants distants indispensables. Pour chaque composant de cette liste, on réalise une copie, puis on installe cette copie dans le serveur local (sauf si une telle copie existe déjà). A l'issue de ces opérations, on peut déconnecter physiquement le serveur du réseau. L'algorithme pour la reconnexion est similaire : pour chaque composant, et pour chacun de ses composants indispensables, on fusionne la copie locale installée lors de la déconnexion et la version originale restée sur le serveur distant.

Cet algorithme devrait normalement être appliqué récursivement sur les copies locales, au fur et à mesure qu'on les ramène. De même, il devrait être complété pour traiter le cas ou plusieurs composants ont besoin d'une copie partielle différente d'un même composant distant. Pour simplifier, nous n'avons pas pris en compte ces cas.

5.4.2 Mise en œuvre

Cette section présente l'implantation de l'algorithme précédent en montrant comment il fonctionne dans un cas concret. Nous considérons donc deux serveurs S_1 et S_2, et un composant client C, situé sur S_1, et relié par un connecteur client-serveur à un composant serveur S situé sur S_2. Nous supposons en outre que le client doit pouvoir être utilisé en mode déconnecté, et qu'il a besoin dans ce cas d'une copie partielle du composant serveur.

L'algorithme de gestion du mode déconnecté est mis en œuvre, dans cet exemple, à l'aide des extensions ci-dessous (cf. figure 5.7 page suivante), qui sont présentées plus en détails dans le reste de cette section :

- une extension E_1 ajoute au serveur S_1 les méthodes `disconnect` et `reconnect`, qui permettent de le déconnecter et de le reconnecter au serveur S_2 ;
- une extension E_c associée au conteneur du composant C stocke la liste des composants distants indispensables au fonctionnement de C en mode déconnecté (cette liste est ici réduite au composant S). Cette extension ajoute également les méthodes `disconnect` et `reconnect` au conteneur de C ;
- une extension E_s ajoute au conteneur du composant S les méthodes `clone` et `merge`, qui permettent de dupliquer ce conteneur, et de fusionner le conteneur original avec une copie ;
- une extension E_t associée au talon du connecteur client-serveur entre C et S permet de modifier le protocole d'envoi de messages `gtp` lorsque S_1 est déconnecté du réseau.

Processus de déconnexion

La déconnexion du serveur S_1 se déroule de la façon suivante (cf. figure 5.8 page 95) :

- l'utilisateur appelle directement ou indirectement la méthode `disconnect` du serveur S_1. Celle-ci appelle à son tour la méthode `disconnect` de chaque conteneur

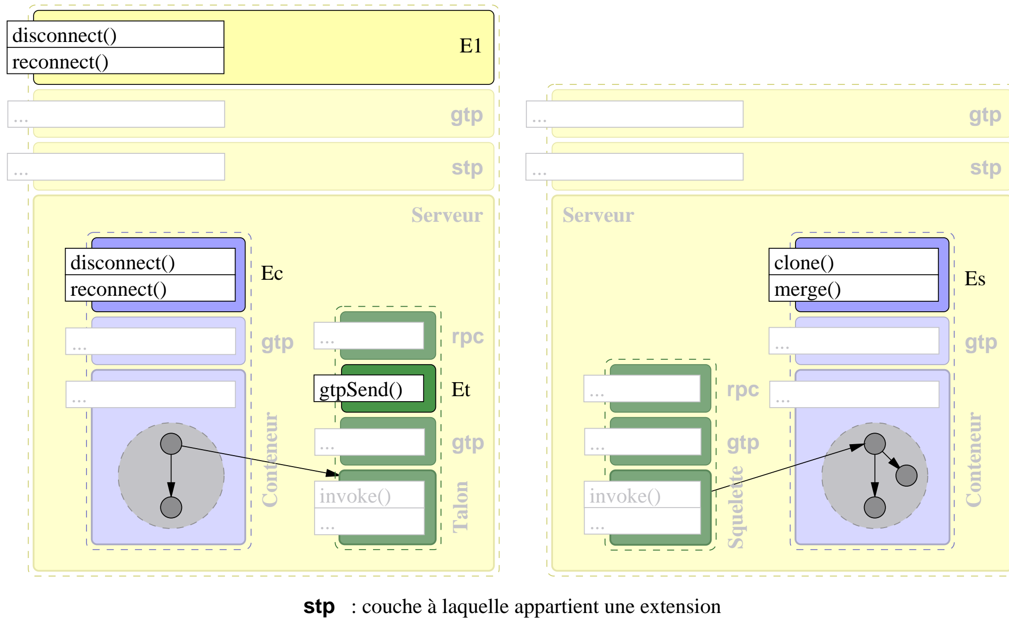


FIG. 5.7 – Extensions pour la gestion du mode déconnecté

du serveur dans lequel cette méthode est définie ;

- la méthode `disconnect` de l'extension E_c appelle la méthode `getDisconnection-Parameters` sur C. Cette méthode, fournie par le programmeur de C, retourne un objet censé spécifier les données du composant serveur qui sont nécessaires pendant le mode déconnecté. Par exemple, si le composant serveur est un agenda (resp. un annuaire), cet objet peut indiquer que seules les données concernant certaines semaines (resp. personnes) sont nécessaires. Dans tous les cas, c'est aux composants applicatifs, ici C et S, de fixer le type et la sémantique du résultat de cette méthode ;
- la méthode `disconnect` de l'extension E_c appelle ensuite la méthode `clone` de l'extension E_s, avec en argument les « paramètres de déconnexion », obtenus à l'étape précédente. Cet appel de méthode à distance est possible car la « liste des composants indispensables » contenue dans E_c (cf. ci-dessus) est en fait une liste de références de connecteurs client-serveur permettant d'accéder à distance aux méthodes `clone` et `merge` des conteneurs de ces composants ;
- la méthode `clone` de E_s réalise une copie *complète* du composant S et de son conteneur. Elle appelle ensuite, sur le clone de S ainsi obtenu, la méthode `clone-Component`, définie par le programmeur de S, et dont le rôle est de supprimer de ce clone les informations inutiles pour le client (spécifiées dans les « paramètres de déconnexion » transmis en argument). Le composant cloné ainsi modifié est ensuite retourné, avec son conteneur également cloné, en résultat de la méthode `clone`. Ces clones, désormais inutiles pour S_2, sont ensuite détruits ;
- la méthode `disconnect` de E_c poursuit ensuite son exécution : elle installe dans

le serveur S_1 le conteneur retourné par l'appel de méthode à distance précédent ;
- elle appelle ensuite la méthode `componentDisconnected` sur C. Cette méthode, fournie par le programmeur de C, permet au composant C d'effectuer des traitements supplémentaires spécifiques à l'application ;
- la méthode `disconnect` de E_c se termine, puis la méthode `disconnect` de E_1.

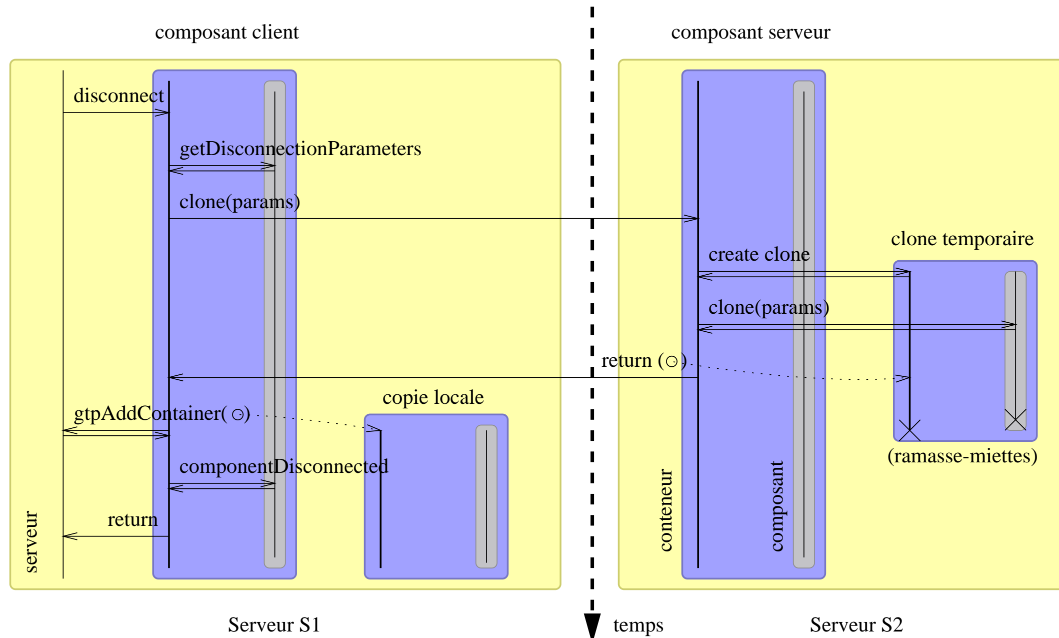


FIG. 5.8 – Processus de déconnexion

Fonctionnement de l'application en mode déconnecté

Le processus de déconnexion précédent permet d'installer une copie de S dans S_1, mais ce n'est pas suffisant pour faire en sorte que cette copie soit utilisée à la place de l'original en mode déconnecté. En effet, pour envoyer un message du client vers le serveur, les couches `gtp` et `stp` extraient l'adresse IP et le numéro de port TCP contenus dans l'adresse du squelette serveur, puis envoient le message sur une *socket* obtenue grâce à cette adresse IP et à ce numéro de port. Autrement dit, que l'on soit en mode connecté ou non, les messages du client sont toujours envoyés vers le composant original, ce qui provoque une erreur en mode déconnecté.

Pour remédier à ce problème, il faut modifier le fonctionnement de ces protocoles en mode déconnecté, de façon à ce que les messages soient envoyés vers le serveur local, ici S_1, censé contenir une copie du composant destinataire. L'extension E_t, placé sur le talon du connecteur client-serveur entre C et S, permet justement de faire cela. Elle surcharge en effet la méthode `gtpSend` (cf. section 4.3.2 page 72) de la façon suivante :
- si le serveur est en mode connecté, elle appelle `dsuper.gtpSend` afin d'envoyer le message selon la méthode habituelle ;

– si le serveur est en mode déconnecté, elle utilise la couche `stp` directement, sans passer donc par la couche `gtp` inférieure, afin d'envoyer le message vers le serveur local.

Cela revient à utiliser non plus une pile mais un graphe de protocoles, et à choisir des chemins différents dans ce graphe en fonction du contexte. Dans ce cas relativement simple, notre modèle de composition d'objets, bien que linéaire, permet quand même de simuler ce graphe de protocoles. Une autre solution, qui permet d'éviter de simuler un graphe de protocoles, consiste à modifier la pile de protocoles dynamiquement, ce qui est possible avec notre modèle de composition.

Processus de reconnexion

Le processus de reconnexion est similaire au processus de déconnexion :

– l'utilisateur appelle la méthode `reconnect` de S_1, qui appelle à son tour la méthode `reconnect` de chaque conteneur du serveur dans lequel cette méthode est définie (cette liste est fournie par l'extension de serveur de la couche `gtp`, à partir de sa table de hachage interne - cf. section 4.3.2 page 72) ;
– la méthode `reconnect` de l'extension E_c appelle la méthode `getDisconnection-Parameters` sur C ;
– elle appelle ensuite, à distance, la méthode `merge` de l'extension E_s, avec en paramètre la copie locale de S et de son conteneur, et le résultat de la méthode `getDisconnectionParameters` ;
– la méthode `merge` de E_s appelle la méthode `mergeComponent` sur le composant original S, avec en paramètre la copie locale et les « paramètres de déconnexion ». La méthode `mergeComponent`, fournie par le programmeur de S, réalise alors la fusion de l'original et de la copie. La copie est ensuite détruite ;
– la méthode `reconnect` de E_c poursuit ensuite son exécution : elle désinstalle la copie locale de S du serveur S_1, puis détruit cette copie ;
– elle appelle ensuite la méthode `componentReconnected` sur C, pour qu'il puisse effectuer d'éventuels traitements spécifiques supplémentaires ;
– la méthode `reconnect` de E_c se termine, puis la méthode `reconnect` de E_1.

Remarques

Ainsi que la section 5.4.1 page 92 l'annonçait, la gestion du mode déconnecté utilise un algorithme générique adapté à chaque application par l'utilisateur. Malheureusement, cette adaptation ne se fait pas entièrement, comme pour la persistance ou la protection, sous forme déclarative, de façon séparée du code fonctionnel de l'application. En effet, bien que la « liste des composants indispensables » à un composant utilisable en mode déconnecté puisse être facilement spécifiée séparément du code fonctionnel, ce n'est le cas de la sémantique des méthodes `clone` et `merge` : cette sémantique étant très variable et très dépendante de l'application considérée, elle est difficile à définir de façon déclarative, séparément du code fonctionnel. Nous avons donc choisi une solution plus simple, mais qui conduit à un mélange de code fonctionnel et non-fonctionnel. Cette

solution consiste en effet à demander au programmeur de définir cette sémantique de façon procédurale, dans les méthodes `cloneComponent` et `mergeComponent`.

Le même problème se pose du coté client, pour spécifier les traitements à effectuer juste après la déconnexion ou la reconnexion. Nous l'avons résolu de la même manière, à savoir en demandant au programmeur de spécifier ces opérations de façon procédurale, dans les méthodes `componentDisconnected` et `componentReconnected`.

5.4.3 Application à BAGHERA

Dans le cas de l'application BAGHERA, on souhaite que les composants `StudentAgent` soient utilisables en mode déconnecté, où l'on suppose que les composants `VirtualSchool`, `Mailbox`, `ElectronicCase`, et `ExerciseRepository` ne sont pas accessibles.

Pour cela, nous associons l'extension E_1 au serveur utilisé par l'élève, l'extension E_c au conteneur du composant `StudentAgent` (qui joue le rôle de C dans l'exemple précédent), et l'extension E_s au conteneur du composant `VirtualSchool` (qui joue le rôle de S). Nous associons également l'extension E_t au talon du connecteur client-serveur entre les composants `StudentAgent` et `VirtualSchool`. Il ne reste plus ensuite qu'à ajouter les méthodes `cloneComponent` et `mergeComponent` dans le composant `VirtualSchool`, ainsi que les méthodes `componentDisconnected` et `componentReconnected` dans le composant `StudentAgent`.

Les méthodes `cloneComponent` et `mergeComponent`

Pour pouvoir travailler en mode déconnecté, un élève a besoin d'une copie complète du composant représentant son cartable électronique. Il a également besoin d'une *partie* de la base d'exercices : il lui faut en effet les énoncés des exercices figurant dans son cartable. Par contre, il n'a pas besoin des énoncés des autres exercices de la base, et les solutions d'exercices contenues dans cette base ne doivent pas lui être communiquées. Finalement, il n'a pas besoin d'une copie des cartables des autres élèves, ni des boîtes aux lettres des professeurs.

La méthode `cloneComponent` ne supprime aucune donnée du composant `VirtualSchool`. Au contraire, elle lui ajoute des données en provenance des autres composants : le contenu du cartable électronique de l'élève, et les énoncés d'exercices appropriés. Autrement dit, la duplication de ce composant s'accompagne d'un changement de topologie (cf. figure 5.9 page suivante) : alors que le composant original utilise des composants auxiliaires pour stocker les cartables, les boîtes aux lettres et la base d'exercices, le composant dupliqué incorpore directement une partie de ces données, et est ainsi autosuffisant.

La méthode `mergeComponent` laisse le composant `VirtualSchool` original intact. Par contre, elle réalise le changement de topologie inverse de celui effectué lors de la déconnexion. Ainsi, elle fusionne le cartable électronique original de l'élève avec la copie incorporée dans la copie du composant `VirtualSchool` : les solutions proposées par l'élève pendant le mode déconnecté sont ajoutés dans le composant original. La fusion

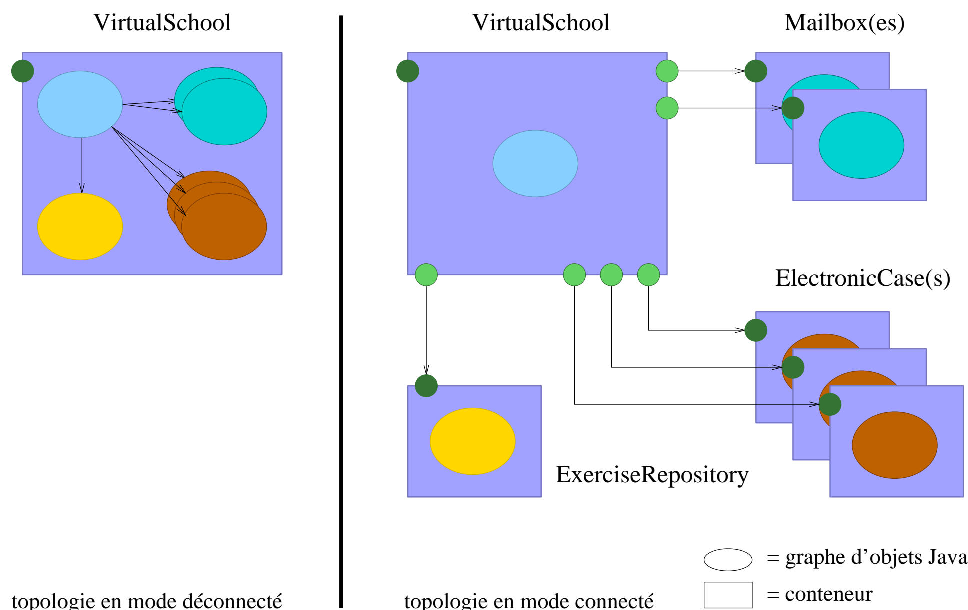


FIG. 5.9 – Topologie de l'application BAGHERA en mode déconnecté

de la base d'exercice est simple : l'original est laissé intact. Enfin, les messages envoyés par l'élève aux autres utilisateurs pendant le mode déconnecté, stockés temporairement dans la copie du composant `VirtualSchool`, sont ajoutés dans les cartables et boîtes aux lettres originales appropriées.

Le « paramètre de déconnexion » utilisé par les deux méthodes précédentes, et retourné par la méthode `getDisconnectionParameters`, est tout simplement le nom de l'élève qui se déconnecte ou qui se reconnecte : ce nom permet en effet de savoir quel cartable doit être dupliqué ou fusionné, et le contenu de ce cartable permet à son tour de savoir quels énoncés d'exercices il faut dupliquer.

Les méthodes `componentDisconnected` et `componentReconnected`

Un composant de type `StudentAgent` possède initialement un connecteur client-serveur « primaire » vers le composant `VirtualSchool`. Mais, au fur et à mesure de son exécution, il peut acquérir, grâce à certaines méthodes du composant `VirtualSchool`, des connecteurs client-serveurs « secondaires » vers des composants de `ElectronicCase`, `Mailbox` ou `ExerciseRepository`.

Juste après une déconnexion, et à cause du changement de topologie effectué à cette occasion, ces connecteurs secondaires éventuels ne sont plus utilisables. La méthode `componentDisconnected` les remplace donc par de nouveaux, en utilisant la copie locale du composant `VirtualSchool` qui vient juste d'être installée (cf. figure 5.10 page suivante). De même, juste après une reconnexion, ces connecteurs secondaires, recalculés lors de la déconnexion, ne sont plus valides. La méthode `componentReconnected`

les remplace donc, en utilisant le composant `VirtualSchool` original.

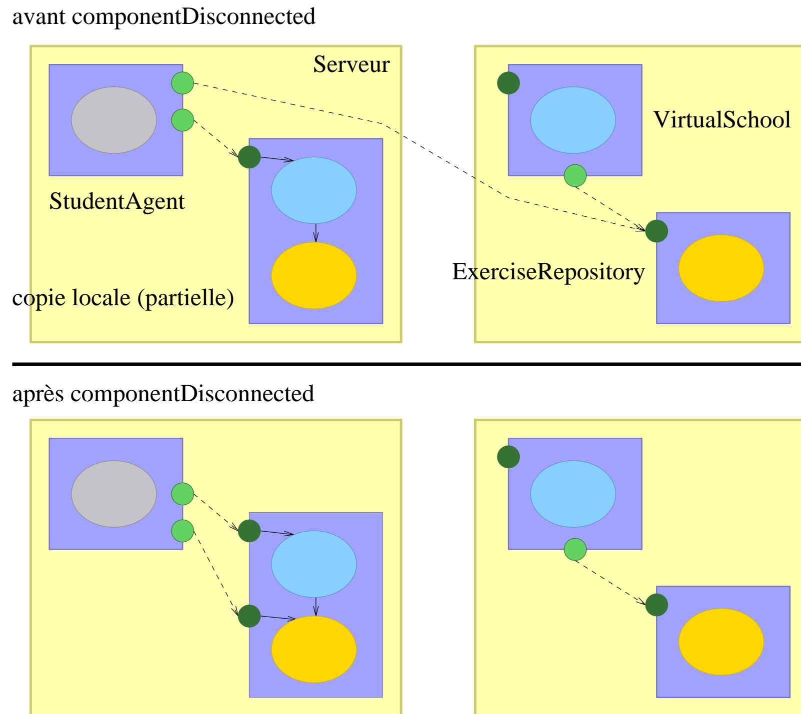


FIG. 5.10 – Effets de la méthode `componentDisconnected`

Chapitre 6

Résultats

Afin d'évaluer dans quelle mesure l'architecture que nous proposons dans cette thèse permet d'atteindre nos objectifs, il était nécessaire, comme nous l'avons déjà expliqué, d'essayer de l'utiliser dans un cas concret (au moins). C'est pourquoi nous avons mis en œuvre l'application BAGHERA, ainsi que trois propriétés non-fonctionnelles associées, avec la plate-forme JavaPod. Après avoir présenté, dans le chapitre précédent, de quelle manière nous avons implanté ces propriétés, nous évaluons dans ce chapitre dans quelle mesure cette mise en œuvre atteint nos objectifs.

6.1 Résultats qualitatifs

Cette section présente les résultats qualitatifs que nous avons obtenus. Les résultats quantitatifs sont présentés dans la section suivante.

6.1.1 Séparation du code fonctionnel et non-fonctionnel

Cas de la persistance et de la protection

L'objectif de séparation du code fonctionnel et non-fonctionnel est atteint en ce qui concerne la persistance et la protection. En effet, le code non-fonctionnel lié à ces propriétés est, comme nous l'avons vu dans le chapitre précédent, entièrement regroupé dans des extensions de la plate-forme. De plus, ces extensions, qui sont génériques (c'est à dire réutilisables avec d'autres applications), sont adaptées à une application particulière grâce à des données de configuration (comme par exemple des listes de contrôle d'accès) qui sont elles-mêmes séparées du code fonctionnel.

On constate cependant que cette séparation complète n'est que *syntaxique*. Il reste en effet des liens *sémantiques* entre le code fonctionnel et le code non-fonctionnel, comme le montrent les exemples suivants :

- la granularité des composants découle, dans une certaine mesure, des propriétés non-fonctionnelles que l'on souhaite leur associer. Par exemple, du point de vue fonctionnel, il est équivalent d'utiliser des composants de type `VirtualSchool`,

`Mailbox`, `ElectronicCase`, et `ExerciseRepository` séparés, ou bien de les regrouper tous en un seul gros composant. Cela ne change rien non plus pour la protection. Par contre, pour la persistance, ce choix influe sur les performances : si on utilise un seul gros composant, toute modification, même mineure, provoque l'enregistrement sur disque de ce gros composant. A l'inverse, avec plusieurs petits composants, une telle modification provoque l'enregistrement sur disque d'un seul petit composant ;
- le choix des interfaces applicatives découle lui aussi, dans une certaine mesure, des propriétés non-fonctionnelles. Par exemple, selon la politique de protection de l'application BAGHERA, seuls les professeurs peuvent accéder aux solutions contenues dans la base d'exercices, alors que n'importe qui peut accéder aux énoncés de ces exercices. Etant donné l'algorithme de protection utilisé, cette politique n'est applicable que si on utilise deux méthodes pour consulter un exercice de la base : l'une pour consulter l'énoncé, l'autre pour consulter la solution. Si ces deux méthodes avaient été réunies en une seule, ce qui n'aurait pas changé pas grand chose du point de vue fonctionnel, nous n'aurions pas pu associer cette politique de protection à l'application BAGHERA[1].

Ces liens sémantiques implicites risquent de rendre difficile, dans le cas général, l'association d'une propriété non-fonctionnelle à une application qui n'aurait pas été réalisée en ayant à l'esprit qu'on pourrait éventuellement lui associer cette propriété dans le futur. Par exemple, si nous avions réalisé l'application BAGHERA sans avoir à l'esprit le fait qu'on lui associerait plus tard des propriétés de persistance et de protection, nous aurions très bien pu utiliser une granularité différente pour les composants et leurs interfaces, qui aurait rendu difficile, voire impossible, l'association ultérieure de ces propriétés à l'application.

Cas de la gestion du mode déconnecté

Le code non-fonctionnel pour la gestion du mode déconnecté est lui aussi regroupé dans des extensions de la plate-forme JavaPod. Par contre, contrairement au cas de la persistance et de la protection, la configuration de ces extensions génériques pour une application particulière ne se fait pas de façon déclarative et séparément du code fonctionnel, mais au contraire de façon procédurale, dans le code fonctionnel. Même si ce code non-fonctionnel spécifique à l'application est regroupé dans des méthodes bien définies comme `cloneComponent` ou `mergeComponent`, et n'est donc pas totalement mélangé au code fonctionnel, il n'en reste pas moins que la séparation n'est pas aussi complète que pour la persistance et la protection.

Comme nous l'avons dit dans le chapitre précédent, il est difficile de spécifier de façon déclarative, séparément du code fonctionnel, la sémantique des opérations `clone` et `merge`. Devant cette difficulté, nous avons choisi en quelque sorte la solution de facilité, sans étudier les solutions proposées dans la littérature pour essayer de résoudre ce problème. Autrement dit, le fait que n'ayons pas pu obtenir une séparation complète ne

[1] sauf en utilisant une extension spécifique à l'application, comme dans le cas de la méthode `addMail` (cf. section 5.3.3 page 90)

prouve pas que l'architecture que nous proposons rend une telle séparation impossible. Pour en décider, il faudrait étudier le problème de la gestion du mode déconnecté en profondeur, ce qui nécessiterait un travail de thèse à part entière.

6.1.2 Composition de propriétés non-fonctionnelles

Le chapitre précédent montre qu'il est possible d'associer *individuellement* les propriétés de persistance, de protection et de mode déconnecté à l'application BAGHERA. Cette section évalue dans quelle mesure on peut *composer* ces propriétés.

Persistance et protection

Il est très facile d'associer à l'application BAGHERA les propriétés de persistance et de protection simultanément. Il suffit en effet de composer les extensions relatives à ces deux propriétés, pour chaque serveur, conteneur, talon ou squelette. Ce qui signifie qu'il suffit d'associer à chaque serveur, conteneur, talon ou squelette x les extensions $E_1, ..., E_n, E'_1, ..., E'_m$, où $E_1, ..., E_n$ sont les extensions associées à x pour la persistance seule, et où $E'_1, ..., E'_m$ sont les extensions associés à x pour la protection seule (ici l'ordre n'est pas important, mais il l'est dans le cas général. Par exemple, quand on compose une extension de surveillance avec une extension de protection sur un squelette, l'extension de surveillance intercepte soit tous les appels à distance, soit seulement ceux qui sont autorisés, selon l'ordre utilisé).

En fait c'est un tout petit peu plus compliqué que cela, car il faut également rendre persistant le composant authentificateur utilisé pour la protection. Heureusement, cette opération est très simple également, puisque ce composant, bien qu'il soit plus de niveau système que de niveau applicatif, est mis en œuvre de la même façon que les composants applicatifs. On peut donc facilement lui associer les mêmes extensions que l'on utilise pour rendre les composants applicatifs persistants.

Persistance et mode déconnecté

Il est également très simple d'associer à l'application BAGHERA les propriétés de persistance et de gestion du mode déconnecté. Il suffit en effet, là encore, de composer les extensions relatives à ces deux propriétés pour chaque serveur, conteneur, talon ou squelette. Il faut également associer au serveur qui se déconnecte l'extension de serveur pour la persistance, de façon à ce que les copies locales de composants persistants puisse être enregistrées, en mode déconnecté, sur un support de persistance local.

Protection et mode déconnecté

L'algorithme utilisé pour la persistance ne dépend pas du fait qu'un composant soit protégé ou non : dans les deux cas, il suffit d'enregistrer le composant et son conteneur sur disque après chaque modification. De même, l'algorithme utilisé pour la protection ne dépend pas du fait qu'un composant protégé soit persistant ou non. On résume cela en disant que la persistance et la protection sont des propriétés *indépendantes*,

ou encore *orthogonales*. C'est cette indépendance entre persistance et protection, mais aussi entre persistance et mode déconnecté, qui permet de se contenter de composer les extensions relatives à deux propriétés pour obtenir les deux simultanément.

Malheureusement, la protection et la gestion du mode déconnecté ne sont pas des propriétés indépendantes. En effet, la protection d'un composant vis à vis d'un client n'est possible que si le client n'est pas sur la même machine que le composant ou, si ce n'est pas le cas, si la machine (virtuelle et physique) en question est sûre. Or, en mode déconnecté, la copie locale du composant est par définition sur la même machine que le client, et cette machine n'est pas sûre a priori. L'algorithme de protection est donc dépendant du fait que le composant protégé soit ou ne soit pas recopié localement en mode déconnecté, d'une façon extrême : il est correct dans le deuxième cas, mais inopérant dans le premier.

A priori, pour pouvoir composer deux propriétés non-fonctionnelles dépendantes, il faut écrire du code, spécifique à ces deux propriétés, pour « traiter » leurs interactions néfastes. En l'occurrence, pour la protection et le mode déconnecté, l'interaction néfaste est la suivante : la méthode `clone`, utilisée pour dupliquer un composant et son conteneur lors du processus de déconnexion, réalise une copie *complète* du conteneur[2], et une copie complète ou partielle du composant. Dans le cas d'un composant protégé, le conteneur contient des extensions pour la protection, qui non seulement sont inopérantes en mode déconnecté, mais qui en plus sont nuisibles : elles contiennent en effet des informations secrètes (comme des clefs RSA), qui ne le resteraient pas longtemps si elles étaient conservées dans la copie du conteneur.

Pour résoudre ce problème, nous avons réalisé une extension afin de surcharger la méthode `clone` des conteneurs des composants protégés. Cette extension permet de supprimer du conteneur dupliqué, obtenu par un appel à `dsuper.clone`, les extensions relatives à la protection. Mais cette extension introduit un nouveau problème, du coté des clients de ces composants protégés, qui contiennent eux aussi des extensions relatives à la protection (notamment pour inclure l'identité de l'appelant dans chaque message d'invocation). En effet, si on ne supprimait pas ces extensions « clientes » lors de la déconnexion, les clients ne pourraient plus communiquer avec les copies locales de ces composants, non protégées (les piles de protocoles seraient différentes coté client et coté serveur). Pour résoudre ce problème induit, nous avons réalisé une autre extension, placée sur les talons clients, qui permet de changer la pile de protocoles en fonction du contexte : on utilise une pile normale en mode connecté, et une pile simplifiée, sans les protocoles liés à la protection, en mode déconnecté.

Finalement, un autre problème à résoudre pour composer la protection et le mode déconnecté consiste à protéger les accès aux méthodes « système » `clone` et `merge`. En effet, la méthode `clone` permet a priori à un élève d'obtenir la copie complète du cartable d'un autre élève (cf. section 5.4.3 page 97), ce qui est contraire à la politique de protection de l'application BAGHERA. Pour éviter cela, il faut interdire l'accès à cette méthode si son argument n'est pas égal au nom de l'élève appelant. Grâce au fait que les méthodes `clone` et `merge` soient appelées à distance en utilisant un connecteur

[2]on désigne par là, dans cette section, le conteneur lui-même ainsi que ses talons et squelettes.

client-serveur identique à ceux utilisés entre les composants applicatifs, ce problème se résout facilement, en utilisant une extension spécifique à l'application pour compléter les extensions standard pour la protection (comme pour la méthode applicative `addMail` - cf. section 5.3.3 page 90).

Synthèse

En résumé, nous pouvons dire que le code de l'application BAGHERA et de ses propriétés non-fonctionnelles permet bien d'associer à l'application plusieurs propriétés non-fonctionnelles simultanément. De plus, les propriétés indépendantes peuvent être composées facilement, puisqu'il suffit de composer les extensions qui les mettent en œuvre (en faisant toutefois attention à l'ordre). Par contre, pour composer des propriétés non indépendantes, il faut écrire du code spécifique à ces propriétés pour qu'elles puissent fonctionner ensemble correctement (mais il n'y a pas besoin de modifier de code existant).

Les propriétés non indépendantes réintroduisent donc une certaine combinatoire, que l'on avait cherché à éviter à tout prix lors de la définition de notre mécanisme de composition (cf. section 3.3.1 page 45). On pourrait alors penser que ces efforts étaient inutiles, mais ce serait oublier qu'un certain nombre de propriétés sont indépendantes, et que dans ce cas un mécanisme de composition qui évite d'introduire une combinatoire artificielle est bien utile.

6.1.3 Modularité et extensibilité

Toutes les extensions de la plate-forme JavaPod, qu'elles aient été réalisées dans le cadre des expérimentations avec l'application BAGHERA ou en dehors, sont implantées de façon modulaire : ces extensions sont en effet regroupées dans des paquetages séparés (cf. section 4.3.1 page 70), et si un paquetage en utilise d'autres, il le fait au travers d'interfaces bien définies, ce qui permet de remplacer un paquetage par un autre, pourvu qu'il ait la même interface.

L'objectif d'extensibilité est également atteint, dans certaines limites. Tout d'abord, on peut considérer que le noyau de la plate-forme JavaPod est complètement extensible, puisque nous avons pu implanter de nombreux protocoles, propriétés non-fonctionnelles et services d'administration à l'aide d'extensions de ce noyau, sans avoir eu besoin de le modifier.

Les extensions du noyau peuvent également être adaptées à une application particulière, grâce à des extensions supplémentaires spécifiques à cette application. Par exemple, la méthode `addMail` est protégée grâce à une extension de ce type. De même, l'enregistrement automatique des nouveaux utilisateurs dans le composant authentificateur est réalisé par une extension spécifique (cf. section 5.3.3 page 90).

Nous avons également pu étendre la plate-forme JavaPod pour mettre en œuvre des propriétés non-fonctionnelles non prévues à l'avance. Par exemple, pour améliorer les performances de l'application BAGHERA, nous avons réalisé des extensions pour modifier les connecteurs client-serveur par défaut, de façon à stocker chez le client, dans un cache,

les résultats des appels précédemment effectués.

Malgré tout, la plate-forme JavaPod n'est extensible que dans certaines limites. Par exemple, lorsque nous avons réalisé les extensions pour la persistance, il a fallu modifier légèrement l'implantation de la couche `gtp`. Cette couche utilisait en effet des adresses qui n'étaient valables que pendant la durée de vie d'un serveur, ce qui était problématique pour désigner les portes des composants persistants, dont la durée de vie est supérieure. Nous avons donc du modifier le format de ces adresses, pour qu'elles soient valables de façon permanente.

La granularité des interfaces des extensions est un autre facteur qui peut limiter l'extensibilité. Par exemple, il n'a pas été possible de modifier le format des adresses de la couche `gtp` à l'aide d'une extension, car la granularité des extensions de cette couche n'était pas assez fine. Il a donc fallu modifier le code des extensions déjà existantes. Le problème est que la granularité des extensions influe non seulement sur l'extensibilité, mais aussi sur les performances. Il faut donc trouver un compromis entre extensibilité et performances.

6.1.4 Évaluation du mécanisme de composition

L'expérience acquise lors de la mise en œuvre de la plate-forme JavaPod et de l'application BAGHERA permet d'évaluer le mécanisme de composition que nous proposons.

Nous avons ainsi constaté que ce mécanisme est utile non seulement dans les talons et les squelettes (ce qui est logique puisque c'est la volonté de pouvoir construire des talons « adaptables » qui est à l'origine de la définition de ce mécanisme - cf. section 3.3.1 page 45), mais aussi dans les conteneurs et les serveurs. Par exemple, dans les serveurs, il est utilisé pour surcharger la méthode `gtpGetComponent`, afin de charger automatiquement les composants persistants en mémoire (cf. section 5.2.2 page 84). De même, dans les conteneurs, il est utilisé pour surcharger des méthodes telles que `createSkeleton` ou `createStub`, afin de configurer automatiquement les portes initiales des composants (cf. section 4.3.3 page 75).

Nous pensons également que la « rigidité » de notre mécanisme de composition est plus un avantage qu'un inconvénient. En effet, à première vue, le fait que la sémantique d'un objet composé ne dépende que de la sémantique de ses membres et de l'ordre dans lequel ils sont composés peut paraître comme un manque de souplesse, et donc comme un inconvénient. On serait même tenté de proposer, pour remédier à ce manque, de laisser au programmeur le soin de définir la sémantique de ses objets composés, à l'aide d'objets « contrôleurs » ou « composeurs » (comme dans Guarana - cf. section 2.1.3 page 16). Comme le montre la figure 6.1 page ci-contre, cela permettrait de définir des stratégies de composition totalement arbitraires, qu'il serait très difficile de mettre en œuvre avec notre mécanisme de composition. Mais, pour éviter de réintroduire des problèmes d'explosion combinatoire, il faudrait utiliser un objet « composeur » générique, paramétrable à l'aide « descripteurs de composition ». En fait, pour conserver des performances acceptables, il faudrait plutôt compiler ces descripteurs, afin de produire, à la volée, des objets « composeurs » spécialisés. Quoi qu'il en soit, ces descripteurs compliqueraient la spécification des scripts de déploiement (cf. section 4.4.2 page 78),

le format des références de connecteurs...En résumé, si on tient compte de toutes les conséquences, la rigidité de notre modèle de composition est bien un avantage et non pas un inconvénient.

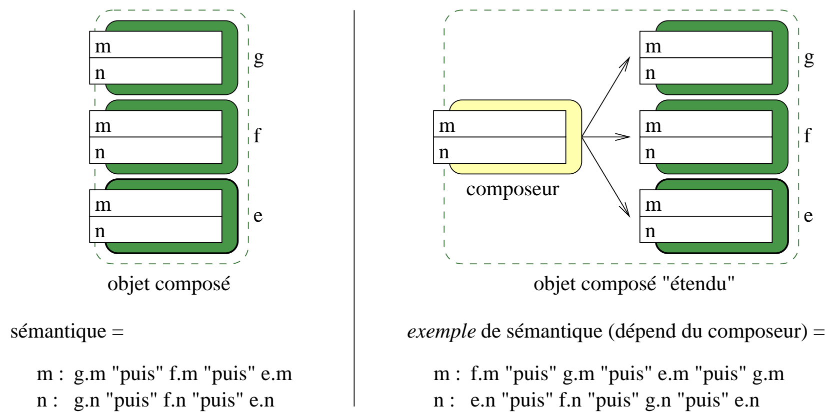


FIG. 6.1 – Une extension possible de notre mécanisme de composition

6.2 Résultats quantitatifs

Nos objectifs étant de nature qualitative, nous pourrions nous contenter dans ce chapitre de l'évaluation qualitative de la section précédente. Nous donnons malgré tout, dans cette section, quelques résultats qualitatifs, afin de donner une idée du coût des mécanismes que nous proposons.

6.2.1 Performances de ejava

Afin d'évaluer le surcoût induit par l'utilisation de notre mécanisme de composition, nous avons mesuré le temps nécessaire pour effectuer un appel de méthode extensible sur un objet composé en ejava, avec une méthode vide, sans arguments ni résultat. Nous avons ensuite comparé les résultats obtenus avec différents types d'appels de méthode en Java, toujours avec une méthode vide.

Pour mesurer le coût d'un appel, nous mesurons en réalité le temps qu'il faut pour réaliser une boucle de n itérations, chaque itération consistant à effectuer un appel. Nous soustrayons ensuite le temps qu'il faut pour réaliser n itérations vides, puis nous divisons par n. En pratique, n est supérieur ou égal à un million. Nous avons ainsi mesuré les coûts suivants :

- `java1` et `java2` : coût d'un appel de méthode normale et d'interface (correspondant respectivement aux instructions de *bytecode* Java `opc_invokevirtual` et `opc_invokeinterface`) ;
- `ejava1` et `ejava2` : coût d'un appel de méthode extensible sur un objet extensible sans extension, et avec une extension contenant la même méthode, réduite à un appel à `dsuper`.

Nous avons fait chaque mesure sur quatre machines virtuelles différentes : le JDK 1.2.2 sur Linux, avec un JIT (*just in time compiler*) de Borland, le JDK 1.3 sur Linux, avec Hotspot ClientVM, le JDK 1.2.2 sur NT, avec le JIT par défaut, de Symantec, et enfin le JDK 1.3 sur NT, avec Hotspot ClientVM. La machine physique était la même dans tous les cas, à savoir un Pentium III cadencé à 730Mhz. Les résultats que nous avons obtenus sont présentés dans la figure 6.2.

Test	Linux		NT	
	JDK1.2.2	JDK1.3	JDK1.2.2	JDK1.3
java1	0.29	0.025	0.007	0.022
java2	0.30	0.027	0.035	0.022
ejava1	1.07	0.335	0.313	0.314
ejava2	1.17	0.361	0.376	0.344

FIG. 6.2 – Performances des appels de méthodes extensibles (temps en μs)

On constate qu'un appel de méthode extensible en ejava est, selon la machine virtuelle considérée, de 3,5 à 14 fois plus lent qu'un appel de méthode d'interface en Java, qui lui même est plus lent que les autres types d'appels Java. L'écart est malheureusement d'autant plus grand que la machine virtuelle est plus efficace. On constate également que le coût d'un appel à `dsuper`, égal à `ejava2`−`ejava1`, est lui du même ordre de grandeur que le coût d'un appel de méthode d'interface.

6.2.2 Performances de l'appel de méthode à distance

Un facteur de 3 à 14 entre un appel de méthode extensible et un appel normal peut paraître assez important. En pratique, du fait que les méthodes utilisées ne sont pas vides, ce facteur est fortement réduit. Pour le montrer, nous avons mesuré les performances de l'appel de méthode à distance avec Java RMI et avec la plate-forme JavaPod.

Pour mesurer l'influence de la taille des arguments, nous avons utilisé une méthode distante qui prend en argument un tableau d'octets, et qui retourne ce tableau tel quel. La figure 6.3 page suivante montre les résultats obtenus en utilisant deux serveurs situés sur la même machine physique (toujours un Pentium III à 730Mhz), et avec le JDK 1.3 pour NT[3], alors que la figure 6.4 page 110 montre les résultats obtenus avec deux machines physiques différentes (deux Pentium III à 730 et 600Mhz, avec NT et le JDK 1.3). On constate qu'un appel de méthode à distance avec la plate-forme JavaPod n'est plus que 1,6 fois plus lent au pire qu'avec Java RMI, alors que, dans les mêmes conditions, le facteur entre un appel de méthode extensible et un appel de méthode d'interface est d'environ 14. De plus, lorsqu'on mesure la part due aux traitements coté

[3] nous avons également utilisé des classes `ObjectInputStream` et `ObjectOutputStream` optimisées en ce qui concerne la gestion des descripteurs de classe. Sans cette optimisation, la taille des en-têtes des messages d'invocation est beaucoup plus grande, et les performances beaucoup moins bonnes. Malheureusement ces classes optimisées ne sont pas portables.

client (4%), aux traitements coté serveur (13%), et aux traitements dus à la sérialisation et au réseau (83%), on s'aperçoit que ce facteur résiduel de 1,6 est essentiellement du au fait que notre algorithme d'appel de méthode à distance n'est pas assez optimisé. Autrement dit, le surcoût du à ejava seul est négligeable dans un appel de méthode à distance.

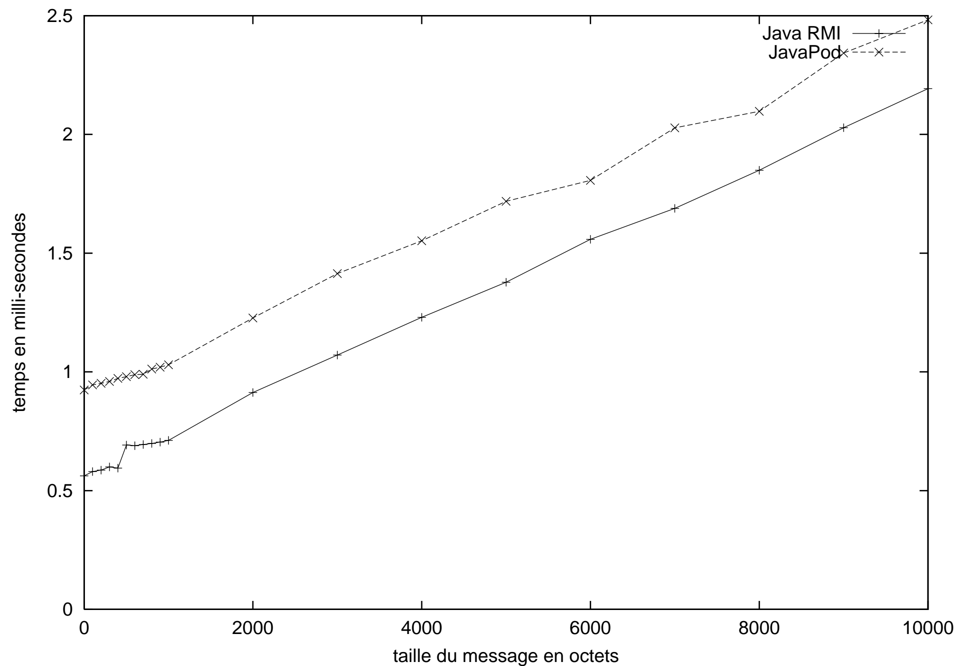


FIG. 6.3 – Performances de l'appel de méthode à distance en local

Nous avons également mesuré la taille de la référence du connecteur client-serveur utilisé pour les mesures précédentes, afin de quantifier la remarque faite dans la section 3.1.3 page 40 à propos de la différence de taille entre une référence de connecteur et une référence d'interface. Nous avons trouvé une taille de 417 octets, ce qui est effectivement bien plus que la taille d'une référence d'interface (qui contient, typiquement, une adresse IP, un numéro de port TCP et une clé de hachage, soit 10 à 15 octets environ). Cette taille importante s'explique en partie par le fait que cette référence contient plusieurs noms de classe d'extensions, qui sont assez longs (comme par exemple `javapodx.protocol.gtp.basic.BasicGTPGatex`).

6.2.3 Performances de la persistance

Pour mesurer les performances de la persistance nous avons mesuré les performances de l'appel de méthode à distance sur un composant persistant. Le composant utilisé était vide, c'est à dire constitué d'un seul objet sans aucun champ. La méthode utilisée était elle-même vide, sans arguments ni résultat. En utilisant deux serveurs situés sur

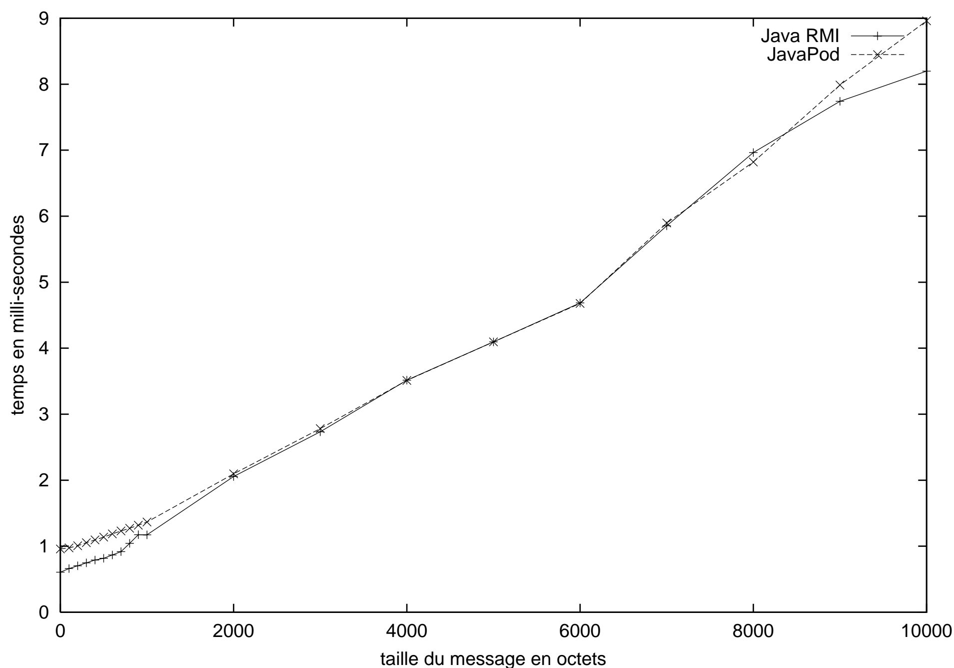


FIG. 6.4 – Performances de l'appel de méthode à distance en réparti

une même machine physique (toujours le Pentium III à 730Mhz avec NT et le JDK 1.3), nous avons obtenus les résultats suivants :

- lorsque la méthode est de type « lecture » les performances sont les mêmes que pour un composant non-persistant : 0,92 ms ;
- lorsque la méthode est de type « écriture », le composant est sérialisé après chaque appel, ce qui dégrade les performances, qui passent à 4,1 ms. Ces trois milli-secondes d'écart correspondent au temps de sérialisation du composant, de son conteneur, de son squelette, et des diverses extensions de ces objets. Bien que le composant lui-même soit vide, le fichier obtenu par sérialisation mesure 2992 octets.

6.2.4 Performances de la protection

Nous avons également mesuré les performances de la protection, en mesurant les performances d'un appel de méthode à distance sur un composant protégé (toujours avec une méthode vide). Dans les mêmes conditions que celles utilisées pour mesurer les performances de la persistance, un tel appel prend 45 ms (!), au lieu de 0,92 ms pour un composant non protégé. Ces très mauvaises performances sont dues en grande partie au chiffrement des messages d'invocation avec l'algorithme RSA (cf. section 5.3.2 page 87). En effet, si on supprime cette fonction, tout en laissant les extensions de cryptographie, on obtient 4,7 ms au lieu de 45. Si on supprime complètement les extensions de cryptographie, on obtient 2,24 ms. La milli-seconde d'écart restante par rapport à un

appel non protégé est due au protocole qui rajoute l'identité de l'appelant dans chaque message d'invocation.

6.2.5 Synthèse

Les résultats qualitatifs obtenus montrent que le surcoût du à l'utilisation de ejava est négligeable par rapport au coût d'un appel de méthode à distance sur un composant normal, et donc a fortiori par rapport au coût d'un appel de méthode à distance sur un composant ayant une ou plusieurs propriétés non-fonctionnelles. Nous pensons donc qu'il est possible de réaliser une plate-forme middleware construite selon l'architecture proposée dans ce document, et ayant des performances similaires aux plates-formes existantes (comme les plates-formes EJB). Toutefois, pour cela, et contrairement à ce qui a été fait dans notre prototype, il faut naturellement utiliser des algorithmes efficaces pour mettre en œuvre les protocoles de communications, les propriétés non-fonctionnelles...

Conclusion

Les plates-formes middleware classiques ne permettent pas de programmer facilement des applications adaptables, dont on a pourtant de plus en plus besoin. Pour essayer de résoudre ce problème, dans le cas particulier de l'adaptation statique (i.e. faite avant l'exécution), nous avons proposé dans cette thèse une architecture de plate-forme modulaire et extensible. Cette architecture est inspirée de celle du modèle EJB, complétée par un nouveau mécanisme de composition d'objets. Afin d'évaluer notre proposition, nous avons réalisé un prototype de plate-forme middleware, nommé JavaPod, selon l'architecture précédente. Puis nous avons réalisé un prototype d'une application réelle à l'aide de JavaPod, et nous avons essayé divers scénarios d'adaptation statique. Ce chapitre effectue une synthèse des résultats obtenus, présentés dans le chapitre précédent, puis conclut en donnant quelques perspectives de recherche.

Synthèse des résultats

Résultats

L'architecture que nous proposons permet d'atteindre la majorité de nos objectifs. En effet, nous avons pu :

- programmer séparément du code fonctionnel de l'application BAGHERA le code non fonctionnel lié à la persistance et à la protection et, de façon partielle, celui lié au mode déconnecté (ce code est actuellement placé dans des *upcalls* ; un travail plus poussé permettrait probablement d'obtenir une séparation complète là aussi) ;
- composer facilement la persistance et la protection, la persistance et le mode déconnecté et, avec un peu plus de travail, la protection et le mode déconnecté (qui ne sont pas indépendantes, contrairement aux deux cas précédents) ;
- étendre la plate-forme JavaPod en ajoutant du code pour une propriété non-fonctionnelle non prévue au départ (gestion de cache dans les connecteurs client-serveur), ou pour étendre les fonctionnalités d'une propriété existante (protection spéciale pour certaines méthodes, dépendant de la valeur de leurs arguments).

Il reste maintenant à savoir si ce résultat peut se généraliser à d'autres propriétés non-fonctionnelles. Malheureusement, faute de pouvoir le prouver formellement, on ne peut qu'essayer de montrer qu'il reste valable dans de nombreux autres cas particuliers, ce qui prendrait beaucoup trop de temps.

Analyse

Le fait que nous ayons pu atteindre nos objectifs, dans le cadre des expérimentations réalisées, découle de plusieurs facteurs. Les principaux sont, selon nous, les suivants :

- le plus important est sans doute le nouveau mécanisme de composition que nous proposons, et que nous avons intégré dans le langage ejava. En effet, *pour faire ce pourquoi nous l'utilisons* (i.e. simuler l'héritage de classes, mais sans combinatoire), il est plus pratique, selon nous, que les mécanismes existants (cf. sections 3.3.2 et 6.1.4) ;
- l'architecture détaillée de la plate-forme JavaPod est également un facteur important. Cela inclut le choix des interfaces et de la granularité des extensions du noyau, ainsi que l'utilisation de principes bien connus, comme la séparation entre interface et implantation ;
- finalement, un autre facteur est le respect du principe consistant à programmer les composants et connecteurs de niveau système de la même façon que les composants et connecteurs applicatifs. Bien que cela soit moins performant a priori que d'utiliser des méthodes de plus bas niveau, cela permet par contre de composer plus facilement les propriétés non-fonctionnelles (cf. section 6.1.2 page 103).

Limitations

Bien que nous ayons pu atteindre à peu près l'ensemble de nos objectifs, les expérimentations effectuées avec l'application BAGHERA mettent malheureusement en évidence quelques limitations, qui semblent intrinsèques :

- pour composer deux propriétés non indépendantes, il faut en général écrire du code spécifique à ces deux propriétés, pour qu'elles puissent fonctionner ensemble correctement. Par exemple, nous avons du écrire du code spécifique pour composer la protection et le mode déconnecté, afin de désactiver la protection en mode déconnecté (cf. section 6.1.2 page 103). On ne peut donc pas s'affranchir de toute combinatoire dans la composition des propriétés ;
- même en cas de séparation *syntaxique* complète entre le code fonctionnel et le code non-fonctionnel, il peut rester des dépendances *sémantiques*. Ces dépendances résiduelles impliquent qu'il n'est pas possible, dans le cas général[4], d'associer à une application une propriété non prévue lors de la mise en œuvre de cette application (sauf en modifiant le code de l'application). Par exemple, le choix des interfaces des composants influe sur les politiques de protection qu'on peut leur associer (cf. section 6.1.1 page 101) ;
- de même, des dépendances entre services systèmes, mêmes séparés (comme entre le protocole `gtp` et le service de persistance - cf. section 6.1.3 page 105), impliquent qu'il n'est pas possible, dans le cas général, d'étendre la plate-forme sans avoir de code existant à modifier.

[4]dire que c'est impossible dans le cas général signifie qu'il existe des cas particuliers où c'est impossible. Cela n'empêche pas que ce soit possible dans de nombreux autres cas.

Il semble donc que, quelle que soit la façon d'aborder le problème, on ne pourra jamais atteindre complètement les objectifs initiaux. Par contre, en considérant dès le départ, c'est à dire avant de réaliser la plate-forme mais aussi les applications, un ensemble de propriétés ou de types de propriétés suffisamment vaste, on diminue les risques d'imprévus, et les limitations précédentes n'apparaissent alors pratiquement plus (sauf la première).

Perspectives

Le travail présenté dans ce document peut être poursuivi de nombreuses façons. Il faudrait tout d'abord faire de nouvelles expérimentations pour compléter l'évaluation des solutions proposées. Il faudrait ensuite trouver des solutions plus satisfaisantes à certains problèmes, qui n'ont été que partiellement résolus. Il faudrait enfin étudier les problèmes qui ont été volontairement laissé de coté.

Nouvelles expérimentations

L'architecture et les mécanismes proposés n'ont été évalués, faute de temps, que sur quelques exemples, et en utilisant un prototype très simplifié. Il serait donc utile de compléter ces évaluations, afin de pouvoir répondre aux questions suivantes :

- les services de persistance et de protection ont été programmés en privilégiant la simplicité, au détriment des performances. Peut-on les mettre en œuvre plus efficacement, tout en gardant l'architecture de la plate-forme et les avantages de la mise en œuvre actuelle de ces propriétés (programmées séparément mais composables entre elles et avec d'autres) ?
- de façon plus générale, le prototype réalisé n'est justement qu'un prototype, où les aspects de bas niveau ont été occultés (pas d'ordonnancement des requêtes coté serveur, utilisation de la sérialisation Java pour éviter l'empaquetage manuel des messages, pas de gestion de ressources...). L'architecture proposée reste-elle applicable à une plate-forme plus « industrielle » ?
- l'architecture détaillée de la plate-forme JavaPod, et en particulier la granularité et les interfaces des extensions du noyau, influe beaucoup sur la modularité et l'extensibilité de la plate-forme. Cette architecture détaillée permet de séparer et composer un certain nombre de propriétés non-fonctionnelles (celles que nous avons mises en œuvre), mais permet-elle de séparer et de composer d'autres propriétés ? La seule façon de le savoir est d'essayer de programmer un nombre assez représentatif de propriétés non-fonctionnelles, mais c'est un travail de longue haleine.

Problèmes partiellement résolus

Sans avoir besoin de nouvelles expérimentations, nous savons déjà que certains problèmes n'ont été que partiellement résolus, et qu'il conviendrait donc d'essayer de mieux les résoudre :

– l'utilisation des scripts de configuration et de déploiement n'est pas très pratique, non seulement parce que le langage utilisé est de trop bas niveau, mais surtout parce que le déploiement doit se faire « à la main », machine par machine. Il serait intéressant d'avoir un mécanisme de (re)déploiement distribué et transactionnel (voir la section suivante) ;
– le mécanisme de protection associé à notre modèle de composition, décrit dans l'annexe B page 125, n'est pas complètement satisfaisant. En particulier, nous ne savons pas comment traiter les extensions non prévues à l'avance, ce qui peut très facilement se produire grâce aux capacités de téléchargement de code de Java.

Problèmes non étudiés

Ainsi que nous l'avons indiqué à plusieurs endroits de ce document, nous avons volontairement écartés certains problèmes, qu'il serait néanmoins intéressant d'étudier :

– nous avons laissé de coté le problème de l'adaptation dynamique et de l'auto-adaptation (cf. le chapitre d'introduction). Le mécanisme de composition de ejava peut certes être utilisé dynamiquement, mais il ne suffit pas à lui seul à résoudre les problèmes précédents. En effet, une reconfiguration dynamique implique généralement de modifier plusieurs objets composés situés sur des machines différentes, de façon atomique si possible. Pour prendre en compte l'adaptation dynamique et l'auto-adaptation, notre mécanisme de composition devrait donc être complété, au minimum, par des mécanismes transactionnels ;
– nous avons également laissé de coté le problème des composants composés de plusieurs composants internes (cf. section 3.1.1 page 36), en disant qu'il semblait difficile d'associer des propriétés non-fonctionnelles à de tels composants, qui peuvent être distribués sur plusieurs sites. L'une des difficultés est que le problème est très dépendant des propriétés non-fonctionnelles considérées. Par exemple, pour la migration, il est préférable de migrer un composant composé d'un seul coup (plutôt que de migrer chaque composant interne un par un), alors que pour la persistance c'est le contraire : il serait en effet inefficace d'enregistrer un composant composé complet alors qu'un seul de ses composants internes aurait été modifié ;
– nous n'avons pas étudié non plus le problème de l'interopérabilité avec les plates-formes existantes, ni celui de l'hétérogénéité (qui ne se pose pas en Java). En particulier, il serait intéressant de pouvoir programmer les extensions de la plate-forme dans n'importe quel langage ([PCB00] propose un modèle inspiré de COM dans un but similaire).

* * *

Pour ceux qui voudraient poursuivre le travail, ou pour ceux qui voudraient simplement utiliser ejava ou la plate-forme JavaPod, le code source de ces logiciels est disponible à l'URL suivante : `http://sirac.inrialpes.fr/Logiciel/`.

Annexe A

Définition du langage ejava

Cette annexe donne une définition du langage ejava plus précise que celle donnée dans la section 4.1.3 page 58. Nous commençons par définir la syntaxe du langage, puis les contraintes de typage associées, vérifiées statiquement. La sémantique du langage est ensuite définie, en s'appuyant sur la sémantique du modèle de composition définie dans la section 3.3.3 page 50.

A.1 Syntaxe

La syntaxe de ejava est une extension de la syntaxe de Java. Nous donnons donc ci-dessous, en gras, les règles de la syntaxe de Java qui ont été modifiées et celles qui ont été ajoutées. Les règles de la syntaxe de Java ne figurant pas ci-dessous restent inchangées. Nous utilisons ici les mêmes conventions que celles de la spécification officielle de Java : *The Java Language Specification*, ou JLS [JLS].

Déclarations de classes et d'interfaces (JLS, §8.1 et §9.1)

Les règles suivantes permettent d'ajouter les mots-clefs `extensible` et `extension` dans l'en-tête des classes et des interfaces. Elles permettent également de définir une clause `dextends` dans l'en-tête des classes :

ClassDeclaration :

> *ClassModifiers*$_{opt}$ `class` *Identifier Super*$_{opt}$ ***DSuper***$_{opt}$ *Interfaces*$_{opt}$ *ClassBody*

DSuper *:*

> **dextends** *InterfaceTypeList*

ClassModifier : one of

> `public` `abstract` `final` **extensible extension**

InterfaceModifier : one of

> `public` `abstract` **extensible**

Modifieurs de méthodes (§8.4.3)

La règle définissant les modifieurs de méthodes est étendue pour permettre de définir des méthodes `extensible` et/ou `reversed` :

MethodModifier : one of

extensible **reversed** `public` `protected` `private` `abstract` `static` `final` `synchronized` `native`

Expressions d'invocation de méthodes (§15.11)

La règle définissant la syntaxe des expressions d'invocation de méthode est étendue d'une part de façon à autoriser les noms de méthode précédés d'un nom de classe et éventuellement d'un nom de paquetage par des « :: », d'autre part pour introduire la pseudo-expression `sub`, qui correspond à `super` pour les méthodes inversées, et les pseudo-expressions `dsuper` et `dsub` qui correspondent à `super` et à `sub` pour les méthodes extensibles.

MethodInvocation :

QualifiedMethodName (*ArgumentList*$_{opt}$)
Primary . ***QualifiedMethodName*** (*ArgumentList*$_{opt}$)
`super` . *Identifier* (*ArgumentList*$_{opt}$)
sub . ***Identifier*** (***ArgumentList***$_{opt}$)
dsuper . ***Identifier*** (***ArgumentList***$_{opt}$)
dsub . ***Identifier*** (***ArgumentList***$_{opt}$)

QualifiedMethodName *:*

Identifier
QualifiedMethodName :: ***Identifier***

Expressions primaires (§15.7)

Finalement, la règle définissant les expressions primaires est étendue par l'ajout du mot-clef `base`. `base` est une expression prédéfinie qui désigne l'objet extensible de l'objet composé auquel appartient l'objet courant `this`.

PrimaryNoNewArray :

Literal
`this`
base
(*Expression*)
ClassInstanceCreationExpression
FieldAccess
MethodInvocation
ArrayAccess

A.2 Contraintes statiques

Les règles de syntaxe précédentes sont complétées par les contraintes suivantes, vérifiées à la compilation.

Déclarations de classes et d'interfaces

– la super-classe d'une classe *normale*, c'est à dire d'une classe qui n'est ni une classe extensible ni une classe d'extension, doit être une classe normale ;
– la super-classe d'une classe extensible doit être une classe normale ou extensible. De même, la super-classe d'une classe d'extension doit être une classe normale ou une classe d'extension ;
– la clause `dexends` ne peut être utilisée que dans les classes d'extension. De plus, les noms spécifiés dans cette clause doivent désigner des *interfaces*, dont toutes les méthodes, y compris celles héritées, doivent être déclarées extensibles ;
– en Java, les méthodes provenant d'une classe, de sa super-classe et des interfaces qu'elle implante doivent être *compatibles* : si au moins deux des classes ou interfaces précédentes contiennent deux méthodes de même nom et de même signature, alors, entre autres contraintes, les types de retour de ces méthodes doivent être identiques. Cette règle est étendue en ejava : les méthodes provenant d'une classe, de sa super-classe, des interfaces qu'elle implante *et des interfaces de la clause* `dextends` doivent être compatibles.

Modifieurs de méthodes

– le mot-clef `extensible` ne peut être utilisé comme modifieur de méthode que dans les classes extensibles et les classes d'extensions. De plus, il ne peut pas être utilisé pour les constructeurs, ni pour les méthodes `final`, `static` ou `native` ;
– si une méthode est déclarée extensible dans une classe C ou une interface I, alors elle doit être déclarée extensible dans toutes les sous-classes de C et dans les classes implantant I. A l'inverse, si une méthode *n'est pas* déclarée extensible dans une classe C ou une interface I, alors elle *peut* être déclarée extensible dans une sous-classe de C ou dans une classe implantant I ;
– le mot-clef `reversed` peut être utilisé dans n'importe quelle classe, mais pas pour les constructeurs ni pour les méthodes déclarées `final`, `static` ou `native` ;
– si une méthode est déclarée `reversed` dans une classe C ou une interface I, alors elle doit être déclarée `reversed` dans toutes les sous-classes de C et dans les classes implantant I. De même, si une méthode *n'est pas* déclarée `reversed` dans une classe C ou une interface I, alors elle *ne doit pas* être déclarée `reversed` dans les sous-classes de C et dans les classes implantant I.

Expressions d'invocation de méthodes

– un appel de méthode du type $[o.]p_1 :: ... :: p_n :: m(...)$, avec $n \geq 1$, doit respecter les contraintes suivantes :

  - $p_1 . \dots . p_n$ doit être le nom (éventuellement précédé d'un nom de paquetage) d'une classe ou d'une interface. On peut ne pas préciser le nom du paquetage (i.e. $n = 1$), auquel cas le nom p_1 est résolu, comme pour les autres noms de classes, en utilisant les clauses `import`,
  - l'expression $[o.]p_1 :: \dots :: p_n :: m(\dots)$ doit de plus respecter les mêmes contraintes que si on la remplaçait par $o'.m(\dots)$, où o' serait de type statique $p_1 . \dots . p_n$. o n'est donc pas nécessairement un sous-type du type $p_1 . \dots . p_n$;
- un appel du type `super.`$m(\dots)$ peut être utilisé si m est une méthode normale ou extensible, mais pas si c'est une méthode inversée ;
- un appel du type `sub.`$m(\dots)$ ne peut être utilisé que si m est une méthode inversée et non extensible, déclarée dans la classe courante ou l'une de ses super-classes ;
- un appel du type `dsuper.`$m(\dots)$ ne peut être utilisé que si m est une méthode non inversée déclarée dans l'une des interfaces de la clause *dextends* ;
- un appel du type `dsub.`$m(\dots)$ ne peut être utilisé que si m est une méthode inversée et extensible, déclarée dans la classe courante ou dans une super-classe.

Expressions primaires

Le mot-clef `base` ne peut être utilisé que dans les classes d'extensions. Son type statique est `java.lang.Object`. Comme pour le mot-clef `this`, on ne peut pas lui affecter une valeur.

A.3 Classes prédéfinies

Le langage ejava utilise une version étendue de certaines classes des paquetages `java.lang` et `java.lang.reflect`, afin de permettre d'associer des extensions aux objets, et afin de compléter les fonctions d'introspection de Java :

- la classe `Object` est étendue par l'ajout des méthodes suivantes :
  - `public final Object[] getExtensions ()` : retourne la liste ordonnée des extensions associées à cet objet,
  - `public final void setExtensions (Object[] exts)` : remplace les extensions associées à cet objet par les extensions `exts` (cf. section suivante) ;
- la classe `Class` est étendue par l'ajout des méthodes suivantes :
  - `public int getExtendedModifiers ()` : retourne les modifieurs étendus (i.e. `extensible` et `extension`) déclarés dans l'en-tête de cette classe, ce qui permet de savoir si c'est une classe extensible ou une classe d'extension,
  - `public Class[] getExtendedInterfaces ()` : retourne la liste des interfaces déclarées dans la clause `dextends` de cette classe ;
- la classe `Method` est étendue par l'ajout de la méthode `getExtendedModifiers`, qui permet de savoir si une méthode est extensible et/ou inversée ;
- la classe `Modifier` est étendue par l'ajout de trois nouvelles constantes, nommées `EXTENSIBLE`, `EXTENSION` et `REVERSED`, et de trois méthodes statiques, nommées `isExtensible`, `isExtension` et `isReversed`, et similaires aux méthodes existantes de cette classe.

Enfin, trois nouvelles classes d'exception sont ajoutées au paquetage `java.lang` :

- l'exception `UnextensibleObjectException` est lancée lorsqu'on tente d'utiliser les méthodes `getExtensions` et `setExtensions` de la classe `Object` sur un objet qui n'est pas extensible ;
- l'exception `IncompatibleExtensionsException` est lancée lorsque les extensions passées en argument de la méthode `setExtensions` sont incompatibles entre elles ou avec l'objet extensible (voir la section suivante) ;
- l'exception `UndefinedMethodException` est lancée lorsqu'on tente d'appeler une méthode extensible sur un objet composé dans lequel cette méthode n'est pas définie (ce qui peut se produire, par exemple, quand on utilise `dsuper` ou des noms de méthodes avec des ::). Cette classe d'exception hérite de `RuntimeException`, ce qui fait que le programmeur n'est pas obligé de capturer ce type d'exception (comme pour les exceptions `NullPointerException` par exemple).

Remarques :

- à cause de ces extensions, les programmes ejava ne sont pas portables à 100%. En effet, ils ne peuvent pas fonctionner dans une machine virtuelle lancée sans l'option `-bootclasspath`, de façon à remplacer la version par défaut des classes systèmes ci-dessus. De plus, ces classes systèmes étendues ne sont elles-mêmes pas portables d'une machine virtuelle à une autre. Pour résoudre ce dernier problème, nous avons réalisé un programme de « portage » automatique qui, grâce à une librairie de manipulation de *bytecode* [Dah99], est capable de transformer les classes systèmes par défaut d'une machine virtuelle, quelles qu'elles soient, en des classes étendues pouvant fonctionner sur cette même machine virtuelle. Malheureusement, ce programme ne peut pas fonctionner si certaines méthodes originales sont natives. Il fonctionne en tout cas pour au moins quatre machines virtuelles différentes : le JDK 1.2 et le JDK 1.3, pour Windows et pour Linux ;
- au besoin, il est possible de fournir toutes les méthodes ci-dessus sous forme de méthodes `static` dans une classe nouvelle, appelée par exemple `EJava`. Cela permet d'obtenir du code 100% portable, mais conduit une intégration moins « élégante » et moins « naturelle » des fonctionalités de ejava dans Java.

A.4 Sémantique

La sémantique de ejava est essentiellement la même que celle de Java. En effet, tout programme Java, considéré comme un programme ejava, garde exactement la même sémantique. Les différences sémantiques entre ejava et Java ne concernent donc que les quelques extensions syntaxiques apportées par ejava.

Méthodes extensibles

La sémantique des appels de méthodes extensibles, inversées ou non, découle de la définition de notre mécanisme de composition. Se reporter à la section 3.3.3 page 50 pour

plus de précisions à ce sujet. De même pour la sémantique des appels du type `dsuper.`m ou `dsub.`m. Les appels du type `super.`m gardent leur sémantique habituelle, mêmes pour des méthodes extensibles.

Méthodes inversées

Le mot-clef `reversed` a été introduit pour pouvoir définir des méthodes inversées *extensibles*, utiles pour simuler des piles de protocoles. Ayant introduit ce concept pour les méthodes extensibles, il nous a semblé naturel de le généraliser à toutes les méthodes, et c'est pourquoi on peut utiliser ce mot-clef également pour des méthodes non extensibles, en conjonction avec la pseudo expression `sub`. La sémantique des méthodes non extensibles mais inversées est similaire à celle des méthodes extensibles inversées. On peut également la définir à l'aide d'un programme Java équivalent, comme le montre la figure A.1 (le programme ejava de gauche est équivalent au programme Java de droite).

```
class C {
  reversed void m (...) {
    // ...
    sub.m(...);
    // ...
  }
}
class D extends C {
  reversed void m (...) {
    // ..
    sub.m(...);
    // ...
  }
}
```

```
class C {
  void m (...) {
    // ...
    subm(...);
    // ...
  }
  void subm (...) {
    throw new UndefinedMethodException();
  }
}
class D extends C {
  void subm (...) {
    // ..
    subsubm(...);
    // ...
  }
  void subsubm (...) {
    throw new UndefinedMethodException();
  }
}
```

FIG. A.1 – Sémantique des méthodes inversées

L'expression `base`

La pseudo expression `base` référence l'objet extensible de l'objet composé auquel appartient l'objet courant `this`. Si l'objet courant est une extension ne faisant pas partie d'un objet composé, la valeur de cette expression est `null`.

Synchronisation

La méthode `setExtensions` et les méthodes extensibles sont synchronisées par un algorithme de type lecteurs-rédacteurs : plusieurs méthodes extensibles peuvent s'exé-

cuter de façon concurrente, mais pendant qu'un *thread* modifie un objet composé, avec la méthode `setExtensions`, aucun autre thread ne peut exécuter cette même méthode, ni aucune méthode extensible (cette synchronisation se fait actuellement de façon globale, mais on pourrait la rendre plus souple, en l'appliquant de façon indépendante à chaque objet composé).

Comme en Java, déclarer une méthode `synchronized` est équivalent à placer son code dans un bloc `synchronized (this) {}`. Pour les objets extensibles et pour les extensions, on obtient donc une synchronisation sur le verrou d'un *membre* de l'objet composé, et non pas sur l'objet composé lui-même (qui n'a pas d'existence concrète, et donc pas de verrou associé). Pour simuler une telle synchronisation globale, il faut utiliser un bloc `synchronized (base) {}` (puisqu'on peut assimiler l'objet extensible, invariable, à l'objet composé entier).

La méthode `setExtensions`

Avant de changer les extensions d'un objet `o`, la méthode `setExtensions` procède aux vérifications suivantes :

- `o` doit être un objet extensible ;
- les objets passées en paramètre doivent être des extensions ;
- chaque extension passée en paramètre doit être associée soit à `o` soit à aucun objet extensible ;
- toutes les méthodes « requises » par une extension doivent être « fournies » par au moins un membre situé avant cette extension. Plus précisément, pour chaque extension `e` passée en paramètre, et pour chaque méthode `m` déclarée dans au moins une interface de la clause `dextends` de `e`, il doit exister un membre situé avant `e` dans lequel `m` est définie (en tant que méthode extensible, avec le même type de retour et une clause `throws` compatible) ;
- les méthodes qui se surchargent dynamiquement doivent être « compatibles ». Plus précisément, si `m` est une méthode extensible définie dans un membre `e`, et si un membre situé après `e` contient une méthode extensible `m'` de même nom et de même signature que `m`, alors :
  - `m` et `m'` doivent avoir le même type de retour et des clauses `throws` compatibles,
  - `m` et `m'` doivent être soit toutes les deux inversées, soit toutes les deux non-inversées.

Si l'une de ces vérifications échoue, la méthode `setExtensions` retourne une exception et laisse les extensions de `o` inchangées. Sinon, les anciennes extensions de `o` sont dissociées de cet objet, et les nouvelles extensions lui sont associées. Autrement dit, la méthode `setExtensions` est atomique : soit elle échoue, soit elle réussit.

Annexe B

Un mécanisme de contrôle d'accès pour ejava

Cette annexe présente un mécanisme de contrôle d'accès pour les objets composés. La première section explique pourquoi ce mécanisme a été défini. La deuxième section présente le mécanisme lui-même, et la troisième section donne quelques indications sur sa mise en œuvre dans le cadre du langage ejava.

B.1 Motivation

De la même façon qu'un système d'exploitation est généralement utilisé par plusieurs applications s'exécutant en parallèle, une plate-forme middleware sert généralement de support d'exécution à plusieurs applications distribuées, non connues à l'avance, et s'exécutant en parallèle. Par conséquent, comme un système d'exploitation, une plate-forme middleware doit se protéger vis-à-vis des applications qu'elle accueille, et doit également protéger chaque application des autres. Ce qui signifie qu'elle doit empêcher les applications d'espionner ou de perturber les autres applications, ainsi que la plate-forme elle-même. Cela signifie également qu'elle doit assurer un partage équitable des ressources (mémoire, processeur, réseau...).

Or la plate-forme JavaPod, telle qu'elle est présentée dans le chapitre 4, n'offre aucun moyen d'assurer ce type protection. Par exemple, n'importe quelle application a accès à l'objet composé représentant le serveur de composants local et, de là, à tous les conteneurs, à tous leurs talons et squelettes, et à tous leurs composants encapsulés. De plus, tous ces objets composés sont facilement modifiables : n'importe quelle application peut y ajouter, modifier ou supprimer des extensions.

Pour pouvoir résoudre ce problème, tout en conservant une architecture ouverte et extensible, où les applications peuvent étendre dynamiquement la plate-forme avec leurs propres extensions (comme c'est le cas pour BAGHERA - cf. section 5.3.3 page 88), nous proposons d'inclure un mécanisme de contrôle d'accès minimal dans le noyau de la plate-forme. *Le but de ce mécanisme n'est pas de résoudre tous les problèmes de protection*, mais simplement de servir de base de départ pour des mécanismes de protection plus

évolués, fournis par les extensions du noyau (de la même façon qu'en Java les mots clefs `private`, `protected` et `public` peuvent servir de base pour des mécanismes de protection plus évolués, basés par exemple sur des capacités cachées [HI97]).

Or, pour le noyau, la plate-forme JavaPod est constituée d'un ensemble d'objets composés, représentant des serveurs, des conteneurs, des talons et des squelettes. Il était donc naturel, pour inclure un mécanisme de protection minimal dans ce noyau, d'utiliser un mécanisme de contrôle d'accès basé sur la notion d'objet composé.

B.2 Modèle

Cette section présente le mécanisme de contrôle d'accès que nous proposons pour notre concept d'objet composé, après quelques rappels sur le contrôle d'accès en général.

Définitions

Le but des mécanismes de *contrôle d'accès* est d'empêcher certains utilisateurs ou programmes, appelés *sujets*, d'accéder à certaines données ou fonctions, appelées *objets*, selon certains *modes d'accès* (lecture, écriture, exécution...).

On peut représenter les droits de chaque sujet sur chaque objet, à un instant donné, sous forme d'une matrice que l'on appelle la *matrice d'accès* [Lam71]. Chaque ligne correspond à un sujet, et chaque colonne correspond à un objet. Chaque case de cette matrice contient les modes d'accès autorisés pour un couple (sujet,objet). Par exemple, la matrice ci-dessous indique que l'utilisateur `u1` peut lire et modifier le fichier `f1`, mais que l'utilisateur `u2` peut seulement lire ce fichier.

	f1	f2
u1	rw	r
u2	r	rw

FIG. B.1 – Un exemple de matrice d'accès

Cette matrice est souvent très grande, mais aussi essentiellement creuse. Elle est donc souvent représentée en compactant les colonnes ou les lignes. Dans le premier cas, on obtient pour chaque objet une liste de couples (identité d'un sujet, modes d'accès correspondants pour ce sujet), que l'on stocke directement avec l'objet lui-même. On parle dans ce cas de *liste de contrôle d'accès* ou ACL (*access control list*). Dans le cas où la matrice est compactée par lignes, on obtient pour chaque sujet une liste de couples (identité d'un objet, modes d'accès correspondants pour cet objet), stockée avec les informations relatives au sujet considéré. Un tel couple est désigné par le terme de *capacité*.

Lorsque la matrice précédente peut évoluer dynamiquement, du fait de l'activité des différents sujets (modification des listes d'accès, passage de capacités entre sujets, création dynamique de nouveaux objets...), on parle de contrôle d'accès *discrétionnaire*.

A l'inverse, lorsque cette matrice n'est pas modifiable dynamiquement, on parle de contrôle d'accès *non discrétionnaire*, ou obligatoire (*mandatory access control*).

Mécanisme de contrôle d'accès pour les objets composés

Le mécanisme que nous proposons est un mécanisme non-discrétionnaire, basé sur une matrice d'accès définie statiquement. Les sujets considérés sont les *classes* d'objets extensibles et d'extensions, alors que les « objets » sont les méthodes extensibles de ces classes. Les modes d'accès possibles sont `e`(xecute) et `o`(verride). Finalement, lorsqu'une classe `C` appelle une méthode dans une classe `D`, celle-ci s'exécute avec les droits de l'appelé, c'est à dire avec ceux de `D`.

La figure B.2 permet d'illustrer ces définitions. Elle montre la matrice d'accès que l'on pourrait utiliser dans le cas de l'objet composé de la figure 3.11 page 51. Selon cette matrice, les extensions de type `Fidelity` ont le droit de surcharger la méthode `withdraw`. De même, les extensions de type `Bonus` ont le droit de surcharger la méthode `addPoints`, et ont le droit d'appeler les méthodes `deposit`, `getPoints` et `resetPoints`. De plus, implicitement, chaque classe à le droit d'appeler ses propres méthodes (cases en italique sur la figure).

	`getBalance`	`withdraw`	`deposit`	`getPts`	`addPts`	`resetPts`
`Account`	*e*	*e*	*e*			
`Fidelity`		`o`		*e*	*e*	*e*
`Bonus`			`e`	`e`	`o`	`e`

FIG. B.2 – Matrice d'accès pour l'objet composé de la figure 3.11 page 51

Ce mécanisme est donc finalement assez similaire à celui-ci de Java, basé sur les mots-clefs `private`, `protected`, `public` et `final`. En effet, les sujets, les objets et les modes d'accès sont les mêmes dans les deux cas. La seule différence est que la matrice d'accès est spécifiée de façon implicite et directement dans le code en Java, alors que nous préférons la spécifier de façon explicite, dans un fichier unique séparé du code. Ce choix permet en effet de modifier facilement la matrice d'accès, et permet également de spécifier plus finement les droits de chaque classe.

L'inconvénient principal de ce mécanisme est qu'il peut difficilement prendre en compte l'ajout dynamique de nouvelles classes d'extensions, comme cela peut se produire lorsque les applications étendent dynamiquement la plate-forme avec leurs propres extensions. En effet, dans ce cas, il faudrait pouvoir étendre dynamiquement la matrice d'accès, afin de prendre en compte les nouvelles extensions. Malheureusement, bien qu'il soit assez simple de remplacer dynamiquement la matrice d'accès par une autre, encore faut-il savoir *comment* étendre la matrice initiale : si les nouvelles classes peuvent spécifier les droits qu'elles accordent aux anciennes (cases B et D sur la figure B.3 page suivante), il est évidemment hors de question qu'elles s'accordent elles-mêmes des droits sur les anciennes classes (case C).

	anciennes méthodes	*nouvelles méthodes*
anciennes classes	A	*B*
nouvelles classes	*C*	*D*

FIG. B.3 – Extension d'une matrice d'accès lors de l'ajout de nouvelles classes

B.3 Mise en œuvre

Le mécanisme de contrôle d'accès défini ci-dessus a été partiellement mis en œuvre dans le langage ejava. Les principales caractéristiques de cette mise en œuvre sont les suivantes (on suppose ici que le lecteur est familier avec le nouveau mécanisme de contrôle d'accès introduit dans le JDK 1.2 [GMPS97]) :

- la matrice d'accès est spécifiée en utilisant la syntaxe des fichiers *policy*. Nous pouvons ainsi intégrer notre mécanisme de contrôle d'accès avec celui du JDK 1.2, par exemple pour accorder à certaines classes d'extensions le droit d'accéder au système de fichiers, au réseau...
- le contrôle d'accès proprement dit est effectué lors de chaque appel à une méthode extensible, grâce à du code supplémentaire placé juste avant le code de redirection pour cette méthode (cf. section 4.1.4 page 60) ;
- ce code supplémentaire détermine la classe de l'appelant grâce aux possibilités d'introspection de la pile d'exécution, introduites dans le JDK 1.2. Il détermine ensuite les droits de cette classe en consultant la matrice d'accès, puis vérifie que la méthode appelée à bien le droit de l'être ;
- aucun contrôle d'accès n'est effectué si aucun `SecurityManager` n'est installé, afin de ne pas dégrader les performances quand ce mécanisme n'est pas nécessaire (les mécanismes précédents sont en effet assez coûteux).

Cependant, cette mise en œuvre n'est que partielle : en effet, il est très facile de contourner les contrôles d'accès en utilisant directement les méthodes auxiliaires générées par le compilateur ejava (cf. section 4.1.4 page 60). Pour résoudre ce problème, la solution la plus efficace serait de vérifier au chargement des classes qu'elles utilisent ces méthodes de façon légale, c'est à dire de la même façon qu'elles sont utilisées dans le code produit par le compilateur ejava. C'est certainement faisable[1], mais probablement complexe à mettre en œuvre et, comme cela nous aurait écarté de notre préoccupation principale, nous n'avons pas essayé de le faire.

[1] à condition d'empêcher l'utilisation de ces méthodes par l'intermédiaire des classes du paquetage `java.lang.reflect`, qui ne serait pas contrôlable (problèmes d'indécidabilité). Heureusement ceci est facile à assurer : il suffit de modifier ces classes, afin de masquer à l'utilisateur toutes ces méthodes auxiliaires (facilement repérables car leurs noms sont de la forme `$...$`)

Bibliographie

[ASM] The asm package.
URL http://sirac.inrialpes.fr/Logiciel/

[AT98] M. Aksit et B. Tekinerdogan, Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. *ECOOP'98 Aspect-Oriented Programming Workshop*, Brussels, Belgium, juillet 1998.

[Bal00] N. Balacheff, Teaching, an emergent property of eLearning environments. *The Information Society for All (IST)*, Nice, France, novembre 2000.

[BCCD00] G. S. Blair, G. Coulson, F. Costa et H. A. Duran, On the design of reflective middleware platforms. *Workshop on Reflective Middleware (RM'2000)*, New-York, USA, avril 2000.

[BCRP98] G. S. Blair, G. Coulson, P. Robin et M. Papathomas, An architecture for next generation middleware. *International Conference on Distributed Systems Platforms and Open Distributed Processing (MIDDLEWARE)*, The Lake District, UK, septembre 1998.

[BR00a] E. Bruneton et M. Riveill, JavaPod : an adaptable and extensible component platform. *Workshop on Reflective Middleware (RM'2000)*, New-York, USA, avril 2000.

[BR00b] E. Bruneton et M. Riveill, JavaPod : une plate-forme à composants adaptable et extensible. Rapport technique RR-3850, INRIA, janvier 2000.

[BR00c] E. Bruneton et M. Riveill, Reflective implementation of non-functional properties with the JavaPod component platform. *Workshop on Reflection and Metalevel Architectures (RMA'2000)*, Cannes, France, juin 2000.

[BS99] N. M. N. Bouraqadi-Saâdani, Un cadre réflexif pour la programmation par aspects. *Langages et Modèles à Objets*, Villefranche sur Mer, France, janvier 1999.

[CBC98] F. M. Costa, G. S. Blair et G. Coulson, Experiments with reflective middleware. *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, juillet 1998.

[CCM99] CORBA 3.0 CCM FTF draft ptc/99-10-04, octobre 1999.

[CM93] S. Chiba et T. Masuda, Designing an extensible distributed language with a meta-level architecture. *European Conference on Object-Oriented Programming (ECOOP)*, Kaiserslautern, Germany, juillet 1993.

[Dah99] M. Dahm, Byte code engineering. *Java Informations Tage (JIT'99)*, Düsseldorf, Germany, septembre 1999.

[DHTS98] B. Dumant, F. Horn, F. D. Tran et J.-B. Stefani, Jonathan : an open distributed processing environment in Java. *International Conference on Distributed Systems Platforms and Open Distributed Processing (MIDDLEWARE)*, The Lake District, UK, septembre 1998.

[EJA] The ejava compiler.
URL `http://sirac.inrialpes.fr/Logiciel/`

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

[GC96] B. Gowing et V. Cahill, Meta-object protocols for C++ : The Iguana approach. *Reflection'96*, San Francisco, California, USA, avril 1996.

[GK98] M. Golm et J. Kleinöder, metaXa and the future of reflection. *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, octobre 1998.

[GMPS97] L. Gong, M. Mueller, H. Prafullchandra et R. Schemers, Going beyond the sandbox : an overview of the new security architecture in the Java Development Kit 1.2. *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, USA, décembre 1997.

[HBDH98] R. Hayton, M. H. Bursell, D. Donaldson et A. Herbert, Mobile Java objects. *International Conference on Distributed Systems Platforms and Open Distributed Processing (MIDDLEWARE)*, The Lake District, UK, septembre 1998.

[HHD98] R. Hayton, A. Herbert et D. Donaldson, FlexiNet : a flexible, component oriented middleware system. *European Workshop Support for Composing Distributed Applications*, Sintra, Portugal, septembre 1998.

[HI97] D. Hagimont et L. Ismaïl, A protection scheme for mobile agents on Java. *3rd ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, Budapest, Hungary, septembre 1997.

[HL98] D. Hagimont et D. Louvegnies, Javanaise : Distributed shared objects for internet cooperative applications. *International Conference on Distributed Systems Platforms and Open Distributed Processing (MIDDLEWARE)*, The Lake District, UK, septembre 1998.

[JBS] The Java Beans specification.
URL `http://java.sun.com/products/javabeans/docs/spec.html`

[JCE] Java code engineering & reverse engineering.
URL `http://www.meurrens.org/ip-Links/Java/codeEngineering/`

[JLS] The Java language specification.
URL `http://java.sun.com/docs/books/jls/html/index.html`

[KdRB91] G. Kiczales, J. des Rivières et D. Bobrow, *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier et J. Irwin, Aspect-oriented programming. *European Conference on Object-Oriented Programming (ECOOP)*, Jyväskylä, Finland, juin 1997.

[KOP] The KOPI project.
URL `http://www.dms.at/kjc/`

[Lam71] B. W. Lampson, Protection. *Fifth Princeton Symposium on Information Sciences and Systems*, pp. 437–443, Princeton University, mars 1971.

[Led98] T. Ledoux, Adaptabilité dynamique des aspects pour la construction d'applications réparties ouvertes. *Colloque International sur les Nouvelles Technologies de la Répartition (NOTERE)*, Montréal, Canada, octobre 1998.

[Led99] T. Ledoux, OpenCorba : a reflective open broker. *2nd International Conference on Metalevel Architectures and Reflection (REFLECTION)*, Saint-Malo, France, juillet 1999.

[LK97] C. V. Lopes et G. Kiczales, D : A language framework for distributed programming. Rapport technique SPL97-010, Xerox Palo Alto Research Center, février 1997.

[LK98] C. V. Lopes et G. Kiczales, Recent developments in AspectJ. *ECOOP'98 Aspect-Oriented Programming Workshop*, Brussels, Belgium, juillet 1998.

[Lun98] C. P. Lunau, Is composition of metaobjects = aspect oriented programming. *ECOOP'98 Aspect-Oriented Programming Workshop*, Brussels, Belgium, juillet 1998.

[McA93] J. McAffer, The CodA MOP. *ECOOP'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Kaiserslautern, Germany, juillet 1993.

[McA95a] J. McAffer, Meta-level architecture support for distributed objects. *International Workshop on Object-Orientation in Operating Systems*, Lund, Sweden, août 1995.

[McA95b] J. McAffer, Meta-level programming with CodA. *European Conference on Object-Oriented Programming (ECOOP)*, Aarhus, Denmark, août 1995.

[OB99] A. Oliva et L. E. Buzato, The design and implementation of Guaraná. *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, San Diego, California, USA, mai 1999.

[ODP95a] ODP reference model : Architecture. ITU-T Recommendation X.903 | ISO/IEC International Standard 10746-3, novembre 1995.

[ODP95b] ODP reference model : Foundations. ITU-T Recommendation X.902 | ISO/IEC International Standard 10746-2, novembre 1995.

[Paw99] R. Pawlak, TOS : A class-based reflective language on Tcl. Rapport technique TR-9902, Laboratoire CEDRIC-CNAM, février 1999.

[PCB00] N. Parlavantzas, G. Coulson et G. S. Blair, Applying component frameworks to develop flexible middleware. *Workshop on Reflective Middleware (RM'2000)*, New-York, USA, avril 2000.

[PDF99] R. Pawlak, L. Duchien et G. Florin, An automatic aspect weaver with a reflective programming language. *2nd International Conference on Metalevel Architectures and Reflection (REFLECTION)*, Saint-Malo, France, juillet 1999.

[RGL98] P.-G. Raverdy, R. L. V. Gong et R. Lea, Dart : A reflective middleware for adaptive applications. *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, octobre 1998.

[Smi82] B. C. Smith, *Procedural Reflection in Programming Languages*. Thèse de doctorat, MIT, 1982. Available as MIT Laboratory of Computer Science Technical Report 272.

[SSC97] A. Singhai, A. Sane et R. Campbell, Reflective ORBs : Supporting robust, time-critical distribution. *ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Systems*, Jyväskylä, Finland, juin 1997.

[US87] D. Ungar et R. B. Smith, Self : The power of simplicity. *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, Orlando, USA, octobre 1987.

[WS98] I. Welch et R. Stroud, Dalang - a reflective Java extension. *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, octobre 1998.