

Une infrastructure pour middleware adaptable

Pierre-Charles DAVID

encadré par Thomas LEDOUX, École des Mines de Nantes

Institut de Recherche en Informatique de Nantes
2, rue de la Houssinière
B.P. 92208
F-44322 NANTES CEDEX 3



RAPPORT DE STAGE DE DEA

Septembre 2001

Pierre-Charles DAVID (encadré par Thomas LEDOUX, École des Mines de Nantes)
Une infrastructure pour middleware adaptable

© Septembre 2001 par Pierre-Charles DAVID

rapport.tex – Une infrastructure pour middleware adaptable – 6/9/ 2001 – 16:46

Une infrastructure pour middleware adaptable

Pierre-Charles DAVID (encadré par Thomas LEDOUX, École des Mines de Nantes)

pcdavid@emn.fr

Chapitre 1

Introduction

Ce rapport présente le travail que j'ai réalisé lors de mon stage de DEA d'Informatique de l'Université de Nantes. Le stage lui-même a été effectué au sein de l'équipe *Objets, Composants et Modèles* (OCM) du Département Informatique de l'École des Mines de Nantes.

L'objectif de ce stage était de réfléchir à la notion d'adaptabilité dynamique des systèmes informatiques et à sa mise en œuvre grâce aux techniques réflexives, en particulier dans le contexte du *middleware*.

1.1 Motivation

Les systèmes informatiques sont de plus en plus complexes et surtout, grâce à l'essor de l'informatique nomade, se retrouvent partout. Alors qu'il y a quelques années la plupart des applications étaient conçues pour fonctionner dans un environnement d'exécution relativement bien connu et maîtrisé, cela n'est plus possible dans le contexte actuel, où les conditions d'exécution possibles pour une application donnée sont très variées.

D'une part, les développeurs ont à faire face à une grande diversité de plates-formes d'exécution et d'environnements logiciels associés. Le nombre et la variété de ces plate-formes n'a jamais été aussi grand, allant du simple téléphone portable doté de capacités minimales au cluster de plusieurs dizaines d'ordinateurs multi-processeurs, en passant par l'ordinateur de bureau. Pourtant, grâce au développement considérable d'internet, toutes ces machines hétéroclites participent du même réseau global et les applications actuelles doivent être capables de gérer cette diversité, voire d'en tirer partie.

D'autre part, la nature même de ces nouvelles plate-formes (assistants personnels, téléphones portables...), rend la tâche des développeurs encore plus complexe. En effet, leurs possibilités de traitement et de stockage limitées les rendent extrêmement dépendantes de leurs capacités de communications avec l'extérieur, qui sont elles-mêmes très variées, et surtout qui peuvent changer dynamiquement au cours de l'exécution d'une application (bande passante hautement variable par exemple).

Toute cette complexité, à la fois statique (plate-formes matérielles) et dynamique (ressources disponibles variables au cours du temps) rend le travail des programmeurs plus complexe qu'il ne l'a jamais été. La solution réside évidemment dans l'écriture d'applications capables de s'adapter à ces conditions d'exécutions nouvelles ou changeantes.

Cette problématique est peu prise en compte actuellement (même si de nombreux travaux de

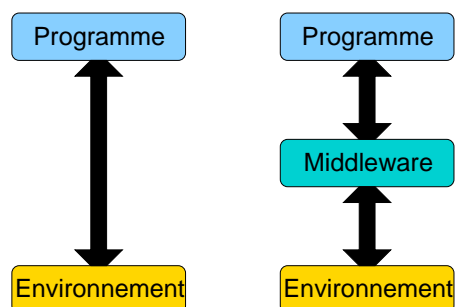


FIG. 1.1 – Introduction de la couche *middleware*

recherche commencent à s’y intéresser), et l’écriture d’une application *adaptable* implique un gros travail supplémentaire de la part du programmeur. Il doit en effet anticiper les différentes conditions dans lesquelles son application pourra être exécutée et prévoir les réactions appropriées. Ces aspects de la programmation, qui ne sont pas liés directement aux fonctionnalités fournies par les applications, ne devraient pas nécessiter de la part des programmeurs autant d’attention qu’actuellement. Il serait préférable de prendre en charge de façon la plus automatisée possible cet aspect du développement.

1.2 Contexte

Le problème évoqué ci-dessus ne date pas d’aujourd’hui, même si les évolutions technologiques récentes l’ont rendu plus aigu. L’essence du problème réside dans le fort couplage qui existe entre une application et son environnement d’exécution. Cet environnement évoluant typiquement plus rapidement que l’application elle-même, un couplage trop fort de l’application avec son environnement implique un gros travail d’adaptation lorsque ce dernier est modifié. Étant donné la rapidité des évolutions technologiques, il n’est pas rare que la durée de vie d’une application corresponde à plusieurs générations de machines et de technologies logicielles.

La solution proposée jusqu’à aujourd’hui a consisté à découpler ces deux éléments (programme et environnement) en introduisant un niveau d’indirection, baptisé *middleware* (ou intergiciel¹ en français), comme le montre la figure 1.2. Un des rôles de cette couche logicielle, située entre les programmes applicatifs et les couches système et matérielles est d’isoler les couches hautes (applications) de la diversité des environnements d’exécution (en plus de fournir des services distribués).

Si cette solution a fait ses preuves par le passé, la façon dont elle est mise en œuvre en général montre ses limites dans le contexte actuel ; la diversité et la complexité des environnements d’exécution ne peuvent pas, et ne doivent pas être cachées par des abstractions trop générales, forcément réductrices. De plus, il ne s’agit que d’une solution partielle qui se contente de déplacer le problème de l’adaptabilité dans une nouvelle couche logicielle sans le résoudre directement. Cependant, le middleware fournit un contexte idéal pour s’attaquer une fois pour toute au problème de l’adaptabilité : si l’on arrive à rendre le middleware adaptable, toutes les applications construites au-dessus bénéficieront de ces améliorations à moindre coût.

¹On utilisera le terme anglais *middleware* dans le reste de ce document, les différentes traductions françaises étant beaucoup moins répandues.

Toutes ces raisons nous ont conduit à décider que le contexte particulier de cette étude serait celui du middleware et des services pour applications distribuées à grande échelle. Notre but est donc de définir une infrastructure permettant de rendre applications et middleware adaptables dynamiquement à des conditions d'exécution changeantes.

1.3 Solution proposée

La solution que nous proposons consiste en une infrastructure générale permettant l'écriture et l'exécution de programmes adaptables.

L'approche retenue pour permettre ces adaptations est basée sur les techniques de réflexion [Maes, 1987], et en particulier sur la notion de protocole à métaobjets [Kiczales et al., 1991]. Un premier ensemble de travaux a démontré les apports de la réflexion pour l'implémentation de systèmes adaptables en général ([Dowling et al., 1999]) et en particulier au niveau du middleware ([Blair and Coulson, 1997], [Ledoux, 1999], [Boyer and Charra, 2000]). Les techniques réflexives permettent de modifier la façon dont est exécuté un programme de façon non intrusive, en modifiant les mécanismes d'interprétation plutôt que le code lui-même. De plus, ces techniques se prêtent bien à des modifications dynamiques, pendant l'exécution des programmes.

Ces modifications se font en réponse à des changements dans le contexte d'exécution. Notre infrastructure inclue donc un système d'*observation* des ressources (physiques et logicielles) constituant l'environnement du middleware et des programmes. Ce système est capable de détecter les modifications significatives dans l'environnement d'exécution, afin de déclencher une réaction.

Enfin, pour faire le lien entre ces deux fonctionnalités, nous introduisons la notion de *politiques d'adaptation*. Ces politiques sont utilisées pour configurer notre infrastructure générale en lui indiquant d'une part quelles sont les modifications de l'environnement à prendre en compte pour une application donnée, et d'autre part comment réagir à ces modifications de façon appropriée. Ces politiques seront interprétées par un *moteur d'adaptation*, qui encapsule les mécanismes généraux d'adaptation ; les politiques sont injectées dans ce moteur pour le configurer en fonction de l'application à exécuter.

1.4 Organisation du rapport

Le chapitre 2 présente un état de l'art des recherches concernant les middlewares adaptables ainsi que quelques protocoles à métaobjets pour Java. Le chapitre 3 propose une analyse plus poussée de la problématique de l'adaptation des systèmes informatiques, tout en restant relativement générale et abstraite. Le chapitre 4 présente le détail de la solution proposée dont l'implémentation actuelle est décrite au chapitre 5. Enfin, le chapitre 6 propose pour conclure une discussion concernant les apports de notre solution par rapport à l'existant, ainsi que ses limites actuelles et les directions envisagées pour les travaux futurs.

Chapitre 2

État de l'art

Ce chapitre présente un certain nombre de travaux de recherche concernant à la fois les protocoles à métaobjets pour Java et le middleware adaptable.

2.1 Réflexion et protocoles à métaobjets

2.1.1 Introduction à la réflexion

Un système réflexif est un système qui possède une auto-représentation décrivant la connaissance qu'il a de lui-même. Un tel système peut alors répondre à des questions le concernant (capacité d'*introspection*), mais aussi s'auto-modifier (capacité d'*intercession*). Il a toujours une représentation précise de lui-même et son comportement est en accord avec cette représentation (principe de la connexion causale, [Maes, 1987]). Les diverses caractéristiques des programmes sont rendues concrètes (i.e. réifiés) et peuvent être manipulées par des programmes d'un niveau d'abstraction plus élevé, les méta-programmes. Dans un système possédant une architecture réflexive, on distingue le *niveau de base*, concerné par le domaine d'application, du *niveau méta*, dont le sujet est la représentation et le contrôle du niveau de base¹.

Grâce à l'abstraction des données (encapsulation) et à leur organisation (héritage, composition), les langages à objets offrent un cadre privilégié à la mise en œuvre d'architectures réflexives. Les objets du niveau méta, appelés *métaobjets*, décrivent la représentation et contrôlent le comportement des objets du niveau de base. Les protocoles à métaobjets (*Meta Object Protocol* ou MOP) règlent la communication entre les objets et les métaobjets [Kiczales et al., 1991], et constituent donc l'interface entre les différents niveaux. Il est alors possible de spécialiser les métaobjets fournis en standard pour introduire de nouvelles sémantiques de représentation et d'exécution des objets (concurrency, localisation des objets répartis [Okamura and Ishikawa, 1994], envoi de messages distants [McAffer, 1995]).

2.1.2 Guaranà

Guaranà est une architecture réflexive implémentée sous la forme d'un protocole de métaobjets pour Java [Oliva et al., 1998].

¹Le niveau méta étant lui-même un programme, il est possible d'introduire un *méta méta niveau*, et ainsi de suite à l'infini (conceptuellement). Une telle architecture, constituée de plusieurs couches (étages) de plus en plus abstraits, est nommée *tour réflexive*.

Description

L'architecture de Guaranà est décomposée en trois parties :

- le noyau (*kernel*) ;
- les métaobjets ;
- et les compositeurs (*composers*).

Le noyau est chargé d'implémenter les mécanismes de base nécessaires à Guaranà. En fonction de la plate-forme choisie (par exemple Java), celui-ci peut être implémenté de façons différentes. Les mécanismes de base sont :

- l'interception et la réification des opérations (envoi de message par exemple) ;
- la gestion du lien méta et redirection des opérations réifiées vers les métaobjets appropriés ;
- et la maintenance des informations concernant la structure du niveau méta.

Les métaobjets sont responsables de l'implémentation de certaines parties du comportement réflexif du système. Chaque objet de base peut être associé (à travers un lien méta) à un métaobjet optionnel. Ce métaobjet est appelé le *métaobjet primaire* de l'objet de base. Le mécanisme d'interception implémenté par le noyau lui permet d'observer les opérations effectuées sur l'objet de base. Une particularité de Guaranà par rapport à la plupart des autres MOPs est que le lien méta est dynamique. Les objets de base n'ont aucun moyen de connaître leur métaobjet primaire ; le lien méta est résolu dynamiquement par le noyau, qui est le seul à connaître les associations objets de base / métaobjets primaires (qui sont modifiables en utilisant la méthode **reconfigure** du noyau).

Un métaobjet primaire est responsable d'interpréter les différentes opérations invoquées sur son objet de base. Pour effectuer cette interprétation, il a trois possibilités :

- renvoyer directement un résultat. Celui-ci sera interprété comme étant le résultat final de l'opération.
- désigner une opération alternative. Le noyau annule alors l'opération en cours et traite la nouvelle comme si elle avait été envoyée directement depuis le niveau de base (et donc la redirige à son tour vers le métaobjet primaire).
- ne rien faire. L'opération est alors appliquée normalement à l'objet de base.

Lorsque le traitement d'une opération est terminé, le noyau présente le résultat au métaobjet primaire (si celui-ci l'a préalablement demandé). Le métaobjet primaire peut alors effectuer n'importe quelle opération, y compris modifier ou remplacer le résultat de l'opération avant qu'il ne soit renvoyé au niveau de base.

Pour permettre l'implémentation de comportements sophistiqués au niveau méta de façon relativement simple, Guaranà introduit un type particulier de métaobjets : les compositeurs. Ces métaobjets sont responsables d'organiser un groupe d'autre métaobjets et de les faire collaborer pour traiter les opérations. On peut imaginer plusieurs types de compositeurs, mais le plus simple est sans doute le compositeur séquentiel, qui redirige les opérations qu'il reçoit de façon séquentielle à tous ses composants et permet ainsi de grouper de façon simple plusieurs traitements de niveau méta. Les compositeurs étant des métaobjets comme les autres, ils peuvent à leur tour être composés, ce qui permet de créer simplement des configurations complexes.

Au lancement d'une application, aucun des objets de base ne possède de métaobjet primaire. La configuration du niveau méta doit se faire explicitement en utilisant la méthode **reconfigure** du noyau.

Évaluation

Guaranà est un MOP *run-time* qui dispose de toutes les fonctionnalités nécessaires pour modifier dynamiquement le comportement des objets de base. Il dispose aussi d'une bibliothèque de métaobjets pour applications distribuées (MOLDS). Cependant, Guaranà est implémenté en utilisant une version modifiée de la machine virtuelle Kaffe OpenVM, ce qui est incompatible avec les objectifs que nous nous sommes fixé (notre implémentation doit tourner sur la plateforme Java standard, pour des raisons de compatibilité et portabilité maximale).

2.1.3 ProActive

ProActive (anciennement Java//) a pour objectif de simplifier la programmation concurrente en Java en fournissant des abstractions de haut niveau [Caromel et al., 1998]. L'approche vise à réutiliser du code séquentiel à travers une bibliothèque ouverte pour construire de façon transparente des applications multithreadées parallèles ou distribuées. Pour atteindre ce but, ProActive base son approche sur un MOP Java. ProActive est basé sur Java RMI et ne nécessite aucun pré-compilateur particulier, ni aucun changement de la machine virtuelle Java. La bibliothèque de ProActive permet de modéliser différents modèles pour les objets (objets actifs, objets distribués, agents mobiles), différentes politiques de communication et synchronisation (appels asynchrones, objets futurs).

Description

Le MOP de ProActive permet de contrôler l'invocation de méthode sur un objet Java. Pour arriver à ce résultat, sans modification de la machine virtuelle, ProActive propose un protocole de création particulier et une génération dynamique d'encapsulateurs (appelés stubs). Ce protocole de création doit être utilisé en lieu et place du `new` Java afin de générer automatiquement les encapsulateurs qui permettront d'intercepter les messages reçus par l'objet. Ces messages sont alors délégués à un méta-objet appelé proxy, chargé de réaliser le traitement méta. Soit une classe `A`. La création d'une instance de `A` via ce protocole de création va provoquer :

- la génération de la classe `Stub_A` (le stub), sous-classe de `A`, qui redéfinira toutes les méthodes de `A` et celles dont elle hérite afin de réifier les appels de méthodes et les transmettre au méta-objet.
- la création d'une instance de `Stub_A` et d'une instance d'une classe proxy (métaobjet). La création proprement dite de l'instance de `A` est laissée à la charge du méta-objet (cf. ci-dessous).
- le retour de l'instance de `Stub_A` à la place de celle de `A`.

L'exemple suivant montre le code de la classe `EchoProxy` qui représente la classe des méta-objets réalisant une trace. Elle implémente l'interface `Proxy` et fournit donc une implémentation de la méthode `reify(aMethodCall)`, seule méthode du MOP. Cette méthode prend en argument l'appel réifié, construit par le stub, et doit renvoyer la valeur retournée par l'exécution de cette méthode. Notons que l'appel au constructeur est lui aussi réifié et peut être ainsi modifié.

```
public class EchoProxy implements Proxy {
    // Attributs
    Object myobject;
    // Constructeur
    public EchoProxy(MethodCall c, Object[] p) {
```

```

        this.myobject = c.execute();
    }
    // Méthode de l'interface Proxy
    public Object reify(MethodCall c) throws InvocationTargetException,
                                   IllegalAccessException {
        System.out.println("Echo > "+c.methodname);
        return result = c.execute(myobject);
    }
}

```

La création d'un objet se fait par l'une des méthodes statiques d'une classe particulière de la bibliothèque de ProActive. C'est à ce moment que l'on spécifie — indirectement — la classe du proxy qui sera utilisé pour cette instance particulière, en indiquant le nom d'une interface Java (ici Echo).

```

A a = (A) MOP.newInstanceMeta("A", "Echo", null, null);
a.foo(); // affiche Echo -> foo()

```

L'interface Echo étend l'interface `Reflect`, racine des interfaces des méta-objets de ProActive. Echo a pour unique rôle de désigner le nom de la classe du proxy (méta-objet) :

```

public interface Echo extends Reflect {
    PROXY_CLASS = "EchoProxy";
}

```

Évaluation

Les points intéressants de ProActive sont son aspect dynamique et surtout sa portabilité, puisqu'il fonctionne dans un environnement Java standard. Sa caractéristique qui le distingue le plus des autres MOPs étudiés est la possibilité de faire cohabiter dans une application donnée des instances réflexives et des instances non réflexives d'une même classe. Cela permet de ne payer le prix de la réflexion (en terme de performances) que pour les objets qui en ont vraiment besoin.

Du côté négatif, ProActive utilise un protocole de création d'objets non standard, qui nécessite de modifier le code ou en tout cas de l'écrire d'une façon inhabituelle (remplacer les `new` par `MOP.newInstanceMeta()`). De plus, le MOP lui-même est relativement limité, ne réifiant que l'invocation de méthode. Enfin, le métaobjet attaché à un objet de base ne peut pas être modifié dynamiquement, et ProActive ne supporte pas non plus directement la composition de métaobjets.

2.1.4 Iguana/J

Iguana [Gowing and Cahill, 1996] est un langage de programmation réflexif basé sur C++ . Iguana/J [Redmond and Cahill, 2000] est une implémentation en Java du modèle réflexif sous-jacent.

Description

Iguana/J définit un ensemble de catégories de réification, qui correspondent aux divers mécanismes du langage qui peuvent être pris en charge par des métaobjets. Ces catégories sont séparées en deux groupes suivant qu'elles correspondent à des mécanismes structuraux ou comportementaux (voir tableau 2.1).

Mécanismes structuraux	Mécanismes comportementaux
class	object creation
attribute	object deletion
method	method send
constructor	method dispatch
array	method execute
activation frame	state read
interface	state write
	monitor entry
	monitor exit

TAB. 2.1 – Les catégories de réification proposées par Iguana/J

Ces catégories ne correspondent pas exactement à celles définies dans la première version d'Iguana, basée sur C++. En effet, elles ont dû être adaptées aux particularités de Java et en particulier à la présence de l'API de réflexion déjà présente dans le package `java.lang.reflect` de Java. La catégorie *interface* est aussi spécifique à Iguana/J puisqu'elle réifie le concept d'interface propre à Java.

À chacune de ces catégories correspond une classe prédéfinie dans Iguana/J, que le méta-programmeur peut spécialiser pour modifier les mécanismes d'exécution correspondants. Ce travail de spécialisation se fait tout simplement en sous-classant la classe définie par Iguana/J.

Plusieurs de ces métaobjets, spécialisés ou non, peuvent être regroupés en *protocoles*. Ces protocoles sont définis dans un fichier séparé en utilisant une syntaxe spéciale (cf figure 2.1). Un compilateur fourni avec Iguana/J convertit ce fichier en classe Java. Un protocole ne peut bien sûr pas spécifier plusieurs métaobjets pour une catégorie donnée, mais n'est pas obligé de spécialiser toutes les catégories (le comportement par défaut est alors utilisé). En fait, seules les catégories comportementales ont besoin d'être spécifiées dans la définition du protocole ; en effet, contrairement à ce qui se passe pour C++, l'API de réflexion standard de Java permet d'avoir accès aux catégories structurelles automatiquement.

```
protocol VerboseProtocol {
    reify Execution: VerboseExecution;
    reify StateWrite: VerboseStateWrite;
}
```

FIG. 2.1 – Exemple de définition de protocole Iguana/J

L'association d'un protocole à une classe Java se fait statiquement, en la déclarant dans un fichier spécial (cf exemple 2.2). Iguana/J utilise ce fichier pour détecter le chargement par la machine virtuelle Java des classes concernées. Une machine virtuelle légèrement modifiée est utilisée pour implémenter l'indirection nécessaire à la prise en charge des mécanismes spécialisés par Iguana/J.

L'association d'un protocole à une instance particulière peut aussi se faire, mais nécessite l'utilisation explicite dans le code de base d'une méthode statique définie par Iguana/J.

```
class ie.tcd.test.MyBaseClass ==> VerboseProtocol;
interface ie.tcd.test.MyBaseInterface ==> VerboseProtocol;
```

FIG. 2.2 – Exemple d’association de protocole dans Iguana/J

Évaluation

Iguana/J propose un MOP relativement riche, et surtout bien organisé. La répartition des différents types de mécanismes réifiables en catégories utilisables indépendamment permet de n’utiliser pour une application donnée, voire pour une classe de composants, que ceux réellement nécessaires. Cela peut avoir des répercussions importantes sur les performances par rapport à un système qui réifie automatiquement tous ces mécanismes. Cependant, les associations entre objets de base et métaobjets sont définies statiquement et ne peuvent pas être modifiées à l’exécution, ce qui en limite fortement l’intérêt pour implémenter des composants adaptables.

2.1.5 Comet

Comet est une architecture de composants conçue spécifiquement pour permettre la réalisation d’applications réparties [Peschanski, 2000].

Description

Comet définit un modèle de composants basé les notions de composants, connecteurs et connexions.

Les composants, typés, correspondent à l’unité de répartition. Ils sont similaires à des objets, mais ont potentiellement une granularité plus importante. Leur interface est constituée d’un ensemble de méthodes et de descripteurs d’événements entrant et sortants (eux aussi typés). Ils disposent aussi d’un état interne et de ressources associées.

Les composants du niveau de base sont contrôlés par un niveau *méta-comportement* décomposé en plusieurs fonctionnalités représentant les différentes étapes de traitement des messages par un composant. Dans le cas des messages entrants, il s’agit de : réception, mise en file d’attente, extraction de la file d’attente, exécution. Dans le cas des messages sortant : mise en file d’attente, extraction de la file d’attente et envoi du message. Chacune de ces fonctionnalités est réifiée par un métaobjet modifiable individuellement à l’exécution². Cette couche permet donc de modifier les mécanismes d’exécution des composants de base, mais n’est pas suffisante pour réaliser des adaptations sophistiquées.

Pour cela, un deuxième mécanisme est introduit, qui permet d’associer aux composants de base des *méta-rôles*. Ces rôles, implémentés par des méta-composants, sont différents des métaobjets décrits précédemment en ce qu’ils peuvent contenir un état interne (données) ainsi que des intercepteurs d’événements permettant de surveiller et de modifier les événements reçus et envoyés par les composants de base. Lorsqu’un rôle est affecté dynamiquement à un composant de base, les deux partis (rôle et composant) ont la possibilité de valider cette affectation afin d’éviter les conflits et d’assurer la cohérence du système.

²Ce type de MOP à granularité fine est inspiré de CodA [McAffer, 1995].

Enfin, dernière notion importante, les rôles sont regroupés au sein de *protocoles*. En effet, pour avoir un sens, l'affectation d'un rôle à un composant de base doit souvent s'accompagner d'affectations des rôles similaires ou complémentaires à d'autres composants. Un protocole correspond donc à un ensemble de rôles qui fonctionnent de concert et collaborent pour implémenter une fonctionnalité. Par exemple, pour implémenter un protocole de réplification de composants, on doit affecter un rôle "répliqué" à un composant et des rôles "réplique" correspondants à d'autres. La notion de protocole permet en fait de combler un des manques de Comet par rapport à d'autres modèles de composants, à savoir que Comet ne réifie pas les connexions entre composants.

Évaluation

Comet permet d'effectuer des modifications dynamiques assez sophistiquées, mais celles-ci doivent être programmées explicitement. Aucun support n'est fourni pour rendre ces modifications dépendantes des conditions environnementales.

2.1.6 Kava

Kava est un MOP Java implémenté grâce à des techniques de modification de *bytecode* [Welch and Stroud, 2000].

Description

Le MOP de Kava réifie les aspects suivants des programmes de base :

- envoi de message ;
- réception de message ;
- accès aux champs d'un objet ;
- initialisation d'un objet ;
- finalisation (destruction) d'un objet ;
- création d'une instance par un objet (symétrique de l'initialisation, vue du côté de l'objet créateur).

À chaque classe de base est associée une classe de métaobjets (implémentant l'interface `Metaobject`). À l'exécution, chaque instance de la classe de base sera associée à une instance de la classe de métaobjet (lien méta d'arité 1-1). Lorsqu'une des opérations citées est invoquée sur un objet de base, son métaobjet prend le contrôle et peut effectuer n'importe quel traitement avant et après l'invoque. Il peut aussi spécifier si l'opération doit effectivement être invoquée ou non.

L'association entre classes de base et classes de métaobjet se fait dans un fichier séparé grâce à une syntaxe spéciale (voire la figure 2.3 pour un exemple). Celle-ci permet un contrôle fin de l'association. D'une part, il est possible de n'utiliser que certaines de fonctionnalités d'une classe de métaobjets dans une association. D'autre part, il est possible de ne modifier dans la classe de base que ce qui est nécessaire : si par exemple on veut tracer uniquement certains appels de méthodes dans une classe de base, il est possible d'y associer un métaobjet de trace mais de ne le lier qu'aux méthodes en question.

La transformation nécessaire pour rendre réflexives les classes de base écrites en Java standard est effectué directement sur le *bytecode*. Cela a le double avantage d'être portable puisqu'on n'utilise pas de machine virtuelle modifiée, et de ne pas nécessiter le code source, ce qui est une

```
metaclass kava.MetaTrace
{
  INTERCEPT SEND_METHOD (any-class, "notify", any-desc);
  INTERCEPT SEND_METHOD (any-class, "setObserver", any-desc);
}

class Test metaclass-is kava.MetaTrace;
```

FIG. 2.3 – Exemple de script de configuration pour Kava

contrainte parfois trop forte dans le monde réel. La transformation de bytecode se fait grâce au framework JOIE.

Évaluation

Tout comme pour Iguana/J, le MOP de Kava, bien que suffisamment puissant pour réaliser le type de modifications qui nous intéressent, est implémenté de façon trop statique pour permettre une adaptation réellement dynamique.

2.1.7 RAM

RAM (Reflection for Adaptable Mobility) est un projet réalisé à l'École des Mines de Nantes pour France Télécom R&D, ayant pour but de faciliter la création d'application adaptables prenant en compte la mobilité de code. Son implémentation utilise des techniques réflexives, et est basé sur un MOP décrit dans [Bouraqaadi-Saâdani et al., 2000].

Description

Dans le MOP de RAM, les métaobjets existent explicitement à l'exécution (MOP run-time). Les objets réflexifs du niveau de base sont associés à un métaobjet par un lien méta qu'il est possible de modifier dynamiquement. Lors de la création d'un nouvel objet, RAM lui associe automatiquement un métaobjet, dont la classe est déterminée par une configuration préalable. Cette configuration consiste tout simplement à associer une classe de métaobjets à une classe de base. Il est possible de modifier cette association en cours d'exécution ; la modification n'a alors d'effet que pour les objets instanciés ultérieurement.

Un certain nombre d'opérations effectuées sur les objets du niveau de base sont automatiquement réifiées par RAM et redirigés vers le métaobjet correspondant. Ce métaobjet est chargé d'interpréter cette opération, avant de renvoyer le résultat au niveau de base. Les opérations réifiées sont : initialisation, création d'un autre objet (symétrique de l'initialisation), réception d'un message et réception d'une demande de sérialisation ou de désérialisation³. Au niveau méta, le métaobjet a bien entendu la possibilité d'invoquer ces opérations avec leur sémantique standard. C'est d'ailleurs ce que se contente de faire la classe de métaobjet utilisée par défaut, `MetaObject`.

À la base, RAM ne permet d'associer à un objet de base qu'un seul métaobjet. On peut cependant avoir parfois envie de bénéficier de la sémantique de deux classes de métaobjets différents, sans pour cela avoir à créer une nouvelle classe. RAM fournit pour cela un framework simple de

³La façon très particulière dont ces deux dernières opérations sont traitées par la machine virtuelle Java ne permet pas de les gérer de la même façon que des invocations de méthodes standards.

composition de métaobjets, basé sur le modèle de conception *Composite* [Gamma et al., 1995] (cf figure 2.4).

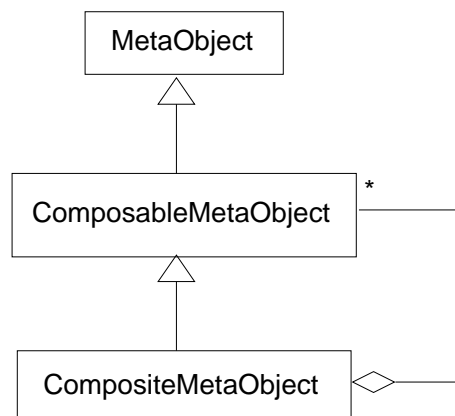


FIG. 2.4 – Composition de métaobjets dans RAM

Une instance de la classe `CompositeMetaObject` est capable de gérer une collection de métaobjets (instances de `ComposableMetaObject`), et de composer leurs comportements. Lorsqu'une opération réifiée est envoyée à un métaobjet composite, il redirige successivement cette opération à chacun de ces composants, réalisant ainsi une composition de leurs différents comportements. La classe `CompositeMetaObject` étant elle-même composable (en tant que sous-classe de `ComposableMetaObject`), il est possible d'appliquer ce schéma récursivement. Le seul type de composition de métaobjets fourni en standard dans RAM est une simple composition linéaire, dans laquelle les composants sont invoqués successivement, toujours dans le même ordre. Il est relativement simple d'étendre ce framework en introduisant d'autres types de composition.

Les classes du niveau de base sont écrites en Java complètement standard, et subissent une transformation lors du chargement par la machine virtuelle. Cette transformation effectuée grâce à la bibliothèque `Javassist` [Chiba, 2000]⁴, introduit dans le bytecode original d'une part le lien méta (sous la forme d'un nouvel attribut de type `MetaObject`), et d'autre part des *hooks*. Ces hooks sont des morceaux de code redirigeant les différentes opérations gérées par RAM vers le niveau méta.

Évaluation

RAM est un MOP Java complet et surtout très souple, car presque entièrement dynamique (si l'on excepte la phase de transformation de code préliminaire). Les opérations qu'il permet de réifier sont à priori suffisantes pour nos besoins et le framework de composition de métaobjets qu'il intègre, bien que simple en lui-même, permet une ouverture vers des systèmes plus complexes.

⁴La partie *run-time* de RAM est relativement indépendante de la façon dont cette transformation est effectuée. Nous avons montré dans [David et al., 2001] qu'il était possible d'utiliser AspectJ pour obtenir un résultat équivalent.

2.2 Middlewares réflexifs et adaptables

2.2.1 Open-ORB

Open-ORB est un projet de l'équipe de Gordon S. Blair à l'Université de Lancaster visant à définir une architecture ouverte et réflexive pour le middleware ([Blair and Coulson, 1997], [Blair et al., 2000], [Andersen et al., 2000]). L'idée de base est d'utiliser des techniques réflexives pour rendre le middleware plus ouvert et adaptable, en donnant accès aux mécanismes d'exécution de la plate-forme à travers un protocole à métaobjets (MOP).

Description

- Open-ORB est basé sur un modèle de composants (RM-ODP) dont les caractéristiques sont :
- la possibilité pour un objet d'offrir plusieurs interfaces, et de déclarer des dépendances envers d'autres interfaces ;
 - le support des interfaces pour média continus (flots de données) ;
 - la gestion explicite des interactions entre composants (à travers des objets de liaison (*binding objects*) ;
 - et le support en standard d'un service de notification d'événements.

Ce modèle de base est complété par un niveau méta appelé *meta-spaces* qui associe à chaque composant quatre modèles (*meta-space models*, cf figure 2.5) :

- le méta-modèle de composition, qui permet d'accéder au graphe d'objets constituant un composant et de le modifier ;
- le méta-modèle d'encapsulation, qui donne accès à l'interface d'un composant à travers la liste des méthodes qu'il supporte et des attributs qu'il définit ;
- le méta-modèle de ressources, permettant de connaître l'utilisation des ressources système (mémoire, threads...) et dans une certaine mesure de les influencer (modification de l'algorithme d'ordonnancement des threads par exemple) ;
- et le méta-modèle d'environnement, qui représente l'environnement d'exécution de chaque interface du composant, et qui réifie des fonctionnalités comme la réception d'un message, sa sélection...

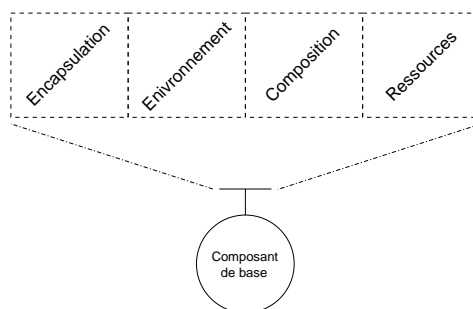


FIG. 2.5 – Structure d'un composant Open-ORB

Les éléments du *meta-space* étant implémentés avec les mêmes techniques que les objets de base, ils peuvent eux aussi avoir leur niveau méta. Cela introduit une récursion qui en théorie conduit à la construction d'une *tour réflexive* infinie. En pratique, les différents niveaux méta ne sont instanciés concrètement que s'ils sont nécessaires (instanciation paresseuse).

Open-ORB est conçu dans le but de permettre des adaptations relatives à la qualité de services (QoS, *Quality of Service*). Cette adaptation est réalisée en introduisant au niveau méta des composants spécifiquement chargés de gérer ces aspects. Pour que leur introduction se fasse de façon non-invasive, leurs interactions avec les composants de base se font uniquement par l'intermédiaire du modèle événementiel supporté en standard par le modèle de composants choisi. Ces *management components* sont ainsi notifiés automatiquement de ce qui se passe au niveau de base.

On distingue deux types de *management components* : les moniteurs et les contrôleurs. Les moniteurs ont un rôle passif, ils se contentent d'observer le système et de recueillir des informations et des statistiques. Les contrôleurs quant à eux ont un rôle plus actif. Certains d'entre eux utilisent les informations récupérées par les moniteurs pour sélectionner une stratégie d'adaptation, tandis que les autres, appelés *activateurs* sont chargés d'implémenter la stratégie choisie. Par exemple, si un moniteur détecte une dégradation brutale de la qualité d'une connexion réseau, un sélecteur pourra choisir parmi les différentes stratégies à sa disposition la plus adaptée aux circonstances, par exemple supprimer le son d'une vidéo en cours de visionnage. Cette stratégie sera elle-même implémentée par un composant activateur.

Le modèle Open-ORB est complètement dynamique, les différents composants existant à l'exécution. On peut donc ajouter, retirer ou reconfigurer ces composants à n'importe quel moment, y compris les composants du niveau méta. Cela signifie que les politiques d'adaptation elles-mêmes sont modifiables dynamiquement.

Un prototype écrit en Python (OOPP, Open-ORB Python Prototype) a été réalisé pour prouver la faisabilité de cette architecture.

Évaluation

Open-ORB est un des rares systèmes à prendre en compte tous les aspects de l'adaptabilité dynamique, de la détection des changements de l'environnement à la modification incrémentale du système.

2.2.2 Olan

Olan ([Balter et al., 1998]) est un environnement pour la création, la configuration et le déploiement d'applications distribuées, basé sur un langage de description d'architecture nommé OCL. Les quatre buts que se fixe le projet Olan sont :

- l'intégration de composants existant, écrits dans différents langages ("legacy code");
- l'encapsulation des mécanismes de communication utilisés par le middleware;
- la description architecturale des applications;
- et le déploiement d'applications.

Description

Le langage de description d'architecture d'Olan, OCL (Olan Configuration Language) permet d'intégrer des composants écrits dans différents langages en définissant un modèle commun leur permettant d'interagir. Dans ce modèle, un composant est décrit par l'ensemble des services qu'il fournit et qu'il requiert pour fonctionner. Ces services sont définis par leur signature, exprimée dans un langage spécifique, OIL (Olan Interface Language). Pour chaque langage de programmation que l'on veut utiliser dans Olan (C++, Java, Python...), on doit définir la correspondance

entre les concepts de ce langage et ceux d'Olan ⁵. À partir d'une description OIL est de quelques informations supplémentaires (localisation du code binaire du composant par exemple), il est possible de générer un "wrapper", qui encapsule un composant écrit dans ce langage et le rend utilisable par le reste du système.

Une fois les composants intégrés au système et leurs interfaces définies, reste à définir l'architecture de l'application en reliant ces composants entre eux. Pour cela, on peut construire des composites, à partir soit de composants primitifs (ceux décrits précédemment) soit, récursivement, d'autres composites. Un composite est une sorte de *Façade* pour un groupe de sous-composants : il définit la façon dont ces sous-composants communiquent entre eux et fournit une interface unifiée et de plus haut niveau permettant d'utiliser ce groupe depuis l'extérieur. Les interconnexions entre composants sont décrites par des *connecteurs*, qui spécifient le type de flot de données, de flot d'exécution, le protocole de communication et les mécanismes d'exécution. L'application elle-même est représentée par un composite, racine de cette hiérarchie.

Olan autorise un certain degré de dynamique dans l'architecture de l'application en permettant l'instanciation dynamique de composants et en introduisant la notion de *collection*, qui permet de regrouper un ensemble de composants dont le nombre peut évoluer au cours de la vie de l'application.

Enfin, OCL permet aussi de spécifier des contraintes de déploiement. Les différents noeuds du système distribué sont décrits par un ensemble de caractéristiques, appelées *management attributes*, spécifiant par exemple le type de plate-forme logicielle, l'adresse IP, ou bien encore des paramètres dynamiques tels que la charge moyenne. Une *politique de distribution* permet alors de poser des contraintes sur les caractéristiques des noeuds accueillant certains composants. On peut ainsi spécifier par exemple que tel composant doit être déployé sur un système ayant au moins une certaine quantité de mémoire.

Concrètement, toute cette description de l'architecture de l'application se fait à travers des fichiers OCL, qui seront traduits par un compilateur en classes de composants, classes de connecteurs et scripts de déploiement. Ces différents éléments sont ensuite introduits dans la *Olan Configuration Machine* qui les instancie pour produire l'application finale (cf. figure 2.6).

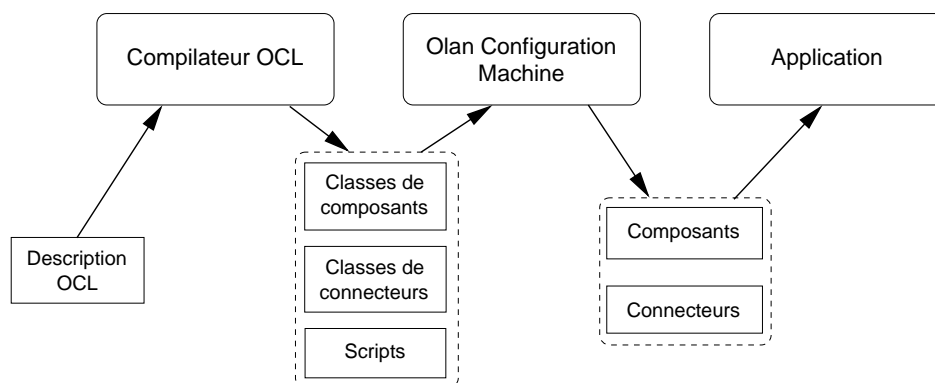


FIG. 2.6 – Utilisation du système Olan

⁵Il s'agit d'un concept similaire aux "mappings" définis par la norme CORBA pour transformer une spécification IDL en un squelette de composant concret, mais dans le sens inverse. Il s'agit ici de décrire dans un langage commun des composants hétéroclites existants.

Aspect dynamique

À la base, Olan est un système qui ne permet que très peu de modifications dynamiques de l'application. Cependant, une extension d'OCL a été développée dans [Riveill and Senart, 2001] pour permettre de décrire l'évolution dynamique d'une architecture. Ce système est basé sur un formalisme de règles actives de type (*Evenement*, *Condition*, *Action*). Lorsque l'événement spécifié dans une règle se produit, le système évalue la condition associée, et si elle est vérifiée, exécute l'action correspondante.

Les types d'événements primitifs pris en compte correspondent au cycle de vie des composants (création, destruction, clonage, déplacement) et des connecteurs (attachement, détachement, envoi et réception de messages). Ces événements primitifs peuvent être composés par des opérateurs de conjonction, disjonction et séquence pour former des événements composites.

La condition est une simple expression booléenne concernant n'importe quel élément du système (par exemple les caractéristiques d'un composant), voire du contexte d'exécution (par exemple le nombre de composants déployés sur un site donné).

Enfin, l'action décrit les traitements à réaliser lorsque la règle est activée. Cette action peut être exprimée soit en utilisant des opérations de reconfiguration inspirées des opérateurs OCL standards, soit dans un fichier externe (JavaScript par exemple) qui sera exécuté par le système et qui permet d'avoir toute la puissance d'un langage de programmation complet.

Évaluation

Si on lui rajoute l'extension décrite plus haut, Olan permet une certaine forme d'adaptation, puisqu'il est alors possible de modifier le système en réaction à des événements. Cependant, les événements pris en compte dans le modèle ne sont pas liés à des conditions d'exécution externes. De plus, les types de modifications permis sont soit trop limités (modification de l'architecture), soit trop généraux (exécution d'un script externe).

2.2.3 R-RIO

R-RIO (Reflective-Reconfigurable Interconnectable Objects, [Loques et al., 2000]) tente d'intégrer dans un cadre unifié la méta-programmation d'une part et les langages de description d'architectures et de configuration d'autre part (ADL, [Medvidovic and Taylor, 2000]). Le but est bien sûr de pouvoir bénéficier des avantages des deux approches. L'implémentation de R-RIO est faite en Java.

D'une part, la méta-programmation favorise la réutilisabilité en permettant l'écriture de services génériques. En plaçant le code fonctionnel et le code non-fonctionnel à des niveaux d'abstraction différents, elle facilite ainsi la séparation des différents aspects d'un programme, le rendant plus modulaire et flexible.

Pour leur part, les langages de description d'architecture et de configuration permettent de spécifier très précisément la configuration de systèmes à base de composants, et cela de façon indépendant du code lui-même. Ils permettent donc d'adapter la façon dont les composants fonctionnels et non-fonctionnels seront utilisés dans un environnement particulier. Ces langages étant souvent basé sur des modèles formels, ils peuvent bénéficier d'outils de vérification de validité. Cette aspect formel est rarement présent dans les approches réflexives et peut grandement faciliter l'écriture d'outils d'aide au développement et au déploiement (entre autre choses).

Description

R-RIO propose un modèle de composants basé sur les concepts de *modules*, *connecteurs* et *ports*. Il s'agit là de concepts standards que l'on retrouve dans tous les ADLs. Les modules sont utilisés pour représenter les composants fonctionnels d'un système, et correspondent au niveau de base dans les approches réflexives. Les connecteurs quand à eux réifient les interactions entre modules, et sont donc particulièrement adaptés pour représenter les points de réification dans le code de base qui activent le niveau méta. Enfin, les ports sont utilisés pour effectuer les connexions entre modules et connecteurs ; ils représentent en quelque sorte l'interface des modules, et donc des objets de base. En pratique, les ports correspondent aux signatures de méthodes de ces objets de base.

Ce modèle de composant est complété par CBabel, un langage de description d'architecture permettant de décrire la structure des interconnexions entre composants au moment du déploiement, sachant que cette structure pourra évidemment être amenée à évoluer dynamiquement pendant l'exécution du programme. Enfin, tous ces éléments sont intégrés dans une plate-forme de type middleware réflexif.

La possibilité de faire évoluer dynamiquement l'architecture permet d'adapter le comportement des composants de base. La figure 2.7 montre par exemple comment il est possible d'ajouter dynamiquement la gestion de la synchronisation des accès à un buffer. Il suffit pour cela de modifier l'architecture en insérant un connecteur entre les ports de communication reliant les différents composants. Ce connecteur générique, et donc programmé au niveau méta, est chargé de synchroniser les accès aux différents ports qu'il contrôle.

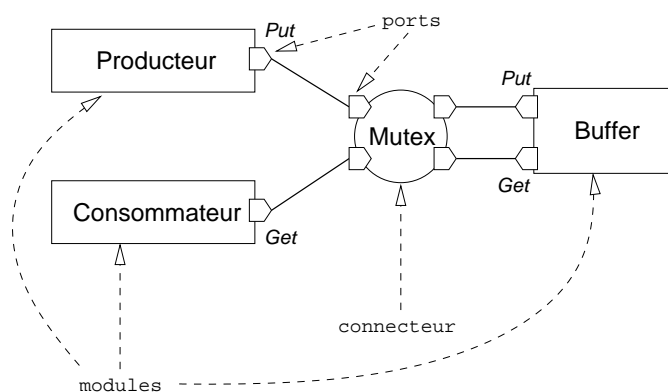


FIG. 2.7 – Adaptation de comportement dans R-RIO

Évaluation

R-RIO permet donc dans une certaine mesure d'adapter dynamiquement le comportement d'un programme en modifiant son architecture, et en particulier la sémantique des connexions entre composants. Cependant, ces modifications doivent être programmées explicitement dans des scripts de configuration, et surtout activées au bon moment. R-RIO ne propose aucun support pour déterminer quand une telle modification est nécessaire, en particulier par rapport aux conditions d'exécution.

2.2.4 LEAD++

LEAD++ [Amano and Watanabe, 1999] est un langage orienté objet basé sur Java auquel il rajoute des fonctionnalités réflexives pour permettre l'écriture de composants adaptables dynamiquement.

Description

LEAD++ est une implémentation d'un modèle plus général appelé DAS. Les éléments de base de ce modèle sont :

- les objets environnementaux (*environmental object*), qui réifient l'état de l'environnement d'exécution ;
- les objets événements (*event objects*), qui servent à notifier d'autres parties du système lorsque certains changements interviennent dans l'environnement ;
- les procédures adaptables (*adaptable procedures*), qui permettent de spécifier plusieurs implémentations d'une méthode ;
- un mécanisme d'adaptation capable de choisir parmi plusieurs implémentations d'une procédure adaptable la plus adaptée, en fonction d'une stratégie d'adaptation.

Une procédure adaptable est en fait un ensemble de méthodes, chacune étant associée à une condition environnementale (le corps de `StateCond` dans l'exemple de la figure 2.8).

```
public adaptable void slotBind(StandardEnv env) {
    StateCond {
        doc: "AC-power. So, com2 is not active.";
        type: boolean;
        body: { return env.power() == "AC-power"; }
    }
    RunCode {
        com2 = null;
    }

    StateCond {
        doc: "Battery. So, com2 is active.";
        type: boolean;
        body: { return env.power() == "Battery"; }
    }
    RunCode {
        com2 = new Com2();
    }
}
```

FIG. 2.8 – Exemple de procédure adaptable LEAD++

Le mécanisme d'adaptation est réalisé par une double indirection (*double dispatch*). Lorsqu'une procédure adaptable est invoquée au niveau de base, l'appel est redirigé au niveau méta. Un objet stratégie est alors instancié et invoqué. Cet objet stratégie est lui-même une procédure adaptable (au niveau méta), et choisit une de ses implémentations (dans ce cas, un mécanisme d'exécution parmi plusieurs possibles) en fonction de la configuration du système. Ce mécanisme est alors utilisé pour choisir au niveau de base quelle méthode doit être invoquée en fonction des

conditions environnementales qui leur sont associées.

Évaluation

LEAD++ fournit tous les éléments nécessaires pour permettre l'écriture d'applications adaptables mais les stratégies d'adaptation et les conditions déclenchant les adaptations doivent être codées de façon très explicites par le programmeur.

2.2.5 DART

DART (Distributed Adaptive Run-Time, [Raverdy and Lea, 1999]) est une plate-forme de développement dont le but est de faciliter l'écriture d'applications distribuées. Par rapport à des systèmes comme CORBA, DART va plus loin en permettant l'adaptation dynamique des applications aux conditions d'exécution. Le système utilise pour cela une combinaison de techniques réflexives et de surveillance (*monitoring*) de l'environnement d'exécution.

Description détaillée

Deux mécanismes d'adaptation sont disponibles, tous deux basés sur des techniques réflexives : les méthodes adaptatives et les méthodes réflexives (resp. *adaptive methods* et *reflective methods*).

Les méthodes adaptatives sont destinées à être utilisées au niveau de base, donc par le programmeur d'application. Chacune de ces méthodes peut disposer de plusieurs implémentations différentes, et chaque envoi de message va déclencher un processus de sélection pour déterminer laquelle de ces implémentations est la plus appropriée. Cette sélection est effectuée en fonction des conditions du moment. Ce mécanisme est assez proche d'un modèle de conception *Stratégie* [Gamma et al., 1995] dans lequel les stratégies concrètes correspondent à différentes implémentations de la méthode, excepté que le choix d'une stratégie particulière se fait dynamiquement et surtout automatiquement.

Le mécanisme des méthodes réflexives est très similaire, mais est destiné à être utilisé au niveau méta. La réception d'un message par un composant est interceptée par un *réflecteur*, chargé de le rediriger vers le ou les méta-objets appropriés. Se situant à un niveau d'abstraction plus élevé, les méthodes réflexives ont l'avantage d'être plus générales que les méthodes adaptatives (liées à une application particulière), et donc de pouvoir être réutilisés dans plusieurs applications.

La gestion de l'adaptation du système se fait grâce à des *politiques d'adaptation* associées aux composants, qui peuvent être définies soit par l'application, soit par les bibliothèques qu'elle utilise. Ces politiques sont notifiées par le système des changements dans l'environnement d'exécution ou dans les préférences utilisateur, et peuvent donc y réagir. L'ensemble de politiques d'adaptations présentes dans le système est coordonné par le DART *manager*, afin d'éviter les adaptations incompatibles ou incohérentes entre elles. Pour faciliter cette coordination, les politiques sont organisées en trois niveaux d'abstraction (*système*, *middleware* et *application*) et, à chaque niveau, groupées en modules suivant leur domaine d'application (communication, thread...). Cette organisation permet d'ordonner les politiques : d'abord suivant leur niveau d'abstraction (les politiques les plus prioritaires étant celles de plus haut niveau d'abstraction), et ensuite en affectant des priorités aux modules d'un même niveau.

La figure 2.9 résume les différents composants du système DART et leur relations.

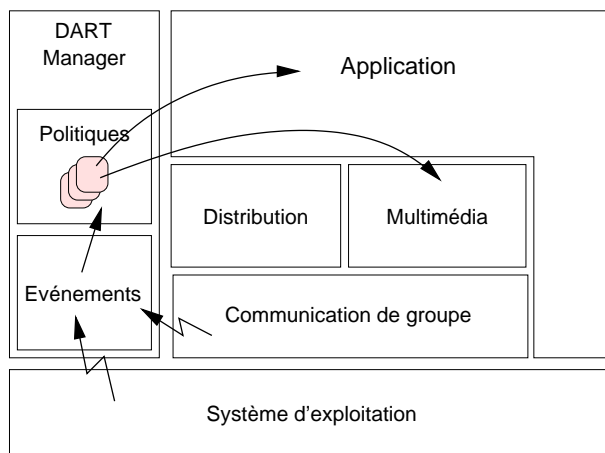


FIG. 2.9 – L'architecture de DART

Au niveau de l'implémentation, DART est basé sur une version modifiée de C++ (réalisée en utilisant OpenC++) à laquelle de nouveaux mots-clef ont été ajoutés pour représenter les deux nouveaux types de méthodes.

Évaluation

DART est un des systèmes adaptatifs les plus sophistiqués parmi ceux étudiés. Il est surtout l'un des plus complets du point de vue du support des différentes fonctionnalités nécessaires à un système adaptatif, de la détection des changements de conditions d'exécution à la modification semi-automatique du comportement de l'application. La modification n'est que semi-automatique, puisque les politiques d'adaptation chargées de réagir aux modifications des conditions d'exécution sont encore à programmer explicitement par les programmeurs (d'application ou de bibliothèque).

2.2.6 Adaptive components

Le modèle des "adaptive components", décrits dans [Boinot et al., 2000] vise à faciliter l'écriture de composants Java capables de fonctionner de façon différente en fonction du contexte d'exécution dynamique, autrement dit des composants adaptatifs.

Description

Le système proposé est une extension du langage Java (un compilateur *ad hoc* est fourni qui génère du *bytecode* Java standard) permettant d'écrire des "classes adaptatives" (*adaptation classes*).

Une telle classe se distingue d'une classe Java standard en ce qu'elle fournit plusieurs implémentations différentes d'une même interface, chacune adaptée à des conditions d'exécution particulières, et définit de façon déclarative quelle implémentation doit être utilisée dans quelles conditions d'exécution. Ces conditions d'exécution sont supposées être réifiées par d'autres composants. L'exemple suivant (figure 2.10, tiré de [Boinot et al., 2000]) présente un composant

capable de changer dynamiquement d'algorithme de compression de son (LPC ou GSM, implémentés dans d'autres classes) en fonction de la quantité de bande passante disponible (réifiée par la classe `RtcpController`). Lorsque la bande passante est en dessous de 13,3 kbit/s, le composant `SoundEncoder` utilisera l'implémentation définie dans la classe `Lpc`, et dans le cas contraire il utilisera la classe `Gsm`. Bien entendu, ce changement d'implémentation se fait dynamiquement pendant l'exécution du programme.

```
adapclass SoundEncoder adapts Encoder {
    RtcpController rtpControl;
    SoundEncoder(RtcpController _rtpControl) {
        rtpControl = _rtpControl;
    }
    when (rtpControl.bandwidth < 13.3) Lpc();
    when (rtpControl.bandwidth >= 13.3) Gsm();
}
```

FIG. 2.10 – Exemple de composant adaptatif

Il serait bien sûr possible d'obtenir le même résultat “à la main”, en implémentant une combinaison des modèles de conception *Observateur* et *Stratégie* [Gamma et al., 1995] (c'est d'ailleurs de cette façon que le compilateur implémente ces classes adaptatives en pratique), mais au prix d'un effort considérable en terme de programmation. De plus, le résultat serait difficile à comprendre et à faire évoluer, sans compter le risque beaucoup plus important d'introduire des erreurs.

Évaluation

Ce système facilite l'écriture de composants adaptables en automatisant l'écriture d'un code complexe et en proposant une représentation très simple et claire des composants adaptables. Cependant, l'écriture des différentes implémentations d'un composant, des “sondes” permettant d'observer l'environnement (`RtcpController` dans l'exemple) et le choix des conditions limites devant entraîner un changement d'implémentation restent à la charge du programmeur d'application. De plus, ce système ne permet pas d'exprimer des critères d'adaptation différents pour différentes instances d'un même classe, et les différentes adaptations se font sans coordination entre elles. Enfin, les stratégies d'adaptation se retrouvent mêlées au code de base et toute modification, même mineure (par exemple la valeur limite de la bande passante dans l'exemple), nécessite une recompilation.

2.2.7 Molène

Molène [André and Segarra, 2000] est un framework destiné à faciliter l'écriture d'applications adaptables dans le contexte des systèmes mobiles et distribués à grande échelle. Pour cela, Molène utilise des techniques réflexives pour séparer le code fonctionnel des fonctionnalités réalisant l'adaptation [Malenfant et al., 2001].

Description

Le code d'adaptation est programmé au méta-niveau dans un *Mobile Environment Management System* (MEMS), et le code fonctionnel correspond au niveau de base. Un MEMS est responsable d'adapter le comportement des composants du niveau de base en fonction de la disponibilité des ressources. Pour réaliser ceci, deux frameworks sont utilisés : le *Detection/Notification Framework* (DNF) et le *Reactive Framework* (RF).

Le DNF est chargé de la surveillance de l'environnement d'exécution (*monitoring*). Les informations qu'il obtient concernant l'environnement sont mises à la disposition du reste du système, qui peut aussi s'enregistrer auprès du DNF pour être notifié (de façon asynchrone) de l'occurrence de certains événements. Dans un environnement distribué, les DNFs des différents noeuds communiquent entre eux pour que chacun ait une vision globale du système. Dans un DNF donné, on distingue les moniteurs de bas niveau (BM, Base Monitors), qui acquièrent les données brutes (par exemple la valeur courante de la bande passant disponible), et les moniteurs de haut niveau (HLM, High-Level Monitors), qui synthétisent les informations fournies par d'autres moniteurs (BMS ou HLMS).

Le RF est la partie active de Molène. Elle permet de construire des composants adaptatifs, capables de changer d'implémentation en fonction des conditions d'exécution. Ces composants comprennent pour cela un **Controller**, à l'écoute des modifications de l'environnement (notifiées par le DNF), chargé de modifier la partie fonctionnelle du composant. Pour décider de quelles modifications effectuer et dans quelles conditions, le contrôleur est configuré par un objet de type **Adaptation Strategy**, spécifié par le concepteur de l'application. Cette stratégie d'adaptation est en fait un automate dont les différents états représentent différentes conditions d'exécution, et dont les transitions correspondent à la réaction prévue lors du passage d'un état à un autre. Il y a deux types de réactions possibles :

- reconfiguration de la partie fonctionnelle du composant (par la spécification de paramètres) ;
- changement complet de l'implémentation du composant.

Molène inclut un protocole permettant d'assurer que les modifications effectuées par le framework causent le minimum d'interférence avec le code fonctionnel. En particulier, dans le cas où l'on doit changer dynamiquement l'implémentation d'un composant, un protocole à trois phases (désactivation, transition et activation) et l'utilisation du modèle de conception *Memento* [Gamma et al., 1995] permet de s'assurer qu'aucune donnée fonctionnelle n'a été perdue.

La figure 2.11 présente la structure d'un composant Molène.

Évaluation

Le framework de surveillance du système proposé par Molène, DNF est l'un des plus complets et sophistiqués parmi ceux étudiés ici. Les types de réactions possibles sont eux moins intéressants : en agissant directement sur la partie fonctionnelle des composants adaptatifs, les seules adaptations rendues possibles sont forcément *ad hoc* et leur implémentation à la charge du programmeur d'application.

2.2.8 JavaPod

JavaPod ([Bruneton and Riveill, 2000]) est une plate-forme destinée à la création d'applications réparties à base de composants adaptables. Le but recherché est relativement proche de

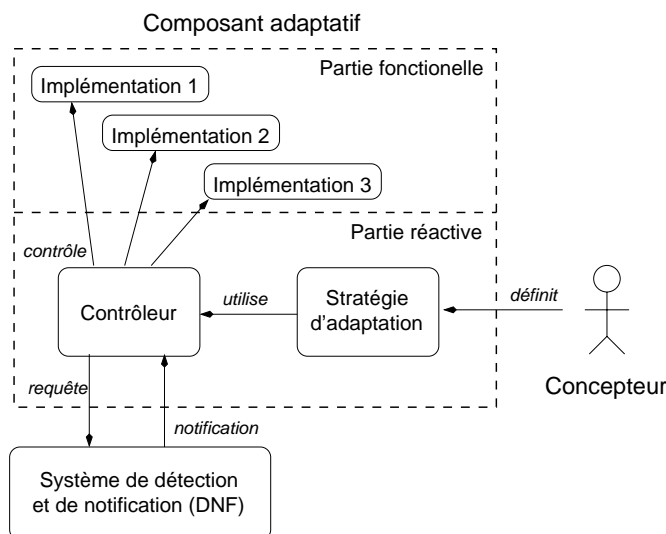


FIG. 2.11 – Structure d'un composant adaptatif dans Molène

celui des EJB (*Enterprise Java Beans*), à savoir une séparation claire au niveau architectural entre le code fonctionnel et les services non-fonctionnels. Cependant, les EJBs se limitent à un nombre fixe de services, alors que JavaPod vise à être beaucoup plus général.

Description

La plate-forme est architecturée autour de la notion de *conteneurs*, qui encapsulent les composants fonctionnels. Ces conteneurs interceptent toutes les communications d'un composant avec l'extérieur. De plus, les liaisons entre composants sont réifiées sous forme d'*objets de liaison*, constitués d'un connecteur et d'un ou plusieurs talons et squelettes. Les talons et squelettes sont utilisés pour représenter la liaison au niveau du conteneur, un talon représentant une connexion sortante et un squelette une connexion entrante. L'ensemble des talons et squelettes attachés à un conteneur constitue sa seule interface avec l'extérieur. Bien entendu, les objets de liaison sont des objets distribués, et les talons et squelettes liés à un connecteur peuvent se trouver sur des machines différentes. La figure 2.12 montre comment ces différents éléments sont organisés.

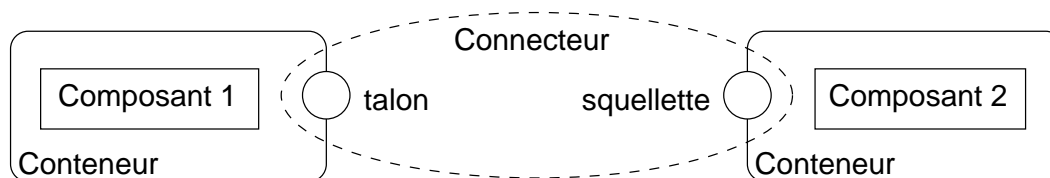


FIG. 2.12 – Composants JavaPod

Tous ces éléments sont construits en se basant sur un modèle de composition d'objets assez général : à un *objet extensible*, on peut ajouter (ou retirer) dynamiquement des *extensions*, et construire ainsi un objet composite. Les différentes extensions sont ordonnées et lors d'un envoi de message à un objet composite, c'est la méthode correspondante définie dans le constituant le "plus haut" qui est exécutée. Ajouter une extension peut donc masquer, ou surcharger des

méthodes existantes. Dans ce cas, la méthode finalement sélectionnée a la possibilité d'appeler la méthode qu'elle surcharge. L'ensemble des messages compris par un composite est donc l'union des messages compris par ses constituants.

Ce modèle permet en particulier de rendre les conteneurs et les objets de liaison modifiables dynamiquement. Ces différents éléments de JavaPod étant génériques, leurs interfaces sont proches de celles d'un MOP. Par exemple, l'interface d'un squelette ou d'un talon pourra contenir une méthode de signature `Object invoke(MethodInvocation)`. Il est alors possible d'utiliser les possibilités de composition pour ajouter des extensions à ces talons et squelettes et ainsi modifier dynamiquement la sémantique de l'appel de méthode. Plus généralement, cette technique d'interposition permet d'associer des services non-fonctionnels aux composants de base.

Évaluation

La plate-forme JavaPod permet donc de créer des composants adaptables (au moins dans leurs interactions), mais le travail consistant à déterminer quelles extensions associer à quelles liaisons et sous quelles conditions reste complètement à la charge du programmeur.

2.2.9 Lasagne

Lasagne est une architecture relativement abstraite (indépendante d'une plate-forme ou d'un langage donné) permettant la reconfiguration dynamique de systèmes à base de composants [Truyen et al., 2001].

Description

Lasagne considère les composants comme étant constitués d'un noyau inaltérable auxquels il est possible de greffer dynamiquement de nouveaux comportements. Deux types de modifications sont possibles, suivant qu'elles modifient ou non l'interface du composant. Ces deux types de modifications sont inspirées des modèles de conception *Décorateur* (modification de la sémantique de certaines opérations, [Gamma et al., 1995]) et *Rôle* (ajout de nouvelles opérations à l'interface du composant, [Bäumer et al., 2000]).

Les modifications se font en utilisant un système d'enveloppes, ou *wrappers*, autour des composants de base. Le rôle de ces wrappers (il peut y en avoir plusieurs couches pour un composant donné) est d'intercepter, et donc de contrôler toutes les interactions d'un composant de base avec l'extérieur. Ce système pose un gros problème de gestion de référence puisque envelopper un composant modifie son identité. Lasagne résout ce problème en introduisant sa propre notion d'identité de composant, indépendante des éventuelles sur-couches ajoutées dynamiquement.

Un des points forts de Lasagne est que les reconfigurations possibles des composants ne se font pas de façon indépendante mais de façon coordonnée, sous la forme d'*extensions*. Une extension correspond à un ensemble de wrappers, éventuellement de types différents, qui doivent être associés en même temps à plusieurs composants de base pour opérer une modification cohérente. Par exemple, l'ajout d'un wrapper qui implémente le rôle de proxy à un composant *A* n'a de sens que si l'on modifie un composant *B* en lui ajoutant un wrapper implémentant le rôle de serveur correspondant. Ces deux wrappers, proxy et serveur, fonctionnent ensemble et forment une extension. L'activation ou la désactivation d'une extension se fait de façon atomique du point de vue du reste du système, pour éviter les états temporaires incohérents.

L'ajout ou le retrait de ces extensions est effectué par des *politiques de composition*, définies indépendamment du code applicatif et des wrappers. Une politique de composition définit quelles extensions doivent être utilisées pour un type donné de collaboration entre composants. En pratique, cela revient à intercepter les envois de message et à décider dynamiquement quels wrappers parmi ceux installés autour d'un composant doivent effectivement être utilisés, et dans quel ordre. Cette décision se fait en fonction entre autre d'informations contextuelles. Lorsqu'un message se propage à travers les différentes couches de wrappers pour finalement atteindre le composant de base, chaque couche peut ajouter des informations contextuelles au message pour permettre aux couches plus basses de savoir comment traiter le message.

Évaluation

Les fonctionnalités proposées par Lasagne sont intéressantes et permettent en théorie l'adaptabilité dynamique, puisqu'il est possible de modifier dynamiquement le comportement des composants. Cependant, le modèle reste d'un niveau d'abstraction assez faible, et est difficile à comprendre. Le modèle d'enveloppes dynamiques choisi pose certains problèmes (identité) et les solutions apportées par Lasagne ne font qu'alourdir le modèle et le rendent encore plus complexe. Enfin, Lasagne ne permet pas directement de rendre les modifications du système dépendantes de contraintes externes (disponibilité des ressources par exemple).

2.2.10 dynamicTAO

dynamicTAO est un ORB réflexif conforme au standard CORBA développé dans le cadre du projet 2K ([Kon and Campbell, 1999], [Kon et al., 2000], [Kon et al., 2001]). Le but est de rendre les systèmes existants (*legacy systems*) capables de s'adapter à la grande diversité des environnements d'exécution.

Description

Comme son nom l'indique, dynamicTAO est une extension d'un ORB existant, TAO. TAO est un ORB libre (au sens logiciel libre) écrit en C++ et qui a la particularité d'être portable, extensible et surtout facilement configurable : les différentes parties du moteur de l'ORB sont implémentées sous la forme de modèles de conception *Stratégie*, ce qui permet de choisir parmi plusieurs implémentations la plus adaptée. Cependant, dans l'implémentation originale, cette configuration se fait au démarrage du système, en lisant un fichier de configuration.

dynamicTAO ajoute la possibilité d'effectuer cette reconfiguration à la volée, pendant l'exécution du programme. Plus précisément, les opérations permises sont :

- migration de composants vers d'autres sites du système distribué ;
- chargement et déchargement de modules du middleware pendant son exécution ;
- inspection et modification de la configuration de l'ORB.

Ces opérations sont accessibles à travers un objet `DynamicConfigurator`, qui permet de découvrir l'ensemble des alternatives possibles pour un service donné (gestion de la concurrence par exemple) et d'activer ou désactiver une implémentation particulière de ce service. Un système de gestion de dépendances sophistiqué, géré par des *configureurs de composants*, permet de conserver la cohérence du système lorsque le système est ainsi modifié.

Deux des services les plus intéressants sont le *2K Monitoring Service*, capable de recueillir des informations sur les interactions entre les composants du système, et le *2K Resource Manager*,

qui permet de connaître l'état d'utilisation des ressources physiques du système. En couplant ces deux services, on peut obtenir une connaissance très complète du système, nécessaire pour décider des modifications à effectuer et du moment où celles-ci doivent être effectuées.

Évaluation

Les modifications du système permises par dynamicTAO sont bien dépendantes des conditions d'exécutions, mais elles sont globales au système. Il n'est pas possible d'adapter séparément différentes parties du système. De plus, aucun support n'est fourni pour définir les décisions d'adaptations, qui sont à programmer explicitement.

2.2.11 Context-aware applications

L'architecture décrite dans cette section, développée à l'Université de Londres, vise à faciliter l'adaptation des applications à des environnements d'exécution variés en leur donnant accès à des informations contextuelles [Capra et al., 2001].

Description

Le modèle proposé considère les différentes couches d'un système informatique, à savoir le système d'exploitation, le middleware, les applications, et enfin les utilisateurs. Entre ces différentes couches, en plus des interfaces d'utilisation normale, on ajoute des *méta-données* (cf figure 2.13). Ces données sont destinées à décrire les différentes couches pour leur permettre de collaborer plus efficacement.

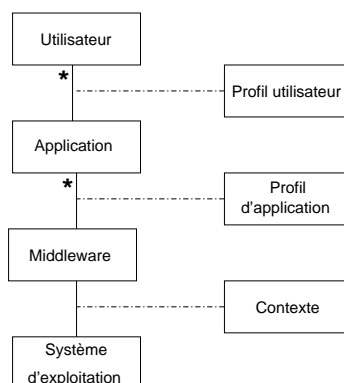


FIG. 2.13 – Profils utilisateur et d'application

Les métadonnées contextuelles correspondent à la réification par le middleware de l'état des ressources du système. Le middleware est responsable de récupérer auprès du système d'exploitation des informations concernant l'état de ces ressources (par exemple le type de connectivité disponible). Ces informations sont ensuite traduites sous forme d'un document XML (cf figure 2.14) que les applications peuvent consulter grâce à des expressions XPath.

Les applications peuvent aussi demander au middleware de réagir à certaines configurations des conditions environnementales, en définissant un *profil d'application* (*application profile*). Il s'agit d'un fichier XML qui indique au middleware comment réagir dans des conditions particulières (cf figure 2.15). Les conditions correspondent simplement à certains états des ressources

```

<CONTEXT>
  <RESOURCE name='battery-power' value='0.65' />
  <RESOURCE name='screen-size' value='1024x768' />
  <RESOURCE name='available-memory' value='32357439' />
  ...
</CONTEXT>

```

FIG. 2.14 – Fragment d'un document XML représentant les ressources du système

système. Il existe par contre deux types de réactions : modification de politique ou rappel (*callback*). Une modification de politique permet de demander au middleware de se reconfigurer, alors qu'un rappel se contente de prévenir l'application que les conditions spécifiées sont réunies, lui laissant le soin de réagir de façon appropriée. Le système fournit une API réflexive permettant aux applications de consulter et de modifier dynamiquement leur profil.

```

<RESOURCE name='battery'>
  <STATUS operator='lessEqual' value='0.25' />
  <BEHAVIOUR policy='disconnect' />
</RESOURCE>

<RESOURCE name='location'>
  <STATUS operator='equal' value='home' />
  <CALLBACK>
    <APPLICATION_REF>reference_objet</APPLICATION_REF>
    <ENTRY_POINT>methode_a_appeler</ENTRY_POINT>
    <PARAMETER>/CONTEXT/RESOURCE[@name='location']/@value</PARAMETER>
  </CALLBACK>
</RESOURCE>

```

FIG. 2.15 – Exemple de configuration XML

Lorsqu'une application invoque un service, elle peut spécifier au système quelle implémentation concrète de ce service est la plus appropriée dans quelles circonstances (cf figure 2.16). Le middleware sélectionnera lui-même la bonne implémentation en fonction de l'état actuel du système.

Évaluation

Ce système permet d'adapter de façon très fine le comportement des applications en fonction de la disponibilité des ressources. Un autre élément intéressant de cette proposition est la notion de politiques, ou profils, permettant à l'application de configurer la plate-forme. Cependant, les invocations de services doivent encore être programmées explicitement par le programmeur d'application.

```
<SERVICE name='accessData'>
  <BEHAVIOUR policy='copy'>
    <RESOURCE name='available-memory'>
      <STATUS operator='greaterEqual' value='26214400'/>
    </RESOURCE>
  </BEHAVIOUR>

  <BEHAVIOUR policy='link'>
    <RESOURCE name='bandwidth'>
      <STATUS operator='greaterEqual' value='150000' />
    </RESOURCE>
    <RESOURCE name='available-memory'>
      <STATUS operator='less value='16777216' />
    </RESOURCE>
  </BEHAVIOUR>
</SERVICE>
```

FIG. 2.16 – Configuration de l'invocation d'un service

Chapitre 3

Analyse de l'adaptabilité

Ce chapitre présente une analyse détaillée des problèmes relatifs à l'adaptabilité dynamique des systèmes informatiques à la lumière des travaux présentés dans l'état de l'art (cf. chapitre 2, page 6). Cette analyse tente de définir clairement les enjeux de l'adaptabilité dynamique et de dégager les fonctionnalités nécessaires à tout système prenant en charge cette problématique.

3.1 Définition de l'adaptabilité dynamique

Un système informatique peut être vu comme l'implémentation d'une solution à un problème donné dans un contexte particulier (cf. figure 3.1). Un tel système dépend donc à la fois des caractéristiques du problème à résoudre (cahier des charges) et de celles du contexte de l'implémentation (ressources matérielles et logicielles disponibles). Si l'un de ces deux éléments vient à changer, la solution initiale peut ne plus être *adaptée*, en ce qu'elle ne résout plus le problème, ou pas aussi bien qu'elle pourrait le faire. Il nous faut donc être capable de modifier la solution si les éléments dont elle dépend évoluent. Idéalement, cette modification devrait être minimale, localisée et incrémentale, pour minimiser son impact sur le système (cela est particulièrement important dans le cas où l'adaptation se fait dynamiquement, c'est-à-dire pendant que le système est utilisé).

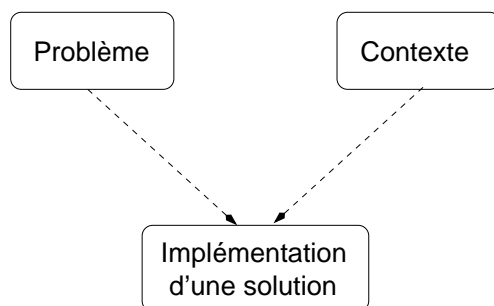


FIG. 3.1 – Problème, contexte et solution

Du point de vue de l'évolution dans le temps, la définition du problème à résoudre tend à se stabiliser au cours du cycle de développement du logiciel, même si elle n'est jamais complètement figée. À l'inverse, le contexte est lui en évolution constante et ce à diverses échelles temporelles.

D'un point de vue macroscopique, les évolutions des technologies font apparaître de nouvelles générations de machines — avec de nouvelles caractéristiques — au rythme d'une tous les quelques mois. Du point de vue microscopique, même pour une machine donnée, les ressources disponibles peuvent évoluer de façon significative à des échelles de l'ordre de la seconde (évolution de la quantité de mémoire disponible par exemple). C'est donc en priorité de ce contexte d'exécution changeant que nous devons nous préoccuper.

On comprend pourquoi construire à un instant donné un système informatique qui devra être utilisé dans un futur changeant et difficilement prédictible est aussi ardu. Traditionnellement, deux approches différentes sont utilisées pour résoudre ce problème. La première consiste à limiter la portée de la solution à des contextes très fortement contraints¹. Elle ne fonctionne que si on peut espérer avoir une bonne maîtrise de l'environnement d'exécution final, et consiste en fait à refuser tout changement, y compris ceux qui pourraient être bénéfiques. La seconde solution consiste à tenter de prédire les évolutions futures et à prévoir à l'avance la meilleure façon de réagir pour le système, ce qui est bien sûr très risqué et impossible en toute généralité.

Aucune de ces deux approches n'est satisfaisante, puisqu'elles ne font que contourner le problème. La solution est d'accepter le changement et de permettre la conception d'applications plus souples, capables de s'adapter à des situations nouvelles, qui ne seront jamais prévisibles ou maîtrisables : des *applications adaptables*. Un système est adaptable si il est capable de fonctionner dans des conditions d'exécutions non prévues au moment de sa conception. On parlera d'*adaptabilité dynamique* si le système est en plus capable de prendre en compte des modifications de son environnement pendant son exécution. L'adaptabilité suppose qu'un certain nombre de décisions d'implémentation, traditionnellement prises pendant le codage du système (et donc basées sur des suppositions quand aux conditions réelles d'exécution), doivent être reportées jusqu'au moment où les informations nécessaires sont réellement connues. De plus, ces décisions qui dépendent des conditions d'exécution doivent pouvoir être constamment remises en question si les conditions correspondantes évoluent.

Dans notre contexte particulier, *adapter* un système informatique va consister à le modifier pour qu'il fonctionne "mieux" dans des conditions d'exécutions particulières (disponibilité des ressources matérielles et logicielles par exemple). On qualifiera d'*adaptable* un système supportant ce type de modifications, c'est-à-dire, entre autres caractéristiques, architecturé de façon à être flexible. Enfin, on désignera par *système adaptatif* un système qui intègre à la fois le *sujet* de l'adaptation (le programme applicatif dans notre cas), et l'*adaptateur*, acteur de l'adaptation.

Un premier pas vers cette solution a été fait avec l'introduction du concept de *middleware*. En effet, cette couche logicielle intermédiaire entre l'application et le système d'exploitation — que l'on peut considérer comme étant une virtualisation des ressources physiques — rend l'écriture de l'application relativement indépendante des couches basses. La notion de machine virtuelle utilisée par de nombreux langages de programmation (Smalltalk, Java...) va aussi dans ce sens. Puisque cette couche logicielle regroupe tous les services indépendants de la logique d'une application particulière, le middleware est la cible idéale pour s'attaquer au problème de l'adaptabilité ; si l'on arrive à rendre le middleware adaptable, toutes les applications construites au-dessus en bénéficieront à moindre coût.

On a vu plus haut qu'adapter un système signifiait le modifier. Cependant, toute modification n'est pas une adaptation ; pour qu'elle le soit il faut que le système une fois modifié soit plus apte à assurer ses fonctions. On voit apparaître ici une idée fondamentale, à savoir qu'on ne

¹Par exemple développer un application qui ne fonctionnera que sur un PC sous Microsoft Windows 98 ayant au moins 64Mo de mémoire, 50Mo d'espace disque et un Pentium III à 500MHz ou plus.

peut pas adapter un système sans tenir compte de sa sémantique. Les couches hautes du système (application et utilisateur) sont les seules à connaître toutes les informations nécessaires pour prendre les décisions d’adaptation de façon générale.

D’un côté, on a donc le middleware, dont le but est de rendre le plus transparent possible un certain nombre de *services techniques* indépendamment d’une application particulière. De l’autre, on dit qu’une adaptation ne peut pas être réalisée sans tenir compte de la sémantique de l’application adaptée. Ces deux éléments, transparence maximale et besoin d’information, peuvent sembler contradictoires. Ils le sont en effet si l’on considère une approche de type “boîte noire”, dans laquelle les différentes couches logicielles sont rendues les plus indépendantes possibles. Une approche plus ouverte (“boîte grise”), basée sur les concepts d’*Open Implementation* ([Kiczales, 1992], [Kiczales, 1996]), permet de résoudre ce problème tout en conservant les avantages du middleware. L’idée est d’ajouter à l’interface standard entre les couches logicielles (interface de service) une seconde interface, destinée à permettre aux couches hautes de configurer l’implémentation de ces services en fonction de leurs particularités (cf. figure 3.2).

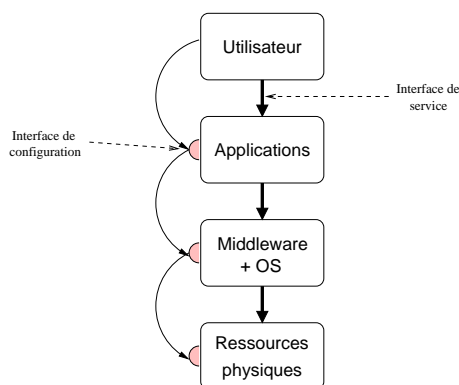


FIG. 3.2 – Les différentes couches logicielles

Puisque cette nouvelle interface doit être générique, indépendante d’une application particulière, il semble naturel de la définir à un niveau d’abstraction plus élevé et donc d’utiliser une approche réflexive. La méta-programmation permet en effet de raisonner et d’agir sur les programmes applicatifs (niveau de base), et donc de les adapter ([Okamura and Ishikawa, 1994], [Cointe and Ledoux, 2000]).

Plus concrètement, nous voulons fournir au programmeur d’application des moyens simples pour permettre l’écriture d’applications adaptables, ainsi qu’une plate-forme d’exécution capable de mettre en œuvre cette adaptation. Cette plate-forme d’exécution prendra la forme d’un middleware réflexif, capable d’adapter les applications qui l’utilisent aux conditions d’exécutions changeantes.

Comme on l’a dit plus haut, toute adaptation doit dépendre de la sémantique de l’application adaptée. On ne peut donc pas espérer construire un adaptateur “universel” en se plaçant uniquement au niveau du middleware (qui ne connaît rien des couches supérieures). Cependant, on peut espérer dégager des mécanismes généraux d’adaptabilité dans un *framework*, mécanismes qui seront configurés de façon appropriée par les couches applicatives et les utilisateurs finaux. Tout le problème revient donc à trouver la bonne nuance de gris : une interface suffisamment puissante pour permettre l’adaptabilité mais qui reste simple d’utilisation et n’expose aux couches supérieures que ce qui est nécessaire.

Les sections suivantes décrivent plus précisément les différents éléments à prendre en compte. Ces éléments correspondent en partie aux questions que se pose [Kon et al., 1998] : *What to adapt? How to adapt? When to adapt?*

3.1.1 Sujet de l'adaptation

La première chose à définir est, parmi les différents éléments constituant le système, lesquels vont être la cible de nos modifications.

On distingue dans le code constituant un programme le *code fonctionnel* du *code non fonctionnel*. Le code fonctionnel est celui qui implémente la logique de l'application, sa sémantique. Le code non fonctionnel quant à lui implémente divers *services*, utilisés pour supporter le fonctionnement de l'application. En pratique, cette séparation sera plus ou moins explicite dans le code source d'une application ; il s'agit ici d'une distinction purement conceptuelle. On peut noter cependant que, comme le montrent les approches de *Separation of Concerns* (séparation des préoccupations, [Hürsch and Lopes, 1995]) et de la programmation par aspects [Kiczales et al., 1997], il est effectivement possible de rendre cette séparation explicite, en obtenant ainsi de nombreux avantages, comme la réutilisation (sous réserve d'avoir les moyens de recombinaison des différents éléments).

Dans notre cas, cette séparation au niveau conceptuel est particulièrement intéressante. Elle nous permet en effet de distinguer deux approches très différentes du problème de l'adaptation, en fonction du sujet choisi pour les modifications.

La première approche choisit de porter ses efforts d'adaptation sur le code fonctionnel lui-même. Elle est bien illustrée par [Boinot et al., 2000] et [André and Segarra, 2000]. Le mécanisme utilisé est souvent inspiré du modèle de conception *Stratégie* [Gamma et al., 1995] : le programmeur d'application fournit plusieurs implémentations différentes de certains éléments du code fonctionnel, et spécifie dans quelles conditions ils devront être utilisés. La granularité de ces éléments peut varier ; il peut s'agir par exemple de toute une classe, ou bien simplement d'une méthode. Cette approche permet au programmeur de conserver un contrôle total sur le code exécuté, mais au prix d'un gros travail supplémentaire : il doit écrire explicitement les différentes implémentations possibles.

La seconde approche, à l'inverse, choisit de se concentrer sur le code non fonctionnel. Le rôle du code non fonctionnel est de modifier, dans une certaine mesure, la façon dont le code fonctionnel sera interprété par la plate-forme, mais sans altérer sa sémantique². On suppose ici que le code non fonctionnel est organisé sous forme de modules, implémentant des *services non fonctionnels*, en fonction du type de modifications qu'ils effectuent. Notre seconde approche va donc consister à modifier les services non fonctionnels et donc l'interprétation du code fonctionnel.

Parmi les systèmes étudiés, la plupart ont choisi cette approche : JavaPod, Open-ORB, dynamicTAO, Comet, "Context-aware applications" et DART³. Un des dangers de cette approche est de ne supporter que des modifications globales au système (c'est le cas de dynamicTAO par exemple), sans tenir compte du fait qu'un middleware est censé pouvoir faire tourner plusieurs applications, chacune étant composée de nombreux modules (de code fonctionnel). Chacune de ces composantes a ses propres spécificités, et ce qui est bon pour l'une peut ne pas l'être pour

²La limite peut être plus ou moins floue entre ce que l'on considère comme faisant partie de la sémantique d'une application et les caractéristiques purement non fonctionnelles. En outre, cette limite sera différente suivant les applications.

³DART est un peu particulier en ce qu'il supporte les deux approches.

une autre. Il doit donc être possible d'adapter de façon relativement indépendante les différentes parties du code fonctionnel, tout en gardant à l'esprit que le système global doit conserver sa cohérence.

Pour conclure sur ces deux approches possibles de l'adaptation, on peut noter que la séparation entre code fonctionnel et non fonctionnel peut aussi être vue comme la séparation de ce qui est spécifique à une application particulière et de ce qui en est indépendant. De ce point de vue là, la première approche ne peut pas être aussi bien généralisée que la seconde. Le travail qu'elle demande est à fournir de nouveau pour chaque application, et ne peut pas être factorisé. À l'inverse, la seconde approche permet de développer des services non-fonctionnels indépendamment d'une application particulière. En ayant des services non-fonctionnels réutilisables, le travail à fournir pour adapter une nouvelle application est beaucoup plus réduit.

3.1.2 Acteur de l'adaptation

Une fois décidé *ce que* nous voulons adapter, reste à savoir *qui* effectue cette adaptation (l'adaptateur). Là encore, il y a plusieurs possibilités.

La première, la plus simple, est de laisser ce problème entièrement à la charge du programmeur d'application. Il devra alors programmer explicitement les réactions de son système aux variations de l'environnement. Parmi les systèmes étudiés dans l'état de l'art, la grande majorité choisit cette approche. Le middleware se contente alors de notifier par un moyen ou un autre que les conditions ont changées, et donne la main au programmeur.

La seconde possibilité est de tenter de rendre ce mécanisme le plus automatisé possible. Les systèmes étudiés qui utilisent cette approche sont Olan (si on lui adjoint son extension dynamique), DART et Open-ORB (dans une moindre mesure pour les deux derniers). Le but final étant de minimiser le travail à effectuer par le programmeur d'application, il semble donc naturel que le middleware prenne en charge le plus gros du travail. Comme on l'a déjà dit, ce travail ne peut en général pas être fait sans la coopération de la couche applicative, qui est la seule à connaître la sémantique de l'application. On ne peut pas espérer atteindre une automatisation complète, mais on doit tenter de limiter le plus possible le travail demandé au programmeur d'application.

On peut résumer ces deux approches en parlant de systèmes *adaptables* dans le premier cas, et de systèmes *adaptatifs* dans le second. La différence est que le premier type de système est essentiellement passif ; l'acteur de l'adaptation est extérieur (le programmeur). Dans le second cas, le système est actif, et est lui même moteur de l'adaptation.

3.1.3 Moments de l'adaptation

On l'a vu, le besoin d'adaptation existe (entre autre) à cause de la nature très changeante des environnements d'exécution. On a vu aussi que ces changements s'opéraient à des échelles de temps très différentes : certains sur quelques mois, d'autres en quelques secondes (voire moins).

Conception et codage

L'adaptation ne peut évidemment pas avoir lieu au moment de la conception d'une application et de son implémentation, mais il est essentiel de "préparer le terrain" à ce moment là. Pour pouvoir permettre des adaptations ultérieures, une application doit avoir certaines caractéristiques. Elle doit être modulaire, pour permettre des modifications localisées. Un système

monolithique ne peut être modifié qu'en le remplaçant intégralement, mais dans une application construite sous forme de modules (composants), il est possible de modifier ou de remplacer certaines parties du système sans interférer avec le reste. Pour permettre l'adaptabilité dynamique, ces différents modules doivent exister explicitement à l'exécution, et pas seulement au moment du codage (on ne parle pas ici de gestion de code source) : c'est exactement ce que nous proposons les approches à objets et plus particulièrement à composants.

Dans le cas où l'on choisit d'effectuer l'adaptation par rapport aux aspects non fonctionnels du code, l'application doit en plus être écrite en séparant explicitement le code fonctionnel du code non fonctionnel. Idéalement, il ne devrait même pas être nécessaire de s'occuper du code non fonctionnel au moment de la conception, si l'on considère les techniques de séparation des préoccupations [Hürsch and Lopes, 1995]. Il est possible d'intégrer certains de ces services techniques nécessaires au bon fonctionnement de l'application, une fois l'application codée, mais avant qu'elle ne soit déployée, par exemple en utilisant une technique de tissage (*weaving*) comme le font les systèmes de programmation par aspects.

Déploiement

Au moment du déploiement, c'est-à-dire de l'installation du système sur une machine (ou un ensemble de machines), on peut effectuer une autre forme d'adaptation : *la configuration*. Une fois que l'on connaît les ressources physiques et logicielles de la machine, qui ne changent qu'à un rythme relativement lent, le degré d'incertitude qui régnait au moment de la conception et du codage peut être considérablement réduit. On peut alors configurer l'application de façon plus spécifique, quitte à perdre en généralité ce que l'on peut gagner en performances (optimisations par rapport à l'architecture cible par exemple).

Exécution

C'est à l'exécution que la partie la plus intéressante de l'adaptation a lieu : l'adaptation dynamique. Il s'agit de *reconfigurer dynamiquement* les différents éléments du système et/ou leurs associations en fonction des évolutions rapides de la disponibilité des ressources.

Cette reconfiguration se fait en tenant compte des caractéristiques particulières des différents éléments (modules) du système, tels qu'ils ont été décrits lors des étapes précédentes (codage et déploiement).

3.2 Fonctionnalités requises pour l'adaptabilité dynamique

Cette section présente rapidement les différentes fonctionnalités nécessaires à un système adaptable, en s'inspirant de l'étude effectuée dans [Efstratiou et al., 2001].

3.2.1 Observation

Le système doit bien entendu être capable d'observer l'environnement d'exécution. Cette fonctionnalité est obligatoire si l'on veut que les modifications apportées à l'application soient liées à cet environnement. Plus précisément, le système doit être capable de découvrir quelles sont les ressources disponibles, à la fois physiques et logicielles, ainsi que de détecter les variations dans la disponibilité de ces ressources. Ce sont ces variations qui seront la cause première déclenchant les adaptations.

Il ne faut pas non plus oublier que le système doit avoir une bonne connaissance de lui-même pour qu'il puisse effectuer des transformations ayant un sens tout en conservant sa cohérence.

3.2.2 Décision

Ayant une bonne connaissance à la fois de l'environnement extérieur et de lui-même, notre système doit maintenant être capable de décider quelles modifications de cet environnement ont un impact sur les applications en cours d'exécution et quelles modifications mettre en œuvre pour y répondre. Ces informations étant en règle générale dépendantes des applications, elles ne peuvent pas être prévues à l'avance.

Comme on l'a vu, certains systèmes se contentent à ce niveau de laisser la main au programmeur d'application. Cette fonctionnalité est donc réduite au minimum dans ces systèmes ; au mieux, ils sont capables de détecter des changements dans l'environnement d'exécution et d'en notifier l'application.

Si l'on veut prendre en charge cette fonctionnalité au niveau du middleware d'une façon plus automatique, le programmeur doit fournir au système certaines informations spécifiques à son application. Ces éléments de configuration du middleware seront nommés *politiques d'adaptation*. Ces politiques peuvent avoir différentes formes suivant les solutions adoptées, mais leur rôle est d'indiquer au *middleware* quelles sont les variations de l'environnement qui ont un impact sur les applications en cours d'exécution, et quelles mesures il doit prendre pour adapter ces applications aux nouvelles conditions. Évidemment, le but vers lequel on tend est d'avoir des politiques d'adaptation simples et d'un niveau d'abstraction élevé.

3.2.3 Action

Une fois que le système a détecté une modification significative de l'environnement d'exécution et qu'il a décidé que celle-ci devait entraîner une réaction de sa part (en se basant sur les politiques d'adaptation), reste à déterminer quels types de modifications du système sont possibles. Là encore, suivant le sujet d'adaptation choisi, les types d'actions possibles varient.

Dans le cas où le sujet d'adaptation est le code fonctionnel, les modifications possibles vont consister à remplacer certains éléments de ce code fonctionnel par une version alternative, plus adaptée aux nouvelles conditions détectées. Les éléments de code fonctionnel que l'on peut ainsi modifier dépendent du système ; cela peut être la classe d'un objet, remplacée par une classe alternative (les composants adaptatifs de [Boinot et al., 2000]), ou bien simplement une méthode (les méthodes adaptatives de DART [Raverdy and Lea, 1999]).

Si l'on a choisi le code non fonctionnel comme sujet de l'adaptation, les différentes actions possibles à ce niveau correspondront à la reconfiguration des associations entre composants fonctionnels et services non-fonctionnels. Par exemple, on peut modifier la sémantique de l'invocation de messages pour certains composants en la rendant asynchrone. Il doit être possible de spécifier de façon précise quels services doivent (ou ne doivent pas) être associés à tel ou tel composant. On doit aussi pouvoir configurer ces services, en spécifiant par exemple des paramètres. Ces actions ne doivent pas être globales au système, mais locales aux composants pour lesquels elles ont un sens.

Chapitre 4

Architecture proposée

Ce chapitre présente de façon détaillée l'architecture proposée pour répondre au problème de l'adaptabilité dynamique, tel qu'il a été présenté au chapitre précédent. Cette architecture est relativement indépendante des techniques de mise en œuvre ; les aspects plus techniques seront abordés en détail dans le chapitre suivant, concernant le prototype réalisé.

4.1 Vision globale

Cette section présente l'architecture proposée de façon très globale, du point de vue des différentes fonctionnalités. Le chapitre précédent a dégagé les trois grandes fonctionnalités nécessaires :

- observation (de l'environnement et du système lui-même) ;
- décision (de déclencher une adaptation) ;
- et action (modification du système).

Par rapport aux différentes possibilités présentées dans le chapitre précédent, nous avons décidé de construire un système adaptatif dans lequel le sujet de l'adaptation est le code non-fonctionnel, et capable d'effectuer cette adaptation dynamiquement (à l'exécution). C'est cette combinaison qui semble offrir le plus grand rapport puissance/facilité d'utilisation.

La figure 4.1 présente les différents éléments de notre architecture répondant à ces critères.

On considère que l'application est constituée de deux parties :

- le code fonctionnel, qui définit la sémantique de l'application,
- et les services non-fonctionnels, qui définissent dans une certaine mesure les mécanismes d'exécution du code fonctionnel.

Le code fonctionnel est en fait composé d'une multitude de composants. Notre modèle actuel ne comprend pas de modèle de composant à proprement parler, et ce que nous appelons *composants fonctionnels* ici sont de simples objets. Ce sont eux qui définissent la sémantique de l'application ; ils correspondent à ce que l'on appelle parfois des composants métiers (*business components*). Ces composants sont définis par le programmeur d'application sans se préoccuper des aspects techniques tels que la persistance, la distribution...

Les *services non fonctionnels* sont quant à eux relativement génériques et indépendants d'une application particulière. Ils implémentent un service technique particulier et peuvent être écrits par des spécialistes du domaine en question (la distribution par exemple).

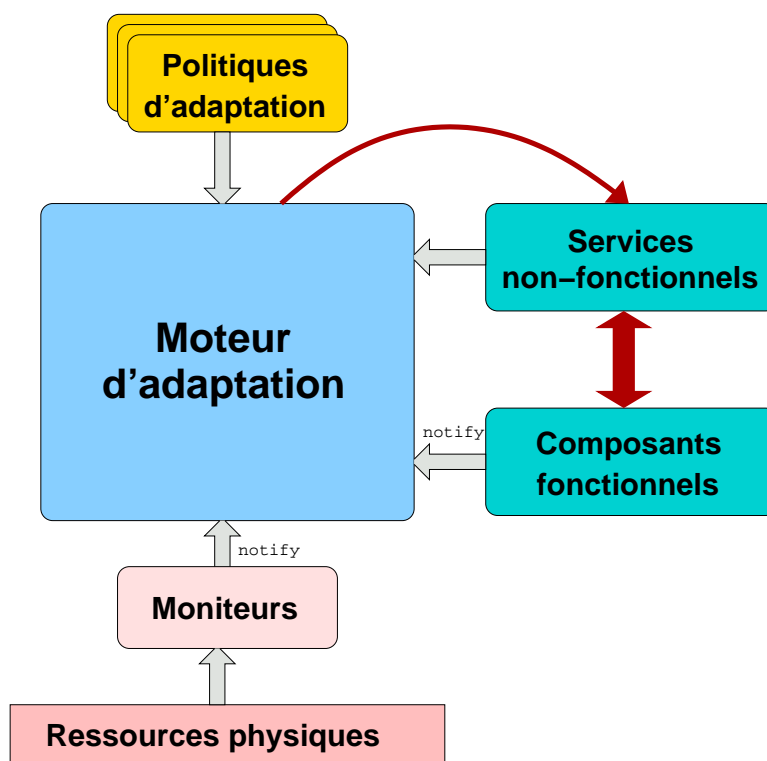


FIG. 4.1 – Décomposition fonctionnelle du système

L'adaptation elle-même sera réalisée par le *moteur d'adaptation*. Les modifications que ce composant est capable de réaliser consistent à *reconfigurer les associations entre composants fonctionnels et services non-fonctionnels*. Il doit donc être capable d'attacher, de détacher ou de reconfigurer dynamiquement des services non-fonctionnels aux composants constituant l'application, modifiant ainsi la façon dont ces composants se comportent. L'approche retenue pour permettre ces adaptations est basée sur les techniques de réflexion [Maes, 1987], et en particulier sur la notion de protocole à métaobjets [Kiczales et al., 1991]. Un premier ensemble de travaux a démontré les apports de la réflexion pour l'implémentation de middlewares adaptables ([Blair and Coulson, 1997], [Boyer and Charra, 2000]). Les techniques réflexives permettent de modifier la façon dont est exécuté un programme de façon non intrusive, en modifiant les mécanismes d'interprétation plutôt que le code lui-même. De plus, ces techniques se prêtent bien à des modifications dynamiques, pendant l'exécution des programmes. On identifiera le code fonctionnel avec le niveau de base, alors que les services non-fonctionnels, plus génériques, seront programmés au niveau méta.

Pour réaliser cette tâche, hautement dépendante de l'application à adapter, le moteur d'adaptation a besoin d'être configuré. Cette configuration est assurée par les *politiques d'adaptation*. Certaines politiques, dites *applicatives*, devront être écrites par les programmeurs des applications à adapter. D'autres, les *politiques système*, pourront l'être par les programmeurs du middleware lui-même, soit parce qu'elle gèrent des détails de trop bas niveau pour être exposées aux couches applicatives, soit parce qu'elles implémentent des adaptations relativement génériques, réutilisables par toute une catégorie d'applications. Ces politiques d'adaptation implémentent en fait

la fonctionnalité de décision en ce qu'elles indiquent au moteur d'adaptation quand et comment réagir.

Le besoin d'observation de l'environnement est quand à lui pris en charge par des *moniteurs*. Ces moniteurs sont des composants logiciels capables de communiquer avec les couches basses (matériel et système) pour en obtenir une description utilisable par le reste du système (par exemple les capacités graphiques d'un hôte, ou bien la quantité de bande passante utilisée). Les informations obtenues sont ainsi en quelque sorte réifiées¹ et rendues disponibles au reste du système. Les moniteurs doivent être capables à la fois de découvrir les différentes ressources disponibles et d'obtenir toutes les informations pertinentes les décrivant. Ces informations étant par nature dynamique, les moniteurs doivent constamment remettre à jour leurs connaissances et notifier le reste du système de tout changement significatif.

Enfin, le besoin d'observation du système lui-même est automatiquement pris en compte par les capacités d'introspection inhérentes aux systèmes réflexifs. Puisqu'on utilise un protocole à métaobjets pour gérer les composants fonctionnels et les services non-fonctionnels, ce protocole nous permet à tout moment de connaître la structure exacte du système, condition nécessaire pour effectuer des modifications cohérentes.

4.2 Action

Cette section décrit les actions rendues possibles par notre système pour modifier certains aspects du comportement des composants fonctionnels.

4.2.1 Composants fonctionnels et conteneurs

Comme nous l'avons dit, les composants fonctionnels sont en réalité des objets, mais des *objets réflexifs*. À chacun de ces composants est associé un métaobjet instance de la classe `Container`². Ce métaobjet sert de représentant pour l'objet du niveau de base du point de vue du reste du système. Pour le reste du framework, et en particulier pour le moteur d'adaptation, seuls les conteneurs sont visibles. C'est donc au conteneur de rendre disponible au reste du système les informations pertinentes qu'il possède concernant son objet de base. Le conteneur dispose pour cela d'un mécanisme décrit en détail plus loin, qui lui permet d'exporter un ensemble d'attributs représentant les propriétés du composant qu'il encapsule.

Le conteneur, instance de `Container`, est un métaobjet. Il contrôle toutes les interactions entre son objet de base et le monde extérieur (tous les autres éléments du système). Le modèle relativement abstrait présenté dans ce chapitre ne requiert aucune fonctionnalité spécifique de la part du protocole à métaobjet utilisé en pratique, excepté que celui-ci doit être de type *runtime* : les métaobjets doivent exister et être manipulables explicitement à l'exécution. On suppose seulement qu'il est "suffisamment puissant" pour permettre l'écriture des services non-fonctionnels nécessaires.

¹Il ne s'agit pas d'une réification complète, qui impliquerait une connexion causale entre les ressources et leur représentation.

²Cette classe est nommée ainsi par analogie avec les conteneurs de Enterprise Java Beans (EJB). Leurs rôles est similaires en ce qu'ils encapsulent les composants fonctionnels et leur servent d'intermédiaire dans leurs interactions avec l'extérieur.

4.2.2 Services non-fonctionnels

Les services non-fonctionnels sont eux aussi implémentés sous forme d'objets. En fait, un service non-fonctionnel est une chose relativement abstraite, identifiée par un nom. Pour un tel service abstrait, il peut exister plusieurs implémentations possibles, chacune pouvant avoir des caractéristiques différentes. On dit que ces implémentations sont des *fournisseurs* de service. Par exemple, dans le cas de la distribution, le service abstrait se nomme tout simplement **distribution**, et on peut avoir diverses implémentations utilisant RMI, CORBA ou encore SOAP. Chacun de ces fournisseurs a des caractéristiques particulières : RMI est spécifique à Java, CORBA est multi-langage mais relativement lourd, SOAP est plus léger à implémenter mais nécessite une bande passante plus importante³. Plusieurs fournisseurs différents peuvent être disponibles à un moment donné. Le choix du "meilleur" est un problème important auquel nous ne nous sommes pas encore attaché.

Certains services très simples, ne font intervenir qu'un seul composant fonctionnel. Par exemple un service de trace pourra être attaché à un composant individuel. Dès qu'un service devient un peu plus complexe, il nécessite la coopération de plusieurs composants. Par exemple dans le cas de la distribution, un composant initial que l'on veut rendre distribué va devenir serveur (RMI ou CORBA, peu importe ici) et un nouveau composant, le proxy, sera créé sur une machine distante. Pour implémenter le service de distribution, on a besoin que ces deux composants collaborent. Pour pouvoir gérer cet aspect, on introduit la notion de *rôle*. Un service abstrait donné définit un certain nombre de rôles, qui devront être utilisés de façon coordonnée pour implémenter le service en question. Dans l'exemple précédent, le service de distribution définit deux rôles : **proxy** et **server**. Activer un service va donc consister à assigner les différents rôles correspondants aux composants de base appropriés. Cette idée de rôles coordonnés implémentant un service est inspirée des rôles de Comet [Peschanski, 2000] et des extensions de Lasagne [Truyen et al., 2001].

C'est ici qu'interviennent les métaobjets. Les différents rôles définis par une implémentation particulière d'un service sont implémentés sous forme de métaobjets. En effet, ce sont les rôles qui implémentent réellement le service. En tant que métaobjets, ils ont la possibilité d'intercepter et d'interpréter toutes les interactions de leur objet de base avec l'extérieur, modifiant ainsi son comportement. La figure 4.2 présente sous la forme d'un diagramme de classes UML la structure des services.

Les métaobjets implémentant les rôles ne sont pas liés directement à l'objet de base puisque le lien méta des objets de base pointe toujours vers leur conteneur. Les rôles sont donc gérés par le conteneur lui-même, qui doit être capable d'effectuer la composition des multiples métaobjets qui peuvent lui être associés. Le modèle actuel ne dit rien à propos de cette composition, le prototype se contentant d'invoquer les différents métaobjets de façon séquentielle. Le conteneur ne permet pas d'associer à un composant deux métaobjets implémentant le même rôle du même service. Si on veut associer à un composant un métaobjet implémentant rôle qu'un autre métaobjet implémente déjà, l'ancien rôle est d'abord détaché.

Même pour une implémentation particulière d'un service donné, il peut exister différentes configurations possibles. Par exemple, dans le cas d'un service de distribution utilisant RMI, on veut pouvoir configurer l'adresse du service de nommage utilisé⁴. Lorsqu'on associe un service

³Les messages SOAP sont encodés en XML, et sont donc plus volumineux que ce que l'on peut obtenir avec un encodage binaire.

⁴Dans ce cas le nom de l'hôte et le numéro de port où trouver un serveur JNDI par exemple.

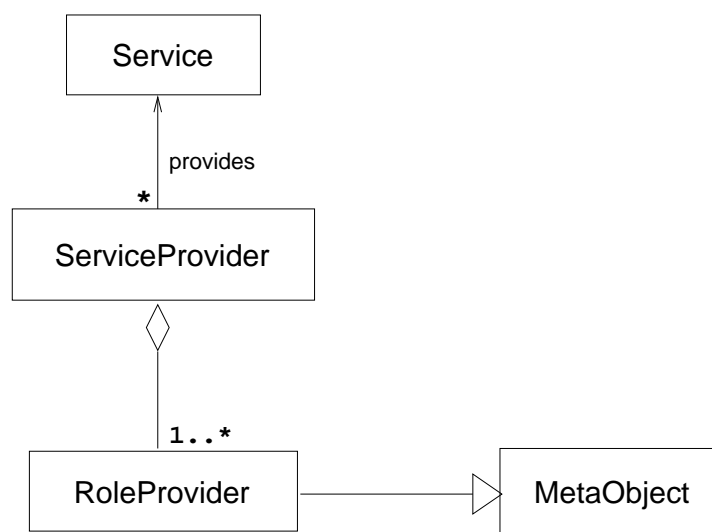


FIG. 4.2 – Structure des services

à un ou des composants, on peut donc spécifier un ensemble de paramètres. Ces paramètres sont tout simplement des couples de chaînes de caractères $\langle nom, valeur \rangle$. Le modèle actuel n'associe aucune sémantique particulière à ces valeurs et se contente de les passer en paramètres aux services, qui se chargent de leur interprétation. Des systèmes plus sophistiqués pour définir l'interface de configuration d'un service non fonctionnel existent [Robben et al., 1999]. De telles approches pourront être intégrées dans le futur.

Exemple : service de trace

La trace est un bon exemple de service très simple. Ce service consiste à imprimer un message, dans un fichier ou sur un terminal, à chaque fois qu'une opération est effectuée sur l'objet de base. Le type d'opération qui donne lieu à un message dépendra évidemment des capacités du MOP utilisé.

Ce service ne définit qu'un seul rôle (**main** par exemple), puisqu'il ne nécessite pas la coopération de plusieurs composants. Une implémentation de ce rôle sera donc constituée de deux classes :

- **TraceProvider**, sous-classe de **ServiceProvider** qui déclare implémenter le service abstrait nommé **trace**. Cette classe sera chargée d'effectuer l'installation du rôle ; son travail est très simple dans ce cas.
- **TraceRole**, sous-classe de **RoleProvider** (et donc indirectement de **MetaObject**) correspondant au rôle **main**. L'implémentation se contente d'intercepter toutes les invocations qu'elle reçoit et d'afficher un message correspondant à l'écran avant de passer la main soit à d'autres métaobjets, soit à l'implémentation normale de l'invocation.

On peut aussi avoir plusieurs implémentations, l'une se contentant de messages textuels simples et d'autres utilisant des formats plus complexes (XML par exemple).

Même dans un cas aussi simple, on peut imaginer utiliser des paramètres pour configurer par exemple la destination des messages ou bien pour spécifier que certaines méthodes ne doivent pas être tracées.

Exemple : service de distribution

Le cas de la distribution est déjà un petit peu plus complexe. Le but est de permettre à un composant existant sur une machine *A* d'être utilisé de façon transparente par le code de base exécuté sur une machine distante *B*. On doit donner l'illusion que ce composant est présent en même temps sur les deux machines.

Ce service fait intervenir deux rôles : **proxy** et **server**. Le rôle **server** sera attaché à un composant existant que l'on veut rendre accessible depuis d'autres machines. Le rôle **proxy** sera quand à lui attaché à un composant créé pour l'occasion, qui doit être du même type que le composant initial ; ce représentant distant du composant initial est uniquement une coquille vide, nécessaire pour que les composants du niveau de base pensent avoir affaire avec un véritable composant (modèle de conception *Proxy*, [Gamma et al., 1995]).

Les classes dont on a besoin pour implémenter ce service (par exemple en utilisant RMI) sont (au moins) au nombre de trois :

- **RMIProvider**, sous-classe de **ServiceProvider** qui déclare implémenter le service abstrait nommé **distribution**. Cette classe doit seulement être capable d'instancier et d'installer correctement les deux rôles **proxy** et **server** lorsqu'on le lui demande.
- **RMIProxyRole**, sous-classe de **RoleProvider** correspondant au rôle **proxy**. Les métaobjets instance de cette classe interceptent toutes les opérations invoquées sur leur composant de base et les redirigent, en utilisant des mécanismes de distribution (ici RMI) vers leur alter-ego distant qui est le seul capable d'effectuer ces opérations. Une fois la réponse reçue, le métaobjet la renvoie au niveau de base en tant que résultat de l'opération.
- **RMIUserRole**, sous-classe de **RoleProvider** correspondant au rôle **server**. Les instances de cette classes sont capables de réceptionner des invocations d'opérations envoyée à travers RMI et de les passer au niveau de base pour qu'elles soient interprétées comme des invocations normales (ce qui implique qu'elles seront redirigées vers le niveau méta). Ayant reçu le résultat de l'opération, le métaobjet renvoie celui-ci vers le proxy.

Les paramètres de configuration d'un tel service pourront être utilisés par exemple pour spécifier l'adresse d'un serveur JNDI à utiliser pour "publier" le serveur.

On peut évidemment imaginer d'autres implémentations de ce même service, utilisant par exemple CORBA, mais la structure générale et les rôles restent les mêmes.

4.2.3 Association dynamique de services

Pendant la vie du composant, l'ensemble des services (en fait des rôles qui lui sont associés) évolue. Étant donné que les rôles sont implémentés par des métaobjets, et que plusieurs rôles peuvent être affectés à un composant en même temps, cela implique que le conteneur (et donc la classe **Container**) soit capable de composer dynamiquement des métaobjets. En pratique, le modèle actuel se contente de composer les différents rôles de façon séquentielle : lorsqu'une opération est invoquée sur un composant de base, le conteneur passe cette invocation aux différents rôles actifs à ce moment précis, les uns à la suite des autres. Quand il reçoit une invocation d'opération chaque rôle peut :

- modifier les données relatives à l'opération demandée (par exemple les paramètres d'une invocation de méthode),
- effectuer au traitement quelconque,
- effectuer l'invocation elle-même (avec la sémantique normale du langage),

-
- invoquer une autre opération (de la même manière que le ferait un composant du niveau de base),
 - demander la poursuite du traitement au niveau méta (les paramètres de l’invocation ayant éventuellement été modifiés).

Attachement et détachement d’un rôle

Lorsqu’un rôle (et donc un métaobjet) est attaché à un composant, le conteneur de ce composant vérifie d’abord qu’il ne possède pas déjà un métaobjet implémentant ce rôle particulier (en tenant compte du service). Si c’est le cas, le métaobjet existant est retiré avant d’installer le nouveau.

Lorsqu’un conteneur ajoute ou retire un rôle, il prévient le métaobjet correspondant (en lui envoyant un message approprié) pour que celui-ci puisse s’initialiser ou libérer des ressources. Dans le cas où un métaobjet en remplace un autre, le conteneur indique aussi au métaobjet ajouté (resp. retiré) quel est celui qu’il remplace (resp. qui le remplace); cela permet aux deux métaobjets de communiquer et d’échanger des données pour permettre une transition plus transparente⁵.

Reconfiguration de service

Lorsqu’un rôle est attaché à un composant, on peut lui passer des paramètres de configuration. Ces paramètres sont simplement de chaînes de caractères, et n’ont aucune signification pour le conteneur. C’est aux rôles de les interpréter correctement. Il est aussi possible de modifier ces paramètres à n’importe quel moment, et ainsi de reconfigurer dynamiquement un rôle et donc un service.

4.3 Observation

Le système propose un modèle simple et homogène pour l’observation à la fois des ressources physiques (monitoring système) et des composants logiciels.

4.3.1 Observation des ressources physiques

Le but du framework d’observation des ressources est de rendre disponible au reste du système, et en particulier au moteur d’adaptation, toutes les informations significatives concernant l’environnement d’exécution.

Le framework est basé sur la notion de *sonde logicielle*. Une sonde est un composant logiciel très spécialisé capable de communiquer avec les couches plus basses, soit directement avec le matériel, soit avec le système d’exploitation.

Les différents éléments constituant l’hôte du système sont réifiés sous la forme d’instance de la classe `MonitoredResource` (ou d’une sous-classe). Ces éléments correspondent essentiellement aux ressources matérielles disponibles, par exemple le ou les microprocesseur(s), les périphériques de stockage, d’entrée/sortie, de communication... Chacun de ces éléments possède un nom, qui

⁵Une amélioration possible ici serait d’utiliser le modèle de conception *Memento* [Gamma et al., 1995], comme le fait Molène [André and Segarra, 2000], pour faciliter l’échange d’information entre implémentations.

doit être relativement générique et représenter le type de la ressource plutôt que ces caractéristiques. Le microprocesseur d'une station de travail sera par exemple représenté par un objet de type `MonitoredResource` nommé `cpu`, plutôt que `Intel Pentium III` ou `AMD Athlon`.

Organisation des ressources

Un système réel étant constitué d'un nombre potentiellement très important de telles ressources, celles-ci doivent être organisées. Une structure hiérarchique semble ici la plus adaptée ; il s'agit d'une structure simple et très courante. Les systèmes d'annuaires tels que X.500 et LDAP ou bien des systèmes comme SNMP sont eux aussi basés sur un tel type de structure et sont utilisés avec succès pour des tâches similaires de description détaillée des caractéristiques d'un système informatique.

En suivant ce modèle, notre système hôte sera donc décrit comme une arborescence, dont les noeuds correspondront aux différentes ressources disponibles, représentés par des objets de type `MonitoredResource`. La racine de cette arborescence est toujours nommée `host` et représente le système dans sa globalité.

Les branches de plus haut niveau (plus proches de la racine) ne représentent pas directement des ressources physiques, mais sont utilisées pour organiser ces ressources en catégories. Ces catégories représentent les différentes fonctionnalités du système, par exemple :

- capacités de calcul,
- de stockage,
- d'interaction avec l'utilisateur,
- et de communication avec l'extérieur.

Ces catégories relativement abstraites ont des noms et une structure standardisés et existent dans toutes les instances du système, même si aucune ressource physique n'y ait associée. Plus on descend dans l'arborescence, plus la description devient détaillée et donc dépendante des caractéristiques particulières de la machine hôte. Certaines branches de l'arborescence n'existeront que sur certaines machines, par exemple celle qui concerne les caractéristiques de l'écran (qui peut ne pas exister dans des systèmes embarqués).

La figure 4.3 présente sous la forme d'un document XML un exemple d'une telle arborescence, qui pourrait correspondre à la configuration d'une station de travail standard. Cet exemple n'est pas complet, mais donne une bonne idée de ce que l'on peut obtenir sans rentrer dans le détail.

Dans un cadre distribué, on a besoin de connaître un certain nombre d'informations concernant les autres machines présentes sur le réseau. Si l'on considère ces autres éléments (ou plus précisément les connexions vers ces éléments) comme des ressources du système local, ils trouvent leur place tout naturellement dans l'arborescence précédente. On définit pour cela une nouvelle catégorie de ressources `remote-hosts` enfant direct de la ressource `host`. Les différents éléments fils de cette ressource correspondent aux machines avec lesquelles une connexion directe existe, et correspondent à des copies (fragmentaires) de l'arborescence maintenue par la machine en question, ne contenant que les éléments pouvant intéresser la machine locale.

La structure représentée par cette arborescence n'est pas forcément statique. Cela est très clair en ce qui concerne l'exemple précédent, dans lequel les ressources représentent des connexions vers des hôtes distants, qui peuvent apparaître ou disparaître pendant l'exécution du système. Même dans des cas plus simple, les ressources physiques disponibles peuvent être amenées à évoluer dynamiquement. Dans le cas d'un ordinateur de bureau (PC) standard, il est tout à fait possible que de nouveaux périphériques soient branchés ou débranchés au cours de l'exécution

```
<resource name='host'>

  <resource name='computation'>
    <resource name='cpu' />
  </resource>

  <resource name='user-interaction'>
    <resource name='input'>
      <resource name='keyboard' />
      <resource name='mouse' />
      <resource name='microphone' />
    </resource>
    <resource name='output'>
      <resource name='screen' />
      <resource name='soundcard' />
    </resource>
  </resource>

  <resource name='communication'>
    <resource name='nic' />
  </resource>

  <resource name='storage'>
    <resource name='volatile'>
      <resource name='ram' />
    </resource>
    <resource name='persistent'>
      <resource name='hard-drive' />
    </resource>
  </resource>

  <resource name='power-source' />

</resource>
```

FIG. 4.3 – Exemple d'arborescence de ressources

d'un programme, par exemple un appareil photo numérique ou bien un assistant personnel. Dans le cas d'un serveur d'entreprise, on peut très bien imaginer l'ajout ou le retrait à chaud d'un disque dur ou d'un processeur. La structure exacte de l'arborescence n'est donc pas figée, et peut se modifier dynamiquement, entraînant l'apparition ou la disparition de certaines branches.

En conclusion, cette représentation de la structure des ressources disponibles permet une bonne organisation tout en restant relativement générale. Elle supporte aussi la *gestion dynamiques des ressources*.

Attributs de ressources

L'arborescence que l'on vient de présenter n'est bien entendu pas suffisante pour décrire complètement un système. Si elle permet une bonne modélisation de la structure de ce système, elle ne donne aucun détail sur les caractéristiques précises des différents constituants. Cette tâche est prise en charge par les *attributs* de ressource. Un attribut est un couple $\langle \text{nom}, \text{valeur} \rangle$ qui décrit une caractéristique de la ressource à laquelle il est attaché.

Les valeurs des attributs sont typées, les types possibles étant : nombre (flottants), booléen et chaîne de caractères. Ce modèle est un peu simpliste mais en pratique suffisant pour décrire la plupart des cas. En particulier, on a parfois besoin de types énumérés, mais ceux-ci peuvent être émulés simplement par des chaînes de caractères.

À chaque nœud de l'arborescence est donc associé un ensemble d'attributs représentant les caractéristiques de la ressource correspondante. Les attributs constituent donc les feuilles de l'arborescence. On peut faire une analogie entre ce modèle et celui utilisé en général pour représenter les systèmes de fichiers : les ressources correspondent aux répertoires, organisés de façon hiérarchique avec une racine unique, et les attributs correspondent aux fichiers contenus dans ces répertoires, le nom de l'attribut étant celui du fichier et sa valeur le contenu de ce fichier.

La figure 4.4 reprend une des ressources présentes (le microprocesseur) dans la figure 4.3 et y ajoute plusieurs attributs.

```
<resource name='cpu'>
  <attribute name='vendor' value='Intel' />
  <attribute name='type' value='Pentium III' />
  <attribute name='speed_mhz' value='800' />
  <attribute name='load' value='0.45' />
</resource>
```

FIG. 4.4 – Exemple d'attributs de ressources

Tout comme la structure des ressources elle-même, ces attributs sont complètement dynamiques. Certains peuvent être créés ou supprimés dynamiquement, et leurs valeurs peuvent évoluer au cours du temps. Cela n'a évidemment pas de sens pour certaines valeurs, comme par exemple **vendor** ou **type** dans l'exemple de la figure 4.4. Par contre, certains autres attributs sont par définition dynamiques, par exemple **load**, qui représente la charge moyenne du processeur pendant les 5 dernières minutes.

La figure 4.5 est un diagramme UML représentant la relation entre ressources et attributs, et les classes correspondantes.

On distingue deux catégories d'attributs : les *attributs primaires* et les *attributs synthétiques*. Les attributs primaires représentent des caractéristiques brutes des ressources auxquelles elles

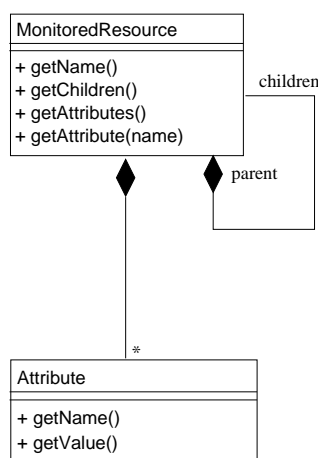


FIG. 4.5 – Diagramme UML des ressources et attributs

sont attachées. Ces attributs correspondent à des caractéristiques d’assez bas niveau. Un bon exemple d’une telle caractéristique serait un attribut `latency` représentant le temps de latence associé à une connexion réseau particulière. Les attributs synthétiques sont le résultat de calculs (arbitraires) basés sur les valeurs d’autres attributs (primaires ou synthétiques). Ils sont utilisés pour présenter des valeurs d’un niveau d’abstraction plus élevé, plus faciles à appréhender. Pour reprendre l’exemple de la connexion, réseau, le temps de latence est une caractéristique d’assez bas niveau. On peut imaginer que cette valeur soit combinée à d’autres (par exemple la bande passante disponible) pour créer un attribut synthétique `quality` représentant la qualité de la connexion sur une échelle de 1 à 10. Ce dernier attribut, même s’il est moins précis, peut s’avérer suffisant dans beaucoup de cas et est beaucoup plus simple à utiliser. On peut rapprocher ce système des moniteurs de bas niveau et de haut niveau utilisés dans Molène (cf page 23), ou bien encore des paramètres de qualité interdépendants du système QualProbes [un Li and Nahrstedt, 2000].

Les attributs primaires sont en général associés aux feuilles de l’arborescence, qui représentent des ressources physiques bien précises. Les attributs synthétiques se retrouvent plus naturellement associés aux noeuds intermédiaires. En effet, ces noeuds sont utilisés pour regrouper plusieurs ressources d’une même catégorie ; il s’agit de l’emplacement idéal pour ajouter des informations qui d’une certaine façon “résumant” les caractéristiques plus détaillées de leurs branches. Parmi les attributs synthétiques, certains sont utilisés, comme dans l’exemple de la qualité de la connexion, pour fournir une information de plus haut niveau, plus simple à manipuler. D’autres attributs synthétiques peuvent par exemple être utilisés pour donner des informations sur l’évolution dans le temps du système. En effet, dans le modèle présenté jusqu’à maintenant, les informations présentes dans l’arborescence ne représente qu’un instantané de l’état du système, qui évolue en même temps. Il ne faut pas perdre de vue que ce sont ces informations qui vont être utilisées pour décider quand on doit déclencher une adaptation. Si certaines de ces ressources subissent des variations très fréquentes de leurs attributs, et si l’on se base uniquement sur un instantané de ces attributs, on peut être amené à effectuer des adaptations inutiles, voire préjudiciables au système si les conditions qui les ont déclenchées n’étaient que transitoires [Efstratiou et al., 2001]. Par exemple, si la bande passante disponible pour une connexion chute brutalement, il est peut-être plus judicieux d’attendre quelques instants avant de déclencher une modification importante

du système, afin de s'assurer que cette chute est durable. Les attributs synthétiques répondent parfaitement à cette problématique en permettant de représenter des valeurs moyennes sur une durée donnée. En utilisant ces valeurs moyennes plutôt que des instantanés pour déclencher les adaptations, on évite une trop grande sensibilité du système à des variations non significatives.

Interfaces d'accès

Pour être utile, la description de l'environnement réalisée par notre arborescence doit être accessible au reste du système, et en particulier au moteur d'adaptation. Deux interfaces sont disponibles pour cela.

La première interface est une interface de consultation, c'est-à-dire qu'elle permet à d'autres parties du système d'interroger la "base de données" que constitue l'arborescence. Le point d'entrée de cette interface se trouve dans une classe nommée `ResourcesManager`. L'unique instance de cette classe (qui implémente le modèle de conception *Singleton*) possède en effet une méthode `getTopLevelResource()` qui renvoie l'objet `MonitoredResource` représentant la racine de l'arborescence. Une fois cet élément obtenu, il est possible de naviguer à travers l'arborescence en utilisant les méthodes définies dans la classe `MonitoredResource` et permettant de connaître l'ensemble des enfants d'une ressource ainsi que ses attributs (cf. figure 4.5).

Lorsque l'on veut connaître la valeur d'un attribut précis d'une ressource dont on connaît la position dans l'arborescence, cette interface peut être fastidieuse à utiliser (surtout si ladite ressource se trouve éloignée de la racine). Une syntaxe simple a été définie pour permettre de désigner facilement un emplacement précis de l'arborescence. Pour désigner une ressource particulière, il suffit d'indiquer le chemin à suivre depuis la racine, en donnant les noms des noeuds à parcourir séparés par un caractère `/`. Le chemin ainsi obtenu est similaire au nom d'un fichier dans un système de fichier Unix. Pour désigner un attribut de cette ressource, il suffit de rajouter à ce chemin le nom de l'attribut précédé d'un point. La figure 4.6 montre quelques exemples de tels chemins se rapportant aux ressources décrites dans les exemples précédents.

```
/host/cpu
/host/user-interaction/output/screen.width
/host/storage/volatile/ram.available
```

FIG. 4.6 – Exemples de désignation de ressources et attributs

La seconde interface est une interface de notification, basée sur un modèle événementiel. La classe `MonitoredResource` implémente le modèle de conception *Observer*. N'importe quel élément du système, s'il implémente l'interface `EventListener`, peut donc s'abonner auprès d'une ressource particulière pour être prévenu de tout changement concernant cette ressource. Les changements sont des deux types :

- changement de valeur d'un attribut attaché à la ressource, représenté par une instance de la classe `AttributeChangeEvent`,
- et changement dans la structure de l'arborescence (apparition ou disparition d'un enfant de la ressource en question), représentés par des instances des classes `ResourceAddedEvent` et `ResourceRemovedEvent`.

Découverte des ressources & association des sondes

Notre infrastructure devant pouvoir être utilisée avec une grande variété de matériel, on ne peut faire aucune supposition quand aux ressources disponibles, et donc à la structure exacte de l'arborescence. Au cours de son initialisation, le système doit donc être capable de découvrir de façon plus ou moins automatique quelles sont les ressources de l'hôte qui l'accueille. Par exemple, s'il s'agit d'un assistant personnel, le système doit pouvoir détecter la présence d'un écran, d'une batterie à l'autonomie limitée, d'une interface de communication par infrarouge... A l'opposé du spectre matériel, l'hôte peut être un serveur multi-processeur, sans écran ou clavier, qui ne communique avec l'extérieur que grâce à une connexion Ethernet gigabit. Il est bien évident que les services disponibles et les types d'adaptation envisageables ne seront pas les mêmes dans les deux cas.

Deux stratégies différentes sont possibles pour permettre la découverte de ces ressources. L'une statique, l'autre plus dynamique.

La première solution nécessite qu'une personne connaissant bien le système hôte crée une description des ressources de ce système dans un fichier de configuration. Ce fichier sera lu par le système au démarrage pour déterminer la structure initiale de l'arborescence et les sondes à mettre en place. Ce fichier de configuration au format XML indique quelles ressources existent dans le système, et pour chacune de ces ressources, quelles sondes y attacher. Les sondes sont les éléments actifs de l'arborescence, qui communiquent avec les couches système et matérielles pour obtenir les caractéristiques des ressources et les rendent visibles sous la forme d'attributs. Chaque noeud de l'arborescence se voit donc attribuer un ensemble de sondes chargées de créer et maintenir les attributs qu'elles sont capables d'observer. Le fichier de configuration doit donc spécifier pour chaque noeud quels types de sondes y attacher. La figure 4.7 montre un exemple minimal d'un tel fichier de configuration. Cette solution a évidemment l'inconvénient de nécessiter une très bonne connaissance des machines sur lesquelles le système est déployé.

```
<?xml version='1.0' ?>
<resource name='host'>
  <probe class='probes.GenericHostProbe' />

  <resource name='computation'>
    <resource name='cpu'>
      <probe class='probes.X86CPUProbe' />
    </resource>
  </resource>

  ...

  <resource name='power-source'>
    <probe class='probes.ACPowerProbe' />
  </resource>

</resource>
```

FIG. 4.7 – Exemple de fichier de configuration des ressources

À l'inverse, la seconde solution tire partie du fait que l'arborescence est complètement dynamique pour la construire à l'initialisation du système en détectant automatiquement le ma-

tériel disponible. Cette fois-ci le fichier de configuration existe toujours, mais est réduit au minimum : seules sont spécifiés la racine de l'arborescence et la classe utilisée pour la représenter. En effet, cette technique de détection (semi-)automatique est implémentée en sous-classant `MonitoredResource`. Une version spécialisée de cette classe doit être écrite, capable d'effectuer la détection et de créer les noeuds appropriés. Bien entendu, écrire une telle classe réellement générale est impossible, c'est pourquoi le fichier de configuration doit toujours être présent et spécifier laquelle doit être utilisée. On peut imaginer par exemple créer une classe `PalmPilotResource`, sous-classe de `MonitoredResource` et à même de détecter les différents modèles de Palm Pilot et leurs périphériques propres. Le fichier de configuration pourrait alors ressembler à la figure 4.8. Tous les noeuds de l'arborescence et leurs sondes associées seront créés par la racine à l'initialisation, mais aussi dynamiquement si de nouvelles ressources apparaissent (ou disparaissent) en cours d'exécution.

```
<?xml version='1.0' ?>
<resource name='host' class='com.palm.PalmPilotResource' />
```

FIG. 4.8 – Exemple de fichier de configuration automatisé

En fait, même dans le cadre d'un type particulier de machines, par exemple les PCs, il peut y avoir une très grande diversité de matériel possible, et l'écriture d'un composant capable de gérer tous les cas est trop difficile en pratique. La solution consiste donc à mélanger les deux approches extrêmes. On utilise un composant relativement général capable de détecter les périphériques standards du type de matériel utilisé, et si l'on sait que la machine particulière sur laquelle le système est installé dispose de ressources spécifiques, on le précise dans le fichier de configuration. La figure 4.9 montre un exemple de tel fichier de configuration pour un PC ayant comme seule particularité de posséder une carte de décompression vidéo.

```
<?xml version='1.0' ?>
<resource name='host' class='GenericPCResource'>
  <resource name='computation'>
    <resource name='mpeg_decompression' class='com.atl.RageTheaterResource' />
  </resource>
</resource>
```

FIG. 4.9 – Exemple de configuration mixte

Implémentation

La classe de base `MonitoredResource` fournie par le système ne définit aucun attribut. Lorsqu'un noeud de l'arborescence est une instance de cette classe `MonitoredResource`, les seuls attributs qui y sont attachés sont ceux gérés par les différentes sondes associées au noeud. En revanche, si le noeud est une instance d'une classe spécialisée (par exemple une classe représentant un processeur graphique spécifique), cette classe peut définir ses propres attributs. C'est de cette façon que sont gérés les attributs synthétiques.

Les sondes associées à un noeud sont des objets actifs. À intervalle régulier, dépendant de la sonde, elles communiquent avec le système d'exploitation, ou directement avec le matériel pour

obtenir la ou les valeurs actuelles des attributs dont elles ont la charge, et modifient la valeur de cet attribut dans le noeud auquel elles sont attachées. Le noeud prend alors la main et peut éventuellement mettre à jour des attributs synthétiques avant de notifier ses observateurs des changements qui ont eu lieu.

Les sondes sont exécutées de façon concurrente, chacune dans son propre thread. Pour éviter un impact trop important sur les performances du système, il vaut mieux avoir un nombre minimal de sondes associées à un noeud, et les traitements qu'elles effectuent doivent rester légers.

La façon dont les sondes sont implémentées est très dépendante du matériel et du système d'exploitation. Le cas du système GNU/Linux est intéressant ici car le noyau Linux offre en standard une interface simple permettant d'obtenir de nombreuses informations. Linux implémente un système de fichier virtuel dans le répertoire `/proc`, dont les fichiers et répertoires n'existent sur aucun disque, mais sont une interface avec le noyau. Par exemple, le fichier `/proc/cpuinfo` donne un certain nombre d'informations concernant le(s) microprocesseur(s) présents sur la machine. Il s'agit toujours de simples fichiers texte, et il n'est besoin d'aucune interface de bas niveau ou d'appels système pour obtenir les informations qu'ils contiennent : il suffit de savoir lire et analyser un fichier texte. En pratique, il est possible de récupérer grâce au contenu de ces fichiers la liste des périphériques reconnus par le noyau Linux comme étant présents sur la machine, mais aussi des informations plus dynamiques comme la charge système, le nombre de paquets IP envoyés ou reçus par une interface réseau, et de nombreux autres paramètres⁶.

4.3.2 Observation des composants logiciels

L'arborescence décrite en détail dans la section précédente permet au système d'avoir à tout moment une représentation simple des caractéristiques de l'environnement d'exécution. Cependant, connaître l'environnement n'est pas suffisant pour permettre l'adaptation ; il faut aussi connaître l'application à adapter elle-même, et donc être capable d'observer les composants qui la constitue. En effet, il est indispensable de pouvoir distinguer les composants fonctionnels entre eux, en fonction de leurs caractéristiques propres, pour pouvoir effectuer des modifications qui leur soient adaptées, plutôt que des modifications globales.

Par exemple, dans une application distribuée, il est possible d'avoir certains composants qui ne doivent pas être visibles depuis d'autres sites (pour des raisons de sécurité par exemple). On ne peut donc pas activer un service comme la distribution aveuglément, à tous les composants du système, sans tenir compte de leurs caractéristiques. Dans cet exemple, la distinction entre les composants qui peuvent ou qui ne peuvent pas être distribués est liée à leur sémantique (par exemple s'ils représentent un compte bancaire) plutôt qu'à des caractéristiques purement techniques (leur taille en mémoire).

Les techniques réflexives utilisées nous permettent déjà de connaître en partie les composants qui constituent l'application. Cependant, ceux-ci ne sont connus que par l'intermédiaire de leur conteneur. En tant que tels, les conteneurs ne permettent pas au système de distinguer les composants qu'ils encapsulent.

Le modèle actuel ne définit pas réellement de modèle de composant. Les composants logiciels constituant la partie fonctionnelle de l'application sont simplement des objets réflexifs, c'est-à-

⁶En fait, on peut presque considérer ce système de fichier comme une interface réflexive vers le noyau Linux. En plus de la simple lecture d'informations, certains de ces fichiers peuvent être utilisés pour reconfigurer le noyau, tout simplement en écrivant dans le fichier la nouvelle valeur du paramètre que l'on veut changer.

dire contrôlés au niveau méta par un métaobjet. Du point de vue du moteur d'adaptation, les composants fonctionnels n'existent que par leur métaobjet, instance de la classe `Container`. Ces conteneurs encapsulent les objets de base et contrôlent toutes leurs interactions avec l'extérieur.

Pour permettre de distinguer ces composants les uns des autres, on leur associe des attributs. Ces attributs sont définis en utilisant le même formalisme que celui utilisé pour décrire les ressources du système : il s'agit tout simplement de couples $\langle \text{nom}, \text{valeur} \rangle$ décrivant les caractéristiques significatives des composants. En réalité, ces attributs ne sont pas attachés directement aux composants, mais à leur conteneur ; cela évite de modifier plus que nécessaire les objets du niveau de base.

Certains de ces attributs, dits *attributs génériques*, existent pour tous les composants et ont une sémantique définie une fois pour toute. C'est le cas par exemple de l'attribut `className`, qui est une chaîne de caractère représentant le nom de la classe de l'objet de base. Ces attributs sont générés automatiquement par le système.

D'autres attributs sont spécifiques à une classe de composants, voire à une instance particulière. Il est possible de configurer le système pour que certains champs d'une classe donnée soient automatiquement exportés sous forme d'attributs. Les conteneurs associés à des instances d'une telle classe ayant le contrôle sur les accès à ces champs sont capable de maintenir dynamiquement ces attributs pour que leur valeur reflètent toujours celle des champs de l'objet de base. Enfin, les attributs pouvant être créés ou supprimés dynamiquement, les différents services attachés à un composant à un moment donné peuvent utiliser ce mécanisme pour partager des informations.

4.4 Décision

Nous disposons à présent

- d'un mécanisme permettant de connaître à tout moment l'état de l'environnement d'exécution et celui des applications elles-mêmes,
- et d'un système permettant de modifier dynamiquement les mécanismes d'exécution des composants constituant les applications.

Il nous reste donc à faire le lien entre ces deux éléments, c'est-à-dire à décider en fonction des informations fournies par ce premier mécanisme de quand et comment utiliser le second. Par rapport au schéma fonctionnel présenté dans la figure 4.1, page 39, cela correspond aux politiques d'adaptation et au moteur d'adaptation.

On peut considérer le moteur d'adaptation comme un mécanisme général pour l'adaptation dynamique. Cependant, nous avons déjà dit à plusieurs reprises que la notion même d'adaptation fait intervenir des éléments de sémantique qui ne peuvent justement pas être capturés complètement dans un mécanisme générique. Les politiques d'adaptation, qui elles dépendent (plus ou moins comme nous le verrons) de l'application à adapter, sont donc injectées dans le moteur en tant que configuration. *Le moteur d'adaptation est donc un interprète pour les politiques.*

Nous n'avons pas défini clairement jusqu'ici ce que sont ces politiques, mais seulement leur rôle, qui est de faire le lien entre les mécanismes généraux fournis par le système et les particularités d'une application donnée. D'un point de vue plus "opérationnel", une politique d'adaptation spécifie les modifications à appliquer à quelles parties de l'application et sous quelles conditions. Une politique d'adaptation doit donc être capable de :

- décrire des conditions relatives à l'environnement d'exécution ainsi qu'à l'application elle-même,

- spécifier des modifications à appliquer,
- désigner les cibles de ces modifications parmi les différents éléments constituant l'application (*i.e.* les composants fonctionnels).

4.4.1 Différents niveaux d'abstraction

Avant de décrire la façon dont nos politiques implémentent ces différentes fonctionnalités, on peut se demander *qui* va écrire ces politiques. Puisque le rôle de ces politiques est de configurer le moteur d'adaptation par rapport à une application donnée, ces politiques ne peuvent être définies qu'en ayant une bonne connaissance de l'application. À priori, c'est donc au *programmeur de l'application* qu'il revient d'écrire ces politiques d'adaptation. On peut aussi imaginer que la personne qui effectue le déploiement de l'application (*dépoyeur*) sur une machine donnée puisse vouloir modifier ces politiques pour les affiner en fonction des caractéristiques particulières de cette machine, ou tout simplement de besoins différents.

Le problème de cette approche est que les politiques d'adaptation doivent pouvoir être exprimées en fonction de conditions environnementales précises. Or, le but de notre système est justement d'éviter au programmeur d'avoir à se préoccuper de ces détails techniques de bas niveau. Heureusement, ces détails techniques sont souvent assez éloignés des préoccupations des couches applicatives. Ils peuvent donc être décrits de façon relativement indépendante des applications, et par des spécialistes des domaines correspondant qui programment au niveau du middleware.

On ressent donc le besoin d'avoir différents niveaux d'abstraction. Cela va se traduire par plusieurs types de politiques d'adaptation, destinées à être définies par des personnes différentes ayant des rôles différents et se situant aux niveaux d'abstraction correspondants. C'est la réunion de ces différentes politiques qui formera la véritable configuration du moteur d'adaptation.

On choisit de séparer les politiques qui dépendent d'une application particulière de celles qui sont plus générales et peuvent être réutilisées dans différents contextes. On distinguera donc

- les *politiques système* qui traiteront des détails de bas niveau et seront indépendantes de la sémantique des applications ;
- et les *politiques applicatives*, écrites par les programmeurs spécifiquement pour leur application, et d'un niveau d'abstraction plus élevé.

Les politiques système seront écrites par les programmeurs du middleware, plus familiers avec les détails de bas niveau (contraintes matérielles par exemple). Les politiques applicatives seront par contre écrites par les programmeurs d'application, seuls à connaître la sémantique exacte de l'application. Enfin, le dépoyeur, chargé d'installer et de configurer une application, peut avoir besoin de retoucher ces deux types de politiques pour les adapter aux conditions particulières de l'installation (réalisant ainsi une adaptation statique).

Voyons maintenant en détail ces deux types de politiques avant d'expliquer comment elles fonctionnent ensemble.

4.4.2 Politiques système

Les politiques système sont des politiques de bas niveau, dont le but est de définir des règles d'adaptation *de façon indépendante d'une application*. Ces règles spécifient quels services doivent être utilisés, et avec quels paramètres, en fonction des conditions environnementales. Par contre, elles ne spécifient pas à quels composants de l'application (qu'elles ne connaissent d'ailleurs pas) appliquer ces services. Concrètement, une telle politique est un ensemble de règles de la forme

condition \Rightarrow *actions*

où :

- *condition* est une expression booléenne concernant l’environnement d’exécution tel qu’il est représenté par l’arborescence décrite à la section 4.3.
- *actions* est une liste d’actions, chacune correspondant à l’activation ou la désactivation d’un rôle particulier d’un service, tels qu’ils sont décrits à la section 4.2.

En pratique, les politiques, qu’elles soient politiques système ou applicatives, sont écrites dans des fichiers XML chargés par le système au démarrage. La figure 4.10 montre la forme générale d’un fichier contenant une politique système.

```
<?xml version='1.0' ?>
<system-policy name='nom_de_la_politique'>
  <rule>
    <when>
      condition environnementale
    </when>
    <ensure>
      activations ou désactivations de rôles (avec paramètres)
    </ensure>
  </rule>
  .
  .
  autres règles
  .
  .
</system-policy>
```

FIG. 4.10 – Forme générale d’une politique système

Les sections suivantes décrivent la forme et la sémantique précise des conditions et des actions.

Conditions environnementales

Les conditions environnementales présentes dans les règles des politiques système sont des expressions booléennes.

Les variables disponibles pour l’écriture de ces conditions correspondent aux attributs des ressources disponibles (représentées dans l’arborescence décrite plus haut). La syntaxe pour ce faire utilise un élément XML nommé `attribute-value` :

```
<attribute-value name='/chemin/de/la/ressource.nom_de_l_attribut' />
```

Il est possible de spécifier des constantes numériques, booléennes ou chaînes de caractères comme ceci :

```
<number value='42' />
<number value='-3.14159' />
```

```
<boolean value='true' />
<boolean value='false' />
```

```
<string value='une chaîne quelconque' />
```

```
<string value='une chaîne avec une apostrophe: &apos;'/>
<string value="une chaîne avec un guillemet: &quot;"/>
```

Ces valeurs primitives peuvent être combinées avec des opérateurs de comparaison et les opérateurs booléens classiques. Les quatre opérations de base sur les nombres sont aussi disponibles (addition, soustraction, multiplication et division) :

```
<equals> <something/> <somethin_else/> </equals>
<less-than> <a_number/> <another_number/> </less-than>
<greater-than> <a_number/> <another_number/> </greater-than>

<not> <a_boolean/> </not>
<and> <a_boolean/> <another_boolean/> </and>
<or> <a_boolean/> <another_boolean/> </or>

<add> <a_number/> <another_number/> </add>
<subtract> <a_number/> <another_number/> </subtract>
<multiply> <a_number/> <another_number/> </multiply>
<divide> <a_number/> <another_number/> </divide>
```

On peut ainsi construire des conditions relativement complexes (bien que pas toujours très lisibles), par exemple :

```
<and>
  <equals>
    <attribute-value name='/host/computation/cpu.type' />
    <string value='Athlon' />
  </equals>
  <greater-than>
    <attribute-value name='/host/computation/cpu.load' />
    <number value='0.95' />
  </greater-than>
</and>
```

Les objets modélisant ces conditions implémentent le modèle de conception *Observer* pour notifier leurs abonnés lorsque leur valeur change (devient vrai ou fausse). Lorsque la valeur d'un attribut change, les conditions environnementales qui dépendent de cet attribut sont automatiquement réévalués, et si leur valeur s'en trouve modifiée, les objets représentant ces conditions notifient leurs propres abonnés de la modification.

Spécification des associations composants/rôles

Le deuxième élément intervenant dans la définition d'une règle d'adaptation est une liste d'actions. Ces actions correspondent à la spécification des associations entre composants et services (en fait entre composants et rôles).

En fait, les actions ne sont pas exprimées directement dans une règle. Ce que la règle spécifie correspond plutôt à des "contraintes". C'est au moment de l'interprétation de la règle que ces contraintes seront réalisées par des actions. Nous avons fait ce choix pour rendre l'écriture des politiques système plus aisée, en les rendant plus déclaratives qu'impératives. Plutôt que de spécifier le *comment* (actions), la personne qui écrit la politique spécifie le *quoi*. C'est alors au moteur d'adaptation de déterminer les actions à réaliser pour atteindre ce but, en fonction de l'état actuel du système (ici les services déjà attachés et leur configuration).

Cette approche donne en outre la possibilité au moteur d'adaptation de coordonner plusieurs politiques d'adaptation. Si le moteur se contentait d'appliquer aveuglément des ordres spécifiés dans les politiques système, il serait possible d'écrire des politiques incompatibles entre elles (par exemple une politique qui annule systématiquement les actions d'une autre). En ayant une approche plus déclarative, on permet au moteur de faire des choix plus sophistiqués quand aux actions à mettre en œuvre. Bien sûr, il est toujours possible d'écrire des politiques contradictoires, mais le système est maintenant capable de le détecter et de choisir les règles à mettre en œuvre en fonction de priorités associées aux politiques. Le système étant capable de gérer plusieurs politiques système, il ordonne en effet celles-ci en fonction de leur ordre de chargement ; les premières politiques chargées sont les plus prioritaires⁷.

Un seul type de "contrainte" est disponible, mais il peut se traduire concrètement par trois types d'actions en fonction des circonstances. Une contrainte permet de spécifier qu'un rôle particulier doit être actif et configuré d'une certaine manière. Lorsque la condition environnementale associée à cette contrainte est vraie, le système s'assure que cette contrainte est vérifiée. À l'inverse, si la condition est fautive, le système désactive le rôle correspondant. Du point de vue de la syntaxe, une contrainte a la forme suivante :

```
<attached service='nom.du.service.abstrait' role='nom.du.role'>
  <parameter name='paramètre-1' value='valeur-1' />
  <parameter name='paramètre-2' value='valeur-2' />
</attached>
```

Étant donnée une règle complète comme celle-ci

```
<rule>
  <when>
    <une_condition_environnementale />
  </when>
  <ensure>
    <attached service='nom.service' role='nom.role'>
      <parameter name='param1' value='value1' />
      <parameter name='param2' value='value2' />
    </attached>
  </ensure>
</rule>
```

Lorsque la condition environnementale devient vraie, le système doit s'assurer que le rôle désigné par la contrainte est bien attaché et configuré avec les bons paramètres. Deux cas peuvent se produire :

- Dans le cas où aucun métaobjet implémentant ce rôle n'est déjà attaché, le système doit donc trouver, parmi les différentes implémentations concrètes du rôle, une implémentation qui est capable de se configurer suivant ces paramètres. Pour se faire, le moteur d'adaptation va interroger toutes les implémentations de ce rôle existant dans le système, jusqu'à en trouver une qui accepte ces paramètres. Une fois cette implémentation trouvée, il ne reste plus qu'à effectuer l'attachement.

Par exemple, s'il s'agit du rôle **server** d'un service de distribution, l'utilisateur peut avoir spécifié en paramètre l'adresse d'un serveur JNDI où enregistrer les composants. Si deux

⁷Le problème des priorités entre politiques, voire entre règles d'adaptation, est très complexe. La solution actuelle n'est que temporaire, et des systèmes plus sophistiqués, comme par exemple celui proposé par DART [Raverdy and Lea, 1999], seront étudiés dans l'avenir.

implémentations de ce service, par exemple CORBA et RMI coexistent dans le système, seule l'implémentation basée sur RMI est capable d'interpréter correctement cette valeur de paramètre.

- S'il y a déjà un métaobjet implémentant le rôle demandé attaché, le système lui demande simplement de se reconfigurer avec les nouveaux paramètres. Si le rôle indique qu'il n'est pas capable d'effectuer cette configuration, le système le retire et le remplace par un autre en utilisant la méthode précédemment décrite.

Lorsque la condition devient fausse, le système se contente tout simplement de détacher le rôle.

La figure 4.11 résume ce mécanisme, sous la forme d'un diagramme d'état. Les états correspondent aux valeurs possibles pour la condition environnementale, et les transitions portent les différentes actions que le système peut être amené à effectuer lors d'un changement de valeur.

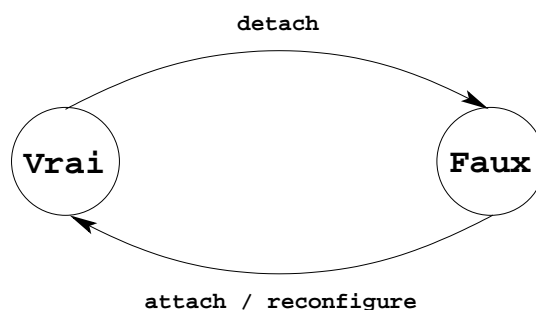


FIG. 4.11 – Fonctionnement d'une règle d'adaptation

4.4.3 Politiques applicatives

Les politiques applicatives correspondent au deuxième type de politiques d'adaptation supportées par le système. Ces politiques sont d'un niveau d'abstraction plus élevé que les politiques systèmes, et sont complémentaires de celles-ci. Comme leur nom l'indique, elles sont spécifiques à une application, et sont destinées à être écrites par les programmeurs d'applications et/ou les administrateurs (chargés d'installer et de configurer une application sur un site donné).

Les politiques système ne sont pas suffisantes pour réaliser une adaptation. Elles indiquent *quand déclencher* une modification de l'application et *quelle modification* appliquer, mais il manque une information essentielle : à quels éléments de l'application doit on appliquer ces modifications. Au moment où sont écrites ces politiques, on ne connaît encore rien de la structure des applications qui les utiliseront, et il est normal que les politiques système ne puissent pas donner cette information. Si on se contente de ces politiques, la seule façon de les utiliser est d'appliquer toutes les règles à tous les composants de toutes les applications. Il est bien évident qu'une telle stratégie de type "tout ou rien" ne peut pas fonctionner dans le cas général, puisque nous avons dit dès le début qu'une adaptation se devait de prendre en compte les particularités des composants à modifier.

Le rôle des politiques applicatives est donc de combler ce manque. Une politique applicative décrit donc comment vont être appliquées les politiques système, et à quels composants parmi ceux constituant l'application.

Gestion des composants par groupes

Dans la plupart des cas, il est impossible de prévoir statiquement quels seront exactement les composants existant au cours de la vie de l'application. On peut éventuellement connaître la configuration initiale de l'application, mais si on identifie les composants aux objets (comme nous le faisons ici), de nouveaux composants seront créés ou disparaîtront régulièrement et de façon imprévisible au cours de la vie de l'application. Au moment où l'on écrit la politique d'une application, il est donc impossible de désigner individuellement les composants pour leur affecter des politiques système.

Pour résoudre ce problème, on introduit la notion de *groupe de composants*. Ces groupes sont destinés à regrouper des composants "similaires", dans le sens où ils doivent être traités de la même manière. Souvent, un groupe correspondra donc plus ou moins à une classe, puisque toutes les instances d'une classe donnée partagent normalement la même sémantique. Cependant, on peut vouloir traiter différemment certaines instances d'une classe donnée, ou bien traiter de la même manière des instances de plusieurs classes différentes ; c'est exactement ce que permettent de faire les groupes de composants.

Une politique applicative est constituée de définitions de groupes de composants, et d'associations entre ces groupes et des politiques systèmes. Les éléments d'un groupe G associé à une politique système S seront ceux auxquels les règles de ladite politique système s'appliquent.

Définition des groupes

Comme on l'a déjà dit à plusieurs reprises, les composants (objets) sont représentés dans le système par leur conteneur (métaobjet), et les seules informations sur le composant auxquelles ce conteneur donne accès sont les attributs qui y sont attachés (cf. section 4.3.2, page 52). C'est donc par rapport à ces attributs que les groupes vont être définis.

Un groupe G est défini par :

- un *super-groupe* parent G_p ,
- un prédicat p exprimé en fonctions des attributs de composants.

Étant donnés ces deux éléments, les membres m de G sont définis par

$$G = \{m \in G_p \mid p(m)\}$$

Autrement dit G est le sous-groupe des éléments de G_p qui vérifient le prédicat p . On peut voir le prédicat comme un fonction de filtrage. Le prédicat faisant référence aux attributs des composants, les groupes rassemble tous les composant ayant certaines caractéristiques communes.

Le prédicat utilisé pour définir un groupe utilise le même formalisme (et la même syntaxe XML) que les expressions utilisées pour définir les conditions environnementales dans les politiques système. La seule différence est que dans un tel prédicat, il est possible de référencer des attributs "libres", dans le sens de *variable libre*, c'est-à-dire qui ne sont pas associés à une ressource. Ces attributs sont interprétés comme étant relatifs aux composants fonctionnels. Ainsi,

```
<attribute-value name='balance' />
```

désignera la valeur de l'attribut nommé **balance** associé à un composant. Une telle expression n'a pas de valeur intrinsèque. Elle doit être évaluée par rapport à un composant. Si ce composant ne possède pas d'attribut **balance**, la valeur de l'expression dépend du contexte : elle vaut **false**

dans un contexte booléen, 0 dans un contexte numérique et "" (chaîne vide) dans un contexte de chaîne de caractère.

En plus des attributs définis explicitement pour un composant, le système ajoute automatiquement à tous les composants des *attributs génériques*. Ces attributs présents par défaut sur tous les composants permettent de décrire ceux-ci selon des critères générique, et facilitent l'écriture de politiques d'application. Actuellement, le seul attribut de ce type se nomme `className`, et a pour valeur le nom de la classe du composant. Cet attribut permet de définir facilement des groupes correspondant à toutes les instances d'une classe, qui doivent souvent être traités de façon similaire (et donc être associés aux mêmes politiques système). On peut imaginer d'autres attributs génériques, comme par exemple le nom de la machine où un composant a été créé, ou bien la quantité de mémoire qu'il occupe.

Concrètement, les politiques applicatives sont définies dans des fichiers XML, comme les politiques système. Une politique applicative peut définir plusieurs groupes de composants, en utilisant la syntaxe présentée dans la figure 4.12.

```
<group name='nom_du_groupe'>
  <select from='nom_du_groupe_parent'>
    <predicat/>
  </select>
</group>
```

FIG. 4.12 – Syntaxe pour la définition de groupes de composants

Par exemple, pour définir un groupe contenant à tout moment toutes les instances de la classe `Foo`, il suffit d'utiliser l'attribut prédéfini `className` de cette façon :

```
<group name='fooComponents'>
  <select from='all'>
    <equals>
      <attribute-value name='className' />
      <string value='Foo' />
    </equals>
  </select>
</group>
```

Puisqu'un groupe est toujours défini par rapport à un super-groupe parent, ceux-ci sont organisés de façon hiérarchique (cf figure 4.13), sachant qu'un composant donné peut évidemment se retrouver dans plusieurs groupes. Pour pouvoir amorcer le processus, le système définit un groupe initial, nommé `all`, qui contient à tout moment tous les composants existants dans le système.

Les valeurs des attributs associés à un composant donné, et même l'ensemble de ces attributs évolue pendant l'exécution de l'application. Les groupes étant définis en fonction de ces valeurs, il sont eux aussi complètement dynamiques. Cela signifie que le contenu d'un groupe n'est pas figé, et qu'un composant peut entrer ou sortir d'un ou plusieurs groupes à chaque fois qu'un de ses attributs change. Le système est responsable de faire en sorte qu'à tout moment, tous les composants soient membres de groupes appropriés, et uniquement de ceux-ci.

Dans l'exemple précédent, à chaque fois qu'une nouvelle instance de la classe `Foo` est créée, le système lui affecte un attribut `className` avec la valeur `"Foo"`. Ceci déclenche automatiquement

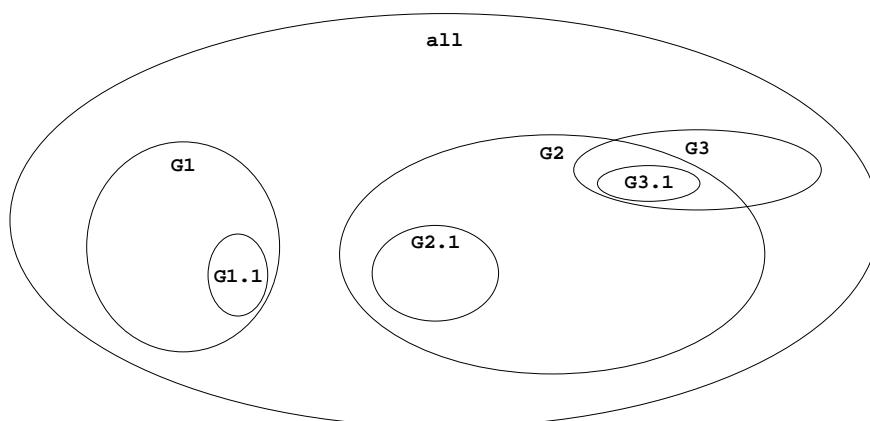


FIG. 4.13 – Organisation des groupes de composants

l'entrée de ce composant dans le groupe `fooComponents`. De la même manière, lorsqu'un composant de type `Foo` est détruit, son attribut disparaît et le composant est retiré automatiquement du groupe.

Cette gestion complètement dynamique et transparente de la composition de groupes est basée sur un modèle événementiel de type *Observer* [Gamma et al., 1995]. Les groupes dont la composition dépend de certains attributs s'abonnent auprès du système pour être notifiés à chaque fois que la valeur d'un tel attribut est modifiée dans le système, afin de pouvoir automatiquement décider si le composant en question doit être ajouté ou retiré du groupe.

4.4.4 Liaison entre politiques système et applicatives

Nous avons dit plus haut que le rôle des politiques applicatives était de compléter les politiques système en désignant quels composants parmi ceux constituant une application doivent être affectés par quelle politique système. Il nous reste donc à voir comment spécifier l'association d'un groupe de composants avec une politique système.

C'est en associant une ou plusieurs politique(s) système à un groupe de composants que l'on va pouvoir spécifier comment les membres de ce groupes doivent être traités. Si par exemple les classes `Foo` et `Bar` doivent être persistantes, il suffit de définir dans la politique applicative un groupe `composants-persistants` défini comme l'ensemble des instances de ces deux classes, et d'associer à ce groupe une politique système de persistance. Cette politique de plus bas niveau se chargera d'implémenter la persistance de façon adaptée aux circonstances, en utilisant de façon appropriée divers services de plus bas niveau (persistance simple, réplication, caches...).

Cette association entre un groupe et une (ou plusieurs) politiques système se fait lors de la définition du groupe. La figure 4.14 montre la syntaxe générale d'une telle association.

En fait, il est aussi possible de spécifier directement dans une politique applicative une association entre un groupe de composants et un rôle, plutôt que de passer par l'intermédiaire d'une politique système, en utilisant la syntaxe suivante :

```
<bind service='nom_du_service' role='nom_du_rôle'>
  <parameter name='nom_du_paramètre' value='valeur' />
</bind>
```

```
<group name='nom_du_groupe'>
  <select from='groupe_parent'>
    <predicat/>
  </select>
  <bind policy='nom_politique_systeme' />
</group>
```

FIG. 4.14 – Syntaxe pour l’association entre groupes de composants et politiques système

Une politique système associée à un groupe affecte tous les membres de ce groupe, et uniquement ceux-ci. Lorsqu’un composant entre dans un groupe, parce que l’un de ces attributs a changé et que le prédicat définissant le groupe devient valide pour ce composant, le système applique automatiquement les règles de cette politique système au nouveau membre. De manière symétrique, l’effet des règles est annulé si le composant sort du groupe.

4.5 Utilisation du système

Cette section décrit de façon relativement abstraite un scénario d’utilisation du système, montrant les différentes étapes nécessaires et les différents rôles impliqués. La section 5.8, page 74 décrit un exemple plus concret d’utilisation du prototype lui-même.

4.5.1 Éléments constituant le système

Du point de vue d’un développeur, le système de base est constitué de deux éléments :

- un compilateur spécial à utiliser pour compiler les composants de base,
- et un lanceur d’applications.

Le compilateur est utilisé pour rendre réflexifs les composants de base, de façon à ce qu’ils soient utilisables par le MOP choisi, celui de RAM [Bouraçadi-Saâdani et al., 2000]. Le lanceur d’application se charge de configurer la plate-forme à partir des politiques stockées dans des fichiers XML avant de lancer l’application de l’utilisateur.

Pour être opérationnelle, cette plate-forme doit être complétée par :

- des sondes logicielles concrètes ;
- des bibliothèques de services ;
- des fichiers de configuration (politiques).

Les sondes logicielles sont utilisées pour observer les ressources matérielles et les rendre disponibles sous la forme d’une arborescence de ressources. On peut imaginer plusieurs bibliothèques de sondes, destinées à divers types de plate-formes (PC, PDA...). Ces bibliothèques peuvent être développées de façon complètement indépendantes d’une application, et vendues séparément. Elles peuvent aussi inclure des fichiers de configuration pour automatiser la découverte des ressources, comme le montre la figure 4.7, page 50.

Les bibliothèques de services peuvent elles aussi être développées de façon complètement indépendantes d’une application, par des spécialistes du domaine (distribution, sécurité, persistance...). Ces bibliothèques sont alors disponibles pour le programmeur d’application, qui peut choisir parmi les services dont il a besoin les implémentations qui lui conviennent le mieux. Ces bibliothèques peuvent être plus ou moins complexes et couvrir plusieurs services non-fonctionnels.

Elles peuvent aussi fournir des politiques système appropriées directement utilisables par le programmeur.

4.5.2 Travail du programmeur d'application

Le programmeur n'a donc plus qu'à :

1. développer ses composants métier, et les compiler avec le compilateur fourni ;
2. se procurer une ou plusieurs bibliothèques de sondes logicielles, spécifiques aux plates-formes sur lesquelles l'application doit être déployée ;
3. déterminer les services non-fonctionnels nécessaires à son application ;
4. se procurer des implémentations de ces services ;
5. éventuellement écrire des politiques système ou personnaliser celles fournies avec les implémentations de services ;
6. écrire une politique applicative reliant ces services aux composants fonctionnels.

L'application est alors prête à être installée et exécutée ; le système se chargera automatiquement de l'adapter aux conditions d'exécution en fonction des informations fournies par les politiques.

Dans la liste précédente, les étapes 2, 4 et 5 peuvent éventuellement être réalisées sans l'intervention directe du programmeur d'application ; il s'agit plutôt de tâches d'administration et de déploiement, qui peuvent être réalisées par une autre personne, sans qu'elle n'ait besoin de connaître les détails de l'architecture de l'application.

4.5.3 Conclusion

Ce système permet une bonne séparation des rôles et permet aussi d'augmenter la réutilisabilité de certains composants (services, sondes).

D'un côté, le programmeur d'applications se concentre presque exclusivement sur le code métier ; en dehors de ce code, il doit seulement écrire une politique applicative d'un niveau d'abstraction relativement élevé. De l'autre, les programmeurs de middleware, qui peuvent être des spécialistes de domaines très techniques (distribution, réplication, sécurité...), ont la possibilité d'implémenter des services non-fonctionnels de qualité, indépendamment d'une application particulière, tout en sachant qu'ils pourront être utilisés facilement par les applications qui en auront besoin. Enfin, le déployeur fait le lien entre ces deux rôles, en choisissant — en respectant les contraintes spécifiées par le programmeur d'application — et en configurant les services non-fonctionnels nécessaires à une application dans un contexte (matériel) donné.

Chapitre 5

Description du prototype

Ce chapitre décrit de façon plus concrète le prototype que nous avons réalisé et qui implémente la plupart des fonctionnalités décrites dans le chapitre précédent.

5.1 Architecture générale

Le prototype est écrit en Java standard (Java Development Kit 1.3, Linux). Le cœur du système est organisé dans cinq packages Java :

- `proto.monitoring` : framework d'observation. Ce package contient les classes permettant de gérer l'arborescence des ressources et leurs attributs associés.
- `proto.expressions` : framework d'évaluation d'expressions. Ce package implémente un petit interprète d'expressions destiné à être utilisé pour évaluer les conditions environnementales et les prédicats de définition de groupes de composants.
- `proto.events` : modèle événementiel. Ce package contient la définition des différents types d'événements utilisés dans le reste du système, ainsi que l'interface `EventListener` associée.
- `proto.groups` : gestion des groupes de composants. Ce package contient les classes implémentant la notion de groupe dynamique de composants.
- `proto` : classes de base et moteur d'adaptation. Ce package contient toutes les classes de bases du système (conteneurs, services, rôles, politiques) ainsi que le lanceur d'application.

Ce système de base n'est pas suffisant en lui-même. Il doit être complété par :

- Le MOP RAM [Bouraçadi-Saâdani et al., 2000] et le compilateur associé [David et al., 2001].
- Une ou plusieurs bibliothèques de services non-fonctionnels, implémentés en tant que métaobjets.
- Une ou plusieurs bibliothèques de sondes logicielles.

Faute de temps (et de connaissances techniques), nous n'avons pas pu implémenter de véritable bibliothèque de sondes logicielles. De même, la bibliothèque de services est actuellement assez limitée. Cependant, tous les services qui ont été développés dans le cadre du projet RAM (en particulier concernant la mobilité de code) peuvent être intégrés directement à notre système. Les autres sous-systèmes sont par contre beaucoup plus complets, et sont détaillés dans les sections suivantes.

5.2 Framework d'observation

Le framework d'observation, contenu dans le package `proto.monitoring`, est relativement simple, et est conforme à ce qui a déjà été décrit dans la section 4.3, page 44. Il est constitué essentiellement de trois classes :

- **ResourceManager** (figure 5.1), qui sert de point d'entrée au framework. Le système crée une unique instance de cette classe au démarrage et lui passe les informations contenues dans le fichier de configuration des ressources. Cette instance crée alors les différents noeuds de l'arborescence et y installe les sondes appropriées, en se servant de ces informations. Pendant l'exécution, cet objet permet d'obtenir une référence vers la ressource racine (méthode `getTopLevelResource()`). Il est aussi capable d'interpréter des chaînes représentant des chemins dans l'arborescence (par exemple `/host/computation/cpu.type`) et de renvoyer la ressource ou l'attribut ainsi désigné (méthodes `findResource()` et `findAttribute()`).
- **MonitoredResource** (figure 5.2), utilisée pour représenter les noeuds de l'arborescence. Chaque ressource, ou catégorie de ressources détectée par le système est représentée par une instance de cette classe ou d'une de ses sous-classes. La classe **MonitoredResource** de base se contente de service de conteneur pour ses attributs et des sous-ressources. Il est possible de créer des sous-classes plus intelligentes, capables par exemple de détecter des sous-ressources particulières ou bien de gérer des attributs synthétiques. En effet, lorsqu'un attribut d'une ressource est modifié, cette ressource en est notifiée ; cela lui permet de notifier à son tour les objets abonnés (modèle de conception *Observer*), ce dont l'implémentation par défaut se charge, mais aussi de mettre à jour les valeurs d'attributs synthétiques.
- **Probe**, utilisée comme classe de base pour implémenter les sondes logicielles. L'implémentation par défaut de **Probe** ne fait rien ; on doit obligatoirement en créer des sous-classes concrètes. **Probe** hérite de `java.lang.Thread`, et possède une référence vers la ressource à laquelle elle est attachée (**MonitoredResource**). Une sonde est chargée dans un premier temps d'ajouter à sa ressource les attributs qu'elle est capable de réifier. Ensuite, au cours de l'exécution, elle doit mettre à jour régulièrement les valeurs de ces attributs récupérés depuis les couches plus basses (système et matériel). Il est possible d'attacher plusieurs sondes à une ressource donnée, et une sonde peut réifier un nombre quelconque d'attributs. La figure 5.3 résume la façon dont ces trois classes sont organisées entre elles.

ResourceManager
+ <code>getTopLevelResource(): MonitoredResource</code> + <code>findResource(path: String): MonitoredResource</code> + <code>findAttribute(path: String): MonitoredResource</code>

FIG. 5.1 – La classe `ResourceManager`

5.3 Évaluation d'expressions

Le package `proto.expressions` contient un petit interprète pour des expressions simples. Ces expressions sont destinées à être utilisées :

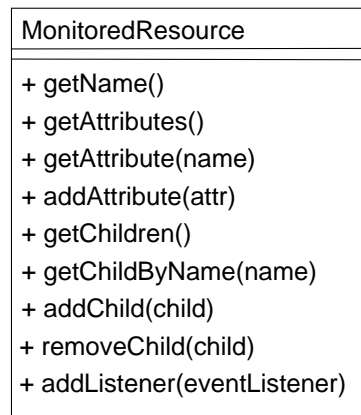


FIG. 5.2 – La classe MonitoredResource

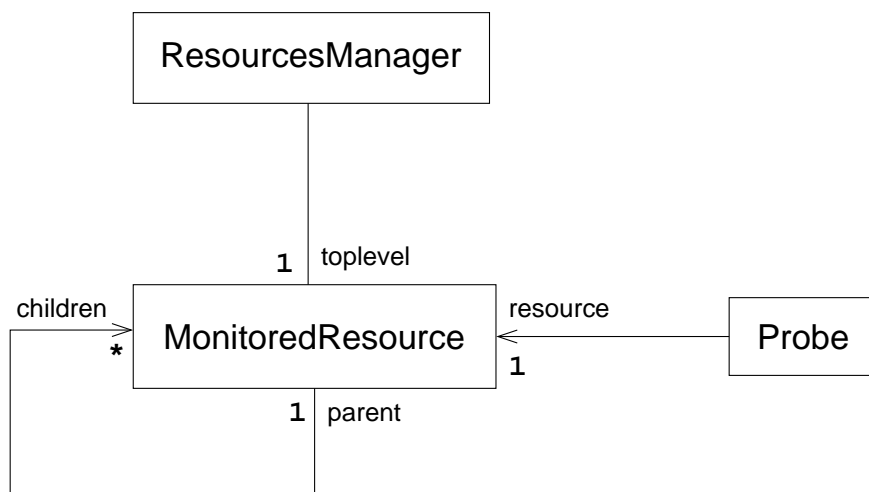


FIG. 5.3 – Les classes du framework d'observation des ressources

- pour représenter les conditions environnementales utilisées dans les règles des politiques système,
- et pour représenter les prédicats utilisés pour définir les groupes de composants fonctionnels.

Dans ces deux cas, les expressions sont de type booléen. Cependant, les sous-expressions peuvent aussi faire intervenir des chaînes de caractères ou des nombres. Au niveau de l'implémentation, une telle expression est représentée par un arbre dont les noeuds sont des instances de sous-classes de la classe **Expression**. Cette super-classe commune définit entre autres les méthodes :

- `evaluate()`, qui renvoie la valeur de la sous-expression correspondante au receveur ;
- `getType()`, qui renvoie le type de cette sous-expression (sous la forme d'une classe Java standard : **String**, **Number** ou **Boolean**),
- `typeCheck()`, appelée automatiquement par le système lors de la construction pour vérifier le type de l'expression (une exception est levée en cas d'erreur de typage).

5.3.1 Expressions terminales

Les expressions sont écrites sous forme de fragments de documents XML. Chaque type d'expression définit un nom d'élément XML correspondant, servant en quelque sorte de constructeur. Les éléments fils dans le document XML correspondent de façon naturelle aux sous-expressions.

Le tableau 5.1 donne la liste des éléments terminaux disponibles pour ces expressions. Le prototype ne supporte actuellement pas les constantes de type booléen. Les attributs étant typés (nombre ou chaîne de caractère), les expressions terminales correspondantes héritent de ce type. Pour ces expressions terminales, leur valeur exacte est définie non pas par un élément XML fils, mais par leur attributs, comme le montre la figure 5.4.

Description	Nom de la classe	Nom de l'élément XML
constante numérique	<code>NumericConstantExpression</code>	<code>number</code>
constante chaîne de caractères	<code>StringConstantExpression</code>	<code>string</code>
valeur d'un attribut	<code>AttributeValueExpression</code>	<code>attribute-value</code>

TAB. 5.1 – Types d'expressions atomiques

```
<number value='999.99' />
<string value='valeur' />
<attribute-value name='nom_attribut_composant' />
<attribute-value name='/chemin/vers/ressource.nom_attribut' />
```

FIG. 5.4 – Syntaxe des éléments terminaux des expressions

5.3.2 Combinateurs d'expressions

Ces éléments terminaux peuvent ensuite être combinés, suivant leurs types, pour former des expressions plus complexes. Ces combinateurs incluent des comparaisons simples, les opérations

booléennes classiques, et les quatre opérations numériques de base. Le framework étant facilement extensible, il est très aisé d'ajouter de nouveaux combinateurs plus sophistiqués ; nous nous sommes limités au minimum pour ce prototype. Le tableau 5.2 donne la liste de ces combinateurs.

Description	Nom de la classe	Nom de l'élément XML
égalité	<code>EqualsExpressions</code>	<code>equals</code>
inférieur à	<code>LessThanExpression</code>	<code>less-than</code>
et logique	<code>AndExpression</code>	<code>and</code>
ou logique	<code>OrExpression</code>	<code>or</code>
non logique	<code>NotExpression</code>	<code>not</code>
addition	<code>AdditionExpression</code>	<code>add</code>
soustraction	<code>SubtractionExpression</code>	<code>subtract</code>
multiplication	<code>ProductExpression</code>	<code>multiply</code>
division	<code>DivisionExpression</code>	<code>divide</code>

TAB. 5.2 – Types de combinateurs d'expressions

5.3.3 Expressions liées et libres

Lorsque les sous-expressions terminales d'une expression font références à des attributs de composants, il est impossible de donner une valeur à cette expression sans spécifier par rapport à quel composant elle doit être évaluée. On parle alors d'expressions libres. Par opposition, les expressions dont la valeur peut être déterminée directement sont dites liées.

Le framework gère les expressions liées simplement, mais nécessite l'ajout de deux classes supplémentaires pour permettre la gestion des expressions libres :

- `UnboundAttributeValueExpression` est utilisée pour représenter les expressions terminales faisant référence à des attributs de composants. Les instances de cette classe ne connaissent que le nom de l'attribut. Si on tente d'évaluer directement une telle expression, sa valeur est `null`. Pour l'évaluer correctement, il faut d'abord la lier à un composant, en utilisant la méthode `setSubject()`.
- `UnboundExpression` est utilisée au plus haut niveau pour encapsuler une expression qui contient une ou plusieurs sous-expressions libres. Elle supporte le même protocole que la classe `UnboundAttributeValueExpression` pour se lier à un composant particulier, et propage ce message (`setSubject()`) à toutes ses sous-expressions libres avant de d'évaluer.

5.3.4 Évaluation incrémentale d'expressions

Dans un système en cours d'exécution, il peut y avoir un nombre important d'expressions. Or, ces expressions sont dynamiques et leurs valeurs doivent toujours être à jour par rapport aux conditions reflétées par l'arborescence de ressources ou par les attributs de composants. Pour des raisons de performance, ces expressions sont donc évaluées de façon incrémentale, de façon à minimiser le temps de calcul.

Les seuls éléments d'une expression dont la valeur peut changer au cours du temps sont les références à des attributs (de ressources ou de composants). Les attributs implémentant le modèle de conception *Observer*, les expressions qui font référence à un attribut s'abonnent auprès de cet attribut. Lorsque la valeur d'un attribut change, toutes les sous-expressions du système qui y

font référence en sont notifiées. Celles-ci notifient à leur tour leur expression parente, afin qu'elle recalcule sa valeur. Si cette valeur change (ce qui n'est pas toujours le cas), la notification est propagé au niveau supérieur dans l'expression, jusqu'à atteindre éventuellement la racine.

Si la valeur de l'expression complète est modifiée, celle-ci doit pouvoir prévenir le reste du système (cette modification peut par exemple déclencher l'attachement d'un nouveau service). Pour cela, toutes les expressions sont encapsulées dans un objet de type `Condition`. Il s'agit d'un type particulier d'expression (uniquement booléenne), dont la valeur reflète celle de son unique sous-expression. La seule valeur ajoutée par cette classe est la possibilité de prévenir le reste du système lorsque la valeur de son enfant (et donc de l'expression globale) change. Encore une fois, cela est réalisé en utilisant le modèle de conception *Observer* associé à une interface nommée `ConditionListener`. La figure 5.5 montre comment les notifications se propagent au sein d'une expression.

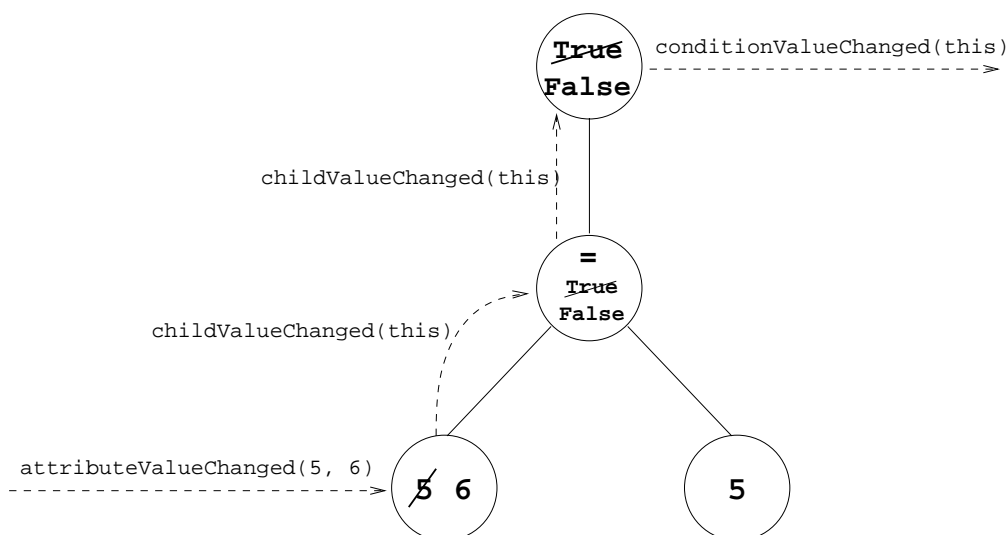


FIG. 5.5 – Propagation des notifications lors de la modification d'un attribut

5.4 Groupes de composants

La gestion des groupes dynamiques de composants est réalisée par le package `proto.groups`. Tout comme dans le cas des ressources, un point d'entrée est fourni par une classe `GroupsManager`. Une unique instance de cette classe est créée automatiquement au démarrage du système. Cet objet permet d'avoir accès au groupe principal, nommé `all`, qui contient tous les composants du système (grâce à la méthode `getRootGroup()`), et permet aussi de retrouver un groupe quelconque par son nom, qui doit être unique dans le système (méthode `getGroup()`) ou bien la liste complète de tous les groupes existants (méthode `getAllGroups()`).

Les classes utilisées pour représenter les groupes eux-mêmes sont organisées de façon un peu particulière :

- L'interface `Group` définit les opérations possibles sur un groupe :
 - obtenir son nom (`getName()`),
 - obtenir l'ensemble de ses membres actuels (`getMembers()`),

- abonner ou désabonner un objet devant être notifié de tout changement dans la composition du groupe (`addGroupListener()` et `removeGroupListener()`).
- La classe abstraite `AbstractGroup` qui implémente quelques fonctionnalités partagées par les deux implémentations de l'interface `Group` (à savoir la gestion du modèle de conception *Observer*).
- La classe concrète `ExplicitGroup`, sous-classe de `AbstractGroup`. Cette classe représente un groupe de composants dont les membres sont ajoutés ou supprimés explicitement par des appels de méthodes (`add()` et `remove()`). Il s'agit donc tout simplement d'une collection d'objets classique. En pratique, une seule instance de cette classe sera créée, pour représenter le groupe `all`. En effet, il s'agit du seul groupe qui n'est pas défini de la même façon que les autres. L'ajout ou le retrait de membres se fait automatiquement par le système.
- Enfin, la classe concrète `ImplicitGroup`, utilisée pour représenter tous les groupes définis par l'utilisateur dans les politiques applicatives. Un tel groupe est défini en fonction d'un super-groupe et d'un prédicat. Dès sa création, une instance de cette classe s'abonne auprès de son groupe parent pour être notifié de tout changement dans sa composition. Lorsqu'une telle notification lui parvient, il modifie éventuellement sa propre composition, et passe la notification à ses propres sous-groupes.

Le digramme UML de la figure 5.6 résume l'organisation de ces différentes classes.

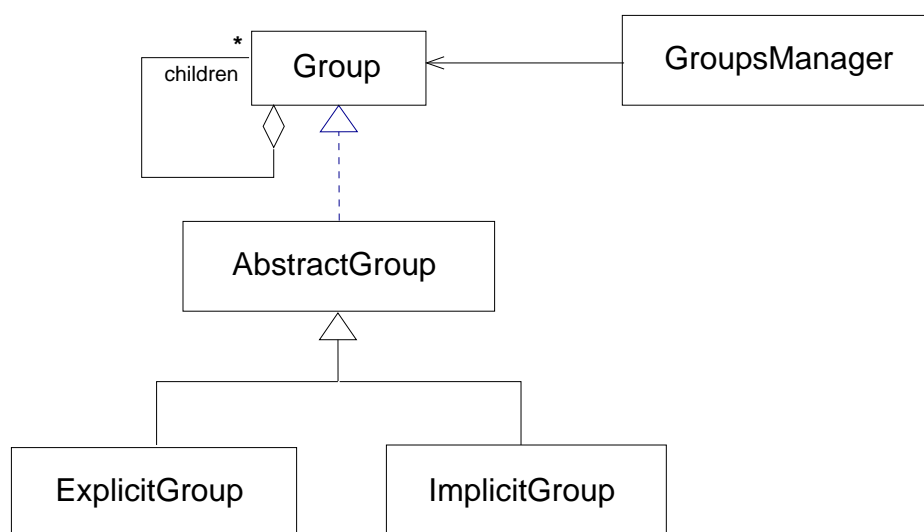


FIG. 5.6 – Les classes du package `proto.groups`

5.5 Services, rôles et métaobjets

Cette section présente les aspects techniques de l'implémentation des services et des rôles à l'aide de métaobjets.

5.5.1 Le MOP

Le MOP retenu pour notre prototype est celui de RAM (voir la section 2.1.7 de l'état de l'art, page 13 ou [Bouraqadi-Saâdani et al., 2000] pour une description plus détaillée). Il s'agit d'un des MOP Java les plus flexibles et dynamiques, et qui ne nécessite pas de machine virtuelle modifiée. De plus, RAM a été réalisé à l'École des Mines de Nantes, et il était donc plus simple que d'autres à appréhender, puisque nous avons accès facilement à toute l'expertise nécessaire.

L'implémentation standard de RAM utilise la bibliothèque Javassist [Chiba, 2000] pour modifier directement le byte-code des classes de base lors de leur chargement dans la machine virtuelle. Ces transformations consistent à insérer des *hooks* dans le code de base. Ces hooks sont en fait des petits fragments de code qui implémentent une indirection vers le niveau méta. Une fois que l'on est passé au niveau méta, les métaobjets sont des objets Java normaux, qui ne nécessitent pas de support technologique particulier. Leur seule différence par rapport à des objets de base est que les messages qu'ils comprennent sont d'un niveau d'abstraction plus élevé.

Dans le cadre d'une expérimentation avec l'outil de programmation par aspects AspectJ [Kiczales et al., 2001], nous avons développé un petit outil utilisant AspectJ pour opérer l'insertion des hooks dans un code de base Java et créer une indirection vers les classes du MOP de RAM [David et al., 2001]. Le système résultant est fonctionnellement identique à l'implémentation originale de RAM, la seule différence étant que la transformation de code se fait sur le code source plutôt que sur le byte-code. La partie runtime du MOP RAM n'a subi que des adaptations mineures pour la découpler complètement du système de transformation de code¹.

Notre expérimentation ayant été un succès, c'est cette version alternative de RAM que nous avons utilisé pour notre prototype.

5.5.2 Implémentation des services et des rôles

L'organisation des classes de base représentant les services et les rôles a déjà été présentée dans la section 4.2.2, page 41 et en particulier avec la figure 4.2. L'implémentation du prototype est très proche. Les classes entrant en jeu sont :

- La classe `ServicesManager`, utilisée pour enregistrer l'ensemble des services connus du système et pour permettre de retrouver un service par son nom (`getServiceByName()`) ou l'ensemble de tous les services (`getAllServices()`);
- La classe `Service`, qui représente un service abstrait et qui connaît l'ensemble des fournisseurs concrets de ce service. C'est cette classe qui se charge de gérer les contraintes d'attachement d'un rôle à un composant :
 - La méthode `attachTo(aContainer, roleName, params)` se charge de trouver une implémentation du rôle en question supportant les paramètres spécifiés. Si un rôle concret correspondant est déjà attaché au composant et est capable de se reconfigurer avec les nouveaux paramètres, cette reconfiguration est effectuée. Sinon, le service recherche une autre implémentation concrète supportant les paramètres et l'installe. Le prototype actuel ne fait rien en cas d'échec, et n'est pas capable de déterminer parmi plusieurs implémentations supportant les paramètres laquelle est "la meilleure".
 - La méthode `detachFrom(aContainer, roleName)` est beaucoup plus simple : elle se contente de détacher le métaobjet correspondant du conteneur.

¹Cette séparation explicite entre le mécanisme de transformation de code et les métaobjets utilisés à l'exécution peut être rapprochée de la philosophie du système Reflex [Tanter et al., 2001].

-
- La classe `ServiceProvider` est une classe abstraite très simple servant de base à l'implémentation de services concrets.
 - Enfin, l'interface `RoleProvider` est minimale. Elle est utilisée pour adapter des métaobjets existants (développés indépendamment dans le projet RAM) en tant que rôle. En dehors des méthodes `getService()` et `getRoleName()` utilisées pour faire le lien entre un métaobjet et le rôle qu'il implémente, l'interface définit une méthode `configure(parameters)` utilisée pour demander à un rôle de se reconfigurer dynamiquement. La méthode renvoie un booléen indiquant si la reconfiguration s'est bien faite.

5.6 Gestion des politiques

Le cœur du système, qui interprète les politiques système et applicatives, se trouve dans le package `proto`.

À chacun des groupes définis par la politique applicative, on associe une instance de la classe `GroupBindings`. Cet objet est chargé de gérer l'association entre les membres du groupe et les règles d'adaptation qui y sont attachées. En particulier, cet objet est notifié par le groupe de tout changement dans sa composition, afin qu'il puisse réagir en activant ou désactivant les services et rôles appropriés aux composants entrants ou sortants. Les liaisons avec ces services sont représentées par des instances de la classe `ServiceBinding`, qui encapsule

- une référence vers un service ;
- un nom de rôle ;
- et des paramètres d'attachement.

En fait, par rapport aux règles d'adaptation telles qu'elles sont exprimées dans les politiques système, les objets `ServiceBinding` correspondent au contenu de la clause `ensure` (cf. figure 4.10, page 55). La condition environnementale qui indique quand cette liaison doit être activée est gérée par l'objet `GroupBindings`. Au final, `GroupBindings` est chargé d'appliquer des liaisons conditionnelles (les `ServiceBinding` et leur condition associée) aux membres d'un groupe.

Dynamiquement, les deux éléments qui peuvent évoluer sont :

- le contenu du groupe. Si un nouveau composant entre dans le groupe, on lui applique tous les `ServiceBindings` actifs à ce moment là. Symétriquement, si un composant sort du groupe, on lui retire les services/rôles correspondant aux `ServiceBindings` actifs.
- les conditions environnementales, et donc l'ensemble de `ServiceBindings` actifs. De manière duale, si un `ServiceBinding` inactif est activé (sa condition associée devient vraie), on l'applique à tous les membres actuels du groupe. À l'inverse, lorsqu'un `ServiceBinding` est désactivé, on annule son effet sur les membres courant.

Du point de vue de la concurrence, la mise à jour du contenu des groupes se fait de façon synchrone par rapport au code de l'application (ou en tout par rapport au thread qui a modifié un attribut de composant, entraînant son entrée ou sa sortie du groupe). Les conditions environnementales évoluent quant à elles de façon asynchrone par rapport au code applicatif. Le prototype actuel n'est pas capable de gérer les conflits ou incohérences éventuelles que cela peut entraîner.

Le diagramme UML de la figure 5.7 résume la situation.

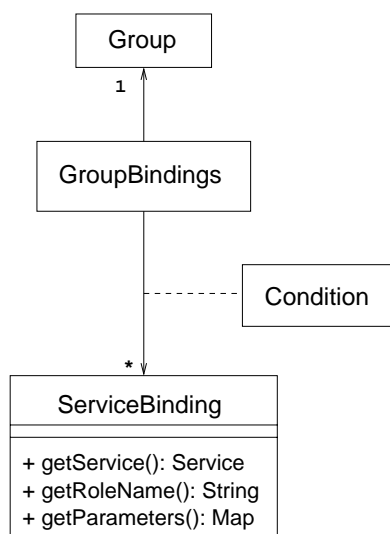


FIG. 5.7 – Classes interprétant les politiques

5.7 Le lanceur

Le lanceur est contenu dans la classe `proto.Launcher`. C'est le point d'entrée de la plateforme. Il est invoqué en lui passant en paramètre le nom d'un répertoire contenant les fichiers de configuration et les politiques, ainsi que le nom et les paramètres du programme à exécuter :

```
% java proto.Launcher répertoire ma.classe.Principale paramètres
```

Le lanceur commence par initialiser le gestionnaire de ressources avant de charger un certain nombre de fichiers XML qui doivent se trouver dans le répertoire spécifié en paramètre :

- `services.xml` : Ce fichier contient la liste de tous les services connus et de toutes les classes qui les implémentent (sous-classes de `ServiceProvider`). La syntaxe est la suivante :

```
<?xml version='1.0' ?>
<services>
  <service name="nom.service"/>
  <service name="nom.autre.service"/>
  .
  <provider class="nom.complet.de.Classe"/>
  <provider class="nom.complet.autre.Classe"/>
  .
</service>
```

Ce fichier n'indique pas explicitement quel service est implémenté par une classe fournisseur (*provider*). En fait, le système interroge directement une instance de chaque classe spécifiée pour connaître quel service elle implémente.

- `components.xml` : Ce fichier contient la liste de toutes les classes réflexives que le système doit prendre en charge en tant que composants. À terme, il devrait aussi contenir la liste des champs de ces classes qui doivent être automatiquement exportés en tant qu'attributs de composants (cet aspect n'est pas implémenté dans le prototype). La syntaxe est la suivante :

```
<?xml version='1.0' ?>
```

```

<components>
  <component class='com.mycompany.Account>
    <exports field='balance' />
  </component>
  .
  .
</components>

```

- **system.xml** : Ce fichier contient l'ensemble des politiques systèmes utilisées. Leur syntaxe a déjà été présentée (cf figure 4.10, page 55). À ce moment de l'initialisation, le système ne connaît pas encore toutes les informations nécessaires pour créer les objets **ServiceBinding** et leur conditions associées, qui vont représenter les règles d'adaptation des politiques système. Ces informations sont donc stockées dans des structures de données temporaires (les classes **SystemPolicy** et **SystemPolicy.Rule**).
- **application.xml** : Ce fichier contient la politique applicative, qui définit les groupes de composants et leurs associations avec les politiques système. Les instances de **Group** et **GroupBindings** correspondantes sont créées, et les informations stockées dans les instances de **SystemPolicy** sont finalement utilisées pour créer les **ServiceBindings** correspondants, terminant ainsi l'initialisation.

Le prototype actuel n'ayant pas de bibliothèque de sondes, le système de détection de ressources et le fichier de description associé, décrits à la section 4.3.1, page 50 ne sont pas implémentés.

Une fois cette initialisation terminée, il ne reste plus qu'à lancer l'exécution de l'application utilisateur, en invoquant la méthode statique **main(String[])** de la classe dont le nom a été passé en paramètre (avec les bons paramètres bien sûr).

5.8 Exemple d'utilisation

Cette section décrit l'utilisation concrète du système sur un petit exemple de démonstration. Le but est de réaliser un système de commerce électronique. Pour notre démonstration, nous nous contentons de la consultation d'un catalogue à distance (architecture client/serveur).

La tâche du programmeur d'application va consister à :

- définir et implémenter les objets métier ;
- implémenter le programme serveur gérant le catalogue ;
- implémenter un client graphique capable de consulter le catalogue.

5.8.1 Le code métier

Les classes correspondant aux objets métier définis par le programmeur sont :

- **Item** : un type d'article à vendre, à un prix donné.
- **Stock** : une collection d'**Items**, avec pour chacun d'entre eux la quantité disponible.
- **Account** : un compte client, contenant des informations telles que son nom, son adresse et ses coordonnées bancaires.
- **Order** : une commande d'un client pour un certain nombre d'articles (**Items**).
- **Shop** : un magasin disposant d'un **Stock** d'articles et d'un fichier client (**Account** et **Order**).

Le code de ces classes est purement fonctionnel. Il ne contient aucun code traitant par exemple de la sauvegarde de ces objets dans une base de données. Ces différentes classes vont devenir les

classes de composants de l'application finale ; elles doivent être compilées avec l'outil fourni par notre système, pour les rendre réflexives et donc manipulables par notre MOP.

Viennent ensuite les deux programmes nécessaires à notre architecture client/serveur, contenus dans les classes Java `Admin` et `Client`. Eux aussi sont réalisés par le programmeur d'application. Leur structure est similaire et extrêmement simple. Tous deux définissent d'abord le code de l'interface graphique (réalisée avec le framework Swing). Cette interface est capable de manipuler une instance de la classe `Shop`. Dans le cas de l'interface d'administration, les manipulations possibles vont consister à ajouter ou supprimer des éléments du catalogue. Dans le cas du client, les manipulations se limiteront à la consultation. Encore une fois, ce code est purement fonctionnel ; par exemple, le code complet de la méthode `main(String[])` de la classe `Admin` se limite à :

```
public static void main(String[] args) {
    Shop shop = new Shop();
    new Admin(shop).show();
}
```

Le reste du code de cette classe est exclusivement lié à la création et à la gestion de l'interface graphique. La méthode `main(String[])` de la classe `Client` est quasi-identique, mais crée une instance de la classe `Client` au lieu de `Admin`.

À ce moment du développement, les deux programmes sont utilisables de façon autonome. L'interface graphique d'administration peut être testée et déboguée indépendamment de tout code non-fonctionnel. On peut l'utiliser pour créer un catalogue, mais celui-ci n'est pas encore accessible par un autre programme et son contenu n'est pas conservé lorsque l'on quitte le programme. Dans le cas de l'interface client, ses fonctionnalités sont beaucoup plus limitées à cet instant : elle démarre avec un magasin vide et ne permet pas d'ajouter de nouveaux articles.

Bien que le système ne soit pas encore opérationnel, *plus aucune modification de code source ni recompilation ne seront nécessaires à partir de cet instant*. Le travail de programmation est essentiellement terminé, et s'est limité à du code métier ; il ne reste plus qu'à adapter ces composants fonctionnels à l'utilisation prévue. Pour obtenir un système réellement utilisable, nous avons besoin de deux services :

- un service de persistance du côté du serveur, pour sauvegarder automatiquement le contenu du catalogue et du fichier client entre deux invocations du serveur ;
- un service de distribution pour faire en sorte que le composant `Shop` manipulé par l'interface du client soit en fait celui exporté par le serveur.

5.8.2 Les fichiers de configuration communs

Nous avons pour cela besoin de deux ensembles de fichiers de configuration, l'un pour le serveur, l'autre pour le client. Certains des fichiers de configuration de la plate-forme peuvent être partagés :

- `components.xml`, qui donne la liste des classes Java qui sont en fait des composants et doivent être gérés par la plate-forme. Dans notre exemple, son contenu sera :

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<components>
  <component class="Account"/>
  <component class="Item"/>
  <component class="Order"/>
```

```

    <component class="Shop"/>
    <component class="Stock"/>
  </components>
- services.xml, qui définit les services utilisés et les classes fournissant ces services. Son
  contenu exact dans notre cas est :
  <?xml version='1.0' encoding='ISO-8859-1' ?>
  <services>
    <service name="communication.rpc"/>
    <service name="persistency"/>

    <provider class="services.RMIProvider"/>
    <provider class="services.PersistencyProvider"/>
  </services>
  En fait, le service de persistance n'étant réellement utilisé que du côté serveur, il pourrait
  être retiré du fichier de configuration du client.
  Reste à écrire les politiques système et applicatives.

```

5.8.3 Les politiques système et applicatives

Politiques du serveur

Dans le cas du serveur, on veut rendre les composants à la fois persistants et distribués. Pour activer le service de persistance, on attache directement aux instances de la classe `Shop` le rôle unique du service de persistance, sans passer par une politique système. Le composant `Shop` étant en quelque sorte la racine du graphe de composants (tous les autres composants y sont reliés directement ou indirectement), cela suffit pour rendre tous les composants persistants.

Pour la distribution, on passe cette fois par un politique système. Faute de temps, nous n'avons pas pu développer de sondes ou de services très sophistiqués. La politique système se résume donc à sa plus simple expression, *i.e.* une condition toujours vraie et une seule action. En pratique, la réalisation de la distribution devrait s'adapter aux conditions, en ajoutant des services de cache ou d'invocation asynchrone par exemple lorsque la qualité de la connexion réseau baisse. Les figures 5.8 et 5.9 donnent le code exact respectivement de la politique applicative et de la politique système.

Politiques du client

Du côté du client, la situation est plus simple puisque l'on n'a pas besoin de la persistance. Il suffit d'attacher le service de distribution aux composants, et plus spécifiquement le rôle symétrique de celui utilisé pour le serveur. Encore une fois, la politique système est simpliste par rapport à ce qu'elle devrait être dans un système finalisé. Les figures 5.10 et 5.11 donnent le code de la politique applicative et de la politique système.

5.8.4 Exécution des programmes

Une fois tous ces fichiers de configuration créés et déployés (par exemple dans les répertoires `config-serveur` et `config-client`), il ne reste plus qu'à lancer les deux programmes. Commençons par le serveur :

```
% java proto.Launcher config-serveur Admin
```

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<application-policy>
  <group name="racines-persistence">
    <select from="all">
      <equals>
        <attribute-value name="className"/>
        <string value="Shop"/>
      </equals>
    </select>
    <bind service="persistency" role="main"/>
  </group>

  <group name="remote-accessible">
    <select from="all">
      <equals>
        <attribute-value name="className"/>
        <string value="Shop"/>
      </equals>
    </select>
    <bind policy='distribution-server'/>
  </group>
</application-policy>
```

FIG. 5.8 – Politique applicative utilisée pour le serveur

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<config>
  <system-policy name="distribution-server">
    <rule>
      <when>
        <equals><number value='1'/><number value='1'/></equals>
      </when>
      <ensure>
        <attached service="communication.rpc" role="server">
          <parameter name="server" value="localhost"/>
        </attached>
      </ensure>
    </rule>
  </system-policy>
</config>
```

FIG. 5.9 – Politique système utilisée pour le serveur

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<application-policy>
  <group name="remote-components">
    <select from="all">
      <equals>
        <property-value name="className"/>
        <string value="Shop"/>
      </equals>
    </select>
    <bind policy="-distribution-client"/>
  </group>
</application-policy>
```

FIG. 5.10 – Politique applicative utilisée pour le client

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<config>
  <system-policy name="distribution-client">
    <rule>
      <when>
        <equals><number value='1' /><number value='1' /></equals>
      </when>
      <ensure>
        <attached service="communication.rpc" role="proxy">
          <parameter name="server" value="localhost"/>
        </attached>
      </ensure>
    </rule>
  </system-policy>
</config>
```

FIG. 5.11 – Politique système utilisée pour le client

Nous obtenons l'interface graphique d'administration normale. Si nous ajoutons des articles au catalogue (et donc modifions le composant **Shop** et ses sous-composants) avant de quitter l'application puis de la relancer, les nouveaux articles sont toujours présents. Le service de persistance s'est chargé de les sauvegarder sur disque puis de les recharger de façon transparente.

Lançons maintenant le client :

```
% java proto.Launcher config-client Client
```

Les articles ajoutés au catalogue dans l'interface d'administration sont automatiquement disponibles dans l'interface du client. Les rôles **server** et **proxy** du service de distribution (et en particulier l'implémentation basée sur RMI dans notre cas) se chargent automatiquement d'exporter les composants du serveur et d'y accéder depuis le client.

5.8.5 Conclusion

Notre système a donc permis au programmeur d'application de développer le code métier sans se préoccuper outre mesure de l'utilisation des techniques telles que la persistance. Il a ensuite exprimé toutes les "contraintes" non fonctionnelles de ce code métier séparément, sous la forme d'une politique applicative, stockée dans un fichier unique. Le travail du programmeur d'application s'arrête normalement là.

De façon complètement indépendante, des programmeurs spécialistes de domaines techniques tels que la distribution ont développé des services de qualité. Ces services sont complètement réutilisables, et ne dépendent pas d'une application particulière. D'autres programmeurs ont développé des sondes logicielles sophistiquées pour diverses plates-formes. Ces bibliothèques de services et de sondes peuvent être vendues séparément.

Au moment du déploiement, le déployeur fait le lien entre ces divers éléments, en choisissant en fonction de la nature de l'hôte une bibliothèque de sondes logicielles et des implémentations concrètes des services nécessaires.

Cette architecture permet donc de bien séparer les différents rôles et de maximiser la réutilisabilité des composants qui ne dépendent pas explicitement d'une application particulière.

Chapitre 6

Conclusion

6.1 Apport de ce travail

Par rapport aux différentes solutions proposées dans les systèmes étudiés au chapitre 2, notre architecture est une des seules à prendre en compte tous les besoins d'un système adaptable (résumés par les trois fonctionnalités : observation, décision, action).

De nombreux systèmes permettent de modifier dynamiquement (ou non) le comportement d'une application (R-RIO, *Adaptive Components*, LEAD++) ou du middleware sous-jacent (dynamicTAO). Certains permettent en plus d'observer l'environnement d'exécution (DART, Molène), mais très peu d'entre eux (OpenORB, Olan) supportent directement la possibilité de relier les deux, c'est-à-dire de rendre les modifications dépendantes des évolutions de l'environnement, ce qui fait partie de la définition même de l'adaptabilité dynamique. Bien sûr, il est toujours possible de programmer cette liaison "à la main", mais aucun de ces systèmes ne permet le même degré d'abstraction et d'indépendance par rapport au code fonctionnel que nos politiques d'adaptation.

Les apports principaux de notre architecture sont donc d'une part l'intégration des différents aspects de l'adaptabilité dans une infrastructure unique, et d'autre part les politiques d'adaptation XML et le moteur d'adaptation qui les interprète. Enfin, malgré la bonne intégration des différents éléments de notre infrastructure, les interfaces qui les relient sont finalement assez lâches. C'est particulièrement vrai dans le cas du MOP utilisé, dont les fonctionnalités ne sont utilisées que de façon relativement abstraite, par l'intermédiaire des conteneurs et des couples services/rôles. On peut parfaitement imaginer de réutiliser la quasi-totalité de l'infrastructure avec un autre MOP, comme par exemple Reflex ou ProActive.

6.2 Questions ouvertes

De nombreuses questions ont été soulevées pendant ce stage, qui n'ont pas pu être abordées sérieusement faute de temps.

Parmi celles-ci, la plus importante est probablement celle concernant le fonctionnement de notre architecture dans un cadre distribué. Si plusieurs instances du système doivent être amenées à coopérer pour implémenter une application répartie, il faut pouvoir s'assurer d'un certain degré de cohérence au niveau global. Cela suppose entre autres que les politiques et services utilisés par ces différentes instances doivent être d'une certaine manière "compatibles", et que le système

est capable de les utiliser de façon coordonnée. Des échanges d'informations doivent aussi avoir lieu entre les systèmes, en particulier en ce qui concerne les ressources disponibles et utilisées sur chaque site (par exemple pour permettre la répartition de charge). Que signifie exactement "politiques compatibles" ? Quelles informations doivent être échangées entre les sites, comment les échanger et comment en tenir compte ?

Un autre problème important qui n'est pas pris en compte actuellement est la problématique de la composition des services. Les travaux effectués dans le domaine de la métaprogrammation (composition de métaobjets) et de la programmation par aspects (tissage de multiples aspects) pourraient sans doute être un point de départ intéressant pour résoudre ce problème.

Enfin, il manque au système actuel, qui n'est qu'une première ébauche, une sémantique plus clairement définie. En particulier en ce qui concerne la gestion de la concurrence dans le moteur d'adaptation (synchronisation des actions du moteur par rapport à l'application). Des approches plus formelles du génie logiciel (langages de description d'architectures, langages temps-réels comme Esterel) peuvent sans doute être utilisées ici.

6.3 Travaux futurs

À court terme, nos objectifs sont multiples. Un premier travail serait d'enrichir le système d'observation des ressources en développant une bibliothèque de sondes, ainsi qu'un système permettant la découverte dynamique des ressources. L'infrastructure nécessaire existe déjà, mais nous n'avons pas eu le temps de développer des composants concrets.

Au cours de nos recherches, il nous est apparu à de nombreuses reprises que la séparation que notre système opère entre les composants fonctionnels et les divers services était d'une certaine façon liée à la problématique plus générale dite de la *separation of concerns* et à la programmation par aspects. On peut en effet considérer que les composants fonctionnels constituent "l'aspect primaire" d'une application, que l'on modifie en lui attachant des services non-fonctionnels, de façon similaire au tissage d'aspects "secondaires". Il serait intéressant de chercher à expliciter cette relation, en particulier en ce qui concerne les rapports entre métaprogrammation et programmation par aspects.

En écrivant des services concrets (comme la persistance), pour valider notre prototype, il nous est apparu que l'interface proposée par un MOP est finalement d'assez bas niveau. Il est parfois difficile d'implémenter des services de haut niveau en utilisant des concepts trop proches des mécanismes du langage utilisé (envoi de message, accès aux champs...). Nous pensons donc remplacer progressivement le modèle actuel à base d'objets par un véritable modèle de composants, offrant des abstractions de plus haut niveau (gestion explicite du cycle de vie par exemple), sans remettre en cause l'utilisation de techniques réflexives.

Comme nous l'avons déjà dit plus haut, il devrait être relativement aisé de réutiliser notre infrastructure avec un MOP différent de RAM. Nous envisageons donc de valider l'architecture en la réutilisant avec Reflex ou JOnAS [ObjectWeb,].

À plus long terme, nous devons définir clairement le fonctionnement de notre infrastructure dans un contexte réparti. Cela implique l'existence de plusieurs instances de notre système, qui doivent communiquer et coopérer pour constituer un tout cohérent. Ceci nécessitera probablement une meilleure formalisation du fonctionnement de l'infrastructure, ce qui constitue notre deuxième objectif à long terme.

Enfin, les politiques d'adaptation telles qu'elles se présentent actuellement (en particulier les

politiques système), sont écrites d'une façon trop impératives. Des politiques plus déclaratives, s'inspirant par exemple des systèmes de contraintes et prenant en compte des critères de qualité de services (QoS), permettraient un niveau d'abstraction encore plus élevé. Une telle évolution implique des modifications dans la nature du moteur d'adaptation, qui devra être beaucoup plus "intelligent", mais ouvre des perspectives intéressantes en donnant plus de contrôle au système sur la façon dont les politiques sont interprétées.

Bibliographie

- [Amano and Watanabe, 1999] Amano, N. and Watanabe, T. (1999). An approach for constructing dynamically adaptable component-based software systems using LEAD++. In *OOPSLA'99 International Workshop on Object Oriented Reflection and Software Engineering (OORaSE'99)*, pages 1–16.
- [Andersen et al., 2000] Andersen, A., Blair, G. S., and Eliassen, F. (2000). OOPP : A reflective component-based middleware. In *Proceedings of NIK*.
- [André and Segarra, 2000] André, F. and Segarra, M.-T. (2000). A generic approach to satisfy adaptability needs in mobile environments. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences (HICSS33)*.
- [Balter et al., 1998] Balter, R., Bellissard, L., Boyer, F., Riveill, M., and Vion-Dury, J.-Y. (1998). Architecturing and configuring distributed applications with Olan. In *Proceedings Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*.
- [Blair and Coulson, 1997] Blair, G. S. and Coulson, G. (1997). The case for reflective middleware. Technical report, Distributed Multimedia Research Group, Department of Computing, Lancaster University.
- [Blair et al., 2000] Blair, G. S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran, H., Parlavantzas, N., and Saikoski, K. (2000). A principled approach to supporting adaptation in distributed mobile environments. In *Int. Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'2000)*.
- [Boinot et al., 2000] Boinot, P., Marlet, R., Noyé, J., Muller, G., and Consel, C. (2000). A declarative approach for designing and developing adaptive components. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 111–119. IEEE Computer Society.
- [Bouraçadi-Saâdani et al., 2000] Bouraçadi-Saâdani, M. N., Douence, R., Ledoux, T., and Südholt, M. (2000). Une infrastructure réflexive pour la mobilité forte en Java. Rapport interne, École des Mines de Nantes. 3ème livrable de la CTI FT R&D Reflection for Adaptable Mobility.
- [Boyer and Charra, 2000] Boyer, F. and Charra, O. (2000). Utilisation de la réflexivité dans les plate-formes adaptables pour applications réparties. In *Notere 2000*.
- [Bruneton and Riveill, 2000] Bruneton, E. and Riveill, M. (2000). JavaPod : une plate-forme à composants adaptable et extensible. Rapport de recherche 3850, INRIA.
- [Bäumer et al., 2000] Bäumer, D., Riehle, D., Siberski, W., and Wulf, M. (2000). Role object. In Harrison, N., Foote, B., and Rohnert, H., editors, *Pattern Languages of Program Design 4*, pages 15–32. Addison-Wesley. Chapter 2.

-
- [Capra et al., 2001] Capra, L., Emmerich, W., and Mascolo, C. (2001). Reflective middleware solutions for context-aware applications. In *Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, LNCS. Springer-Verlag.
- [Caromel et al., 1998] Caromel, D., Klauser, W., and Vayssiere, J. (1998). Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13) :1043–1061.
- [Chiba, 2000] Chiba, S. (2000). Load-time structural reflection in Java. In *ECOOP 2000 – Object-Oriented Programming*, volume 1850 of LNCS, pages 313–336. Springer-Verlag.
- [Cointe and Ledoux, 2000] Cointe, P. and Ledoux, T. (2000). Pour des architectures logicielles ouvertes et adaptables. la réflexion : pourquoi et comment ? In *Séminaire Systèmes distribués et Connaissances*.
- [David et al., 2001] David, P.-C., Ledoux, T., and Bouraqadi-Saâdani, M. N. (2001). Two-step weaving with reflection using AspectJ. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*.
- [Dowling et al., 1999] Dowling, J., Schafer, T., Cahill, V., Haraszti, P., and Redmond, B. (1999). Using reflection to support dynamic adaptation of system software : A case study driven evaluation. In *OORaSE*, pages 169–188.
- [Efstratiou et al., 2001] Efstratiou, C., Cheverst, K., Davies, N., and Friday, A. (2001). Architectural requirements for the effective support of adaptive mobile applications. In *Proceedings of Int Conf on MDM2001 (Mobile Data Management)*.
- [Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns*. Addison-Wesley.
- [Gowing and Cahill, 1996] Gowing, B. and Cahill, V. (1996). Meta-objet protocols for C++ : The Iguana approach. In *Proceedings of Reflection 1996*.
- [Hürsch and Lopes, 1995] Hürsch, W. and Lopes, C. V. (1995). Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, Massachusetts.
- [Kiczales, 1992] Kiczales, G. (1992). Towards a new model of abstraction in the engineering of software. In *IMSA'92 Proceedings (Workshop on Reflection and Meta-level architectures)*.
- [Kiczales, 1996] Kiczales, G. (1996). Beyond the black box : Open implementation. *IEEE Software*, 13(1).
- [Kiczales et al., 1991] Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The art of the Meta-Object Protocol*. MIT Press.
- [Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In Knudsen, J. L., editor, *ECOOP 2001*, volume 2072 of LNCS, pages 327–353. Springer-Verlag.
- [Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of LNCS. Springer-Verlag.
- [Kon and Campbell, 1999] Kon, F. and Campbell, R. H. (1999). Supporting automatic configuration of component-based distributed systems. In *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*.

-
- [Kon et al., 2000] Kon, F., Romàn, M., Liu, P., Mao, J., Yamane, T., Magalhães, L. C., and Campbell, R. H. (2000). Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proceedings of Middleware 2000*, volume 1795 of *LNCS*, pages 121–143. Springer-Verlag.
- [Kon et al., 1998] Kon, F., Singhai, A., Campbell, R. H., Carvalho, D., Moore, R., and Ballesteros, F. (1998). 2K : A reflective, component-based operating system for rapidly changing environments. In *Workshop on Reflective Object-Oriented Programming and Systems at ECOOP'98*.
- [Kon et al., 2001] Kon, F., Yamane, T., Hess, C. K., Campbell, R. H., and Mickunas, M. D. (2001). Dynamic resource management and automatic configuration of distributed component systems. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*.
- [Ledoux, 1999] Ledoux, T. (1999). OpenCorba : a reflective open broker. In *Reflection'99*, volume 1616 of *LNCS*. Springer-Verlag.
- [Loques et al., 2000] Loques, O., Leite, J., Lobosco, M., and Sztajnberg, A. (2000). On the integration of configuration and meta-level programming approaches. In *Object-Oriented Reflection and Software Engineering*, volume 1826 of *LNCS*, pages 191–210. Springer-Verlag.
- [Maes, 1987] Maes, P. (1987). Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*.
- [Malenfant et al., 2001] Malenfant, J., Segarra, M.-T., and André, F. (2001). Dynamic adaptability : the Molène experiment. In *Reflection 2001*.
- [McAffer, 1995] McAffer, J. (1995). Meta-level programming with CodA. In Olthoff, W., editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 190–214. Springer-Verlag.
- [Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on software engineering*, 26(1).
- [ObjectWeb,] ObjectWeb. JOnAS : Java Open Application Server. <http://www.objectweb.org/jonas/jonasHomePage.htm>.
- [Okamura and Ishikawa, 1994] Okamura, H. and Ishikawa, Y. (1994). Object location control using meta-level programming. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, volume 821 of *LNCS*, pages 299–319. Springer-Verlag.
- [Oliva et al., 1998] Oliva, A., Garcia, I. C., and Buzato, L. E. (1998). The reflective architecture of Guaranà. Technical report, Instituto de Computação, Universidade Estadual de Campinas.
- [Peschanski, 2000] Peschanski, F. (2000). Architecture réflexive à base de composants pour la construction d'applications concurrentes et réparties. In *LMO 2000*. Hermès.
- [Raverdy and Lea, 1999] Raverdy, P.-G. and Lea, R. (1999). Reflection support for adaptive distributed applications. In *Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC '99)*.
- [Redmond and Cahill, 2000] Redmond, B. and Cahill, V. (2000). Iguana/J : Towards a dynamic and efficient reflective architecture for Java. In *Workshop on Reflection and Meta-Level Architectures at 14th European Conference on Object-Oriented Programming (ECOOP 2000)*.
- [Riveill and Senart, 2001] Riveill, M. and Senart, A. (2001). Aspects dynamiques des langages de description d'architecture logicielles. *L'objet*, X(X/2001).

- [Robben et al., 1999] Robben, B., Vanhaute, B., Joosen, W., and Verbaeten, P. (1999). Non-functional policies. In *Proceedings of the Second International Conference on Metalevel Architectures and Reflection*.
- [Tanter et al., 2001] Tanter, E., Bouraqadi-Saâdani, N. M. N., and Noyé, J. (2001). Reflex - towards an open reflective extension of Java. In *Proceedings of Reflection 2001*, LNCS. Springer Verlag.
- [Truyen et al., 2001] Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., and Jørgensen, B. N. (2001). Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001)*.
- [un Li and Nahrstedt, 2000] un Li, B. and Nahrstedt, K. (2000). QualProbes : middleware QoS profiling services for configuring adaptive applications. In *Proceedings of Middleware 2000*.
- [Welch and Stroud, 2000] Welch, I. and Stroud, R. J. (2000). Kava - a reflective Java based on bytecode rewriting. In *Reflection and Software Engineering*, volume 1826 of *LNCS*. Springer-Verlag.

Liste des Figures

1.1	Introduction de la couche <i>middleware</i>	4
2.1	Exemple de définition de protocole Iguana/J	10
2.2	Exemple d'association de protocole dans Iguana/J	11
2.3	Exemple de script de configuration pour Kava	13
2.4	Composition de métaobjets dans RAM	14
2.5	Structure d'un composant Open-ORB	15
2.6	Utilisation du système Olan	17
2.7	Adaptation de comportement dans R-RIO	19
2.8	Exemple de procédure adaptable LEAD++	20
2.9	L'architecture de DART	22
2.10	Exemple de composant adaptatif	23
2.11	Structure d'un composant adaptatif dans Molène	25
2.12	Composants JavaPod	25
2.13	Profils utilisateur et d'application	28
2.14	Fragment d'un document XML représentant les ressources du système	29
2.15	Exemple de configuration XML	29
2.16	Configuration de l'invocation d'un service	30
3.1	Problème, contexte et solution	31
3.2	Les différentes couches logicielles	33
4.1	Décomposition fonctionnelle du système	39
4.2	Structure des services	42
4.3	Exemple d'arborescence de ressources	46
4.4	Exemple d'attributs de ressources	47
4.5	Diagramme UML des ressources et attributs	48
4.6	Exemples de désignation de ressources et attributs	49
4.7	Exemple de fichier de configuration des ressources	50
4.8	Exemple de fichier de configuration automatisé	51
4.9	Exemple de configuration mixte	51
4.10	Forme générale d'une politique système	55
4.11	Fonctionnement d'une règle d'adaptation	58
4.12	Syntaxe pour la définition de groupes de composants	60
4.13	Organisation des groupes de composants	61
4.14	Syntaxe pour l'association entre groupes de composants et politiques système	62

5.1	La classe <code>ResourcesManager</code>	65
5.2	La classe <code>MonitoredResource</code>	66
5.3	Les classes du framework d'observation des ressources	66
5.4	Syntaxe des éléments terminaux des expressions	67
5.5	Propagation des notifications lors de la modification d'un attribut	69
5.6	Les classes du package <code>proto.groups</code>	70
5.7	Classes interprétant les politiques	73
5.8	Politique applicative utilisée pour le serveur	77
5.9	Politique système utilisée pour le serveur	77
5.10	Politique applicative utilisée pour le client	78
5.11	Politique système utilisée pour le client	78

Sommaire

1	Introduction	3
1.1	Motivation	3
1.2	Contexte	4
1.3	Solution proposée	5
1.4	Organisation du rapport	5
2	État de l'art	6
2.1	Réflexion et protocoles à métaobjets	6
2.1.1	Introduction à la réflexion	6
2.1.2	Guaranà	6
2.1.3	ProActive	8
2.1.4	Iguana/J	9
2.1.5	Comet	11
2.1.6	Kava	12
2.1.7	RAM	13
2.2	Middlewares réflexifs et adaptables	15
2.2.1	Open-ORB	15
2.2.2	Olan	16
2.2.3	R-RIO	18
2.2.4	LEAD++	20
2.2.5	DART	21
2.2.6	Adaptive components	22
2.2.7	Molène	23
2.2.8	JavaPod	24
2.2.9	Lasagne	26
2.2.10	dynamicTAO	27
2.2.11	Context-aware applications	28
3	Analyse de l'adaptabilité	31
3.1	Définition de l'adaptabilité dynamique	31
3.1.1	Sujet de l'adaptation	34
3.1.2	Acteur de l'adaptation	35
3.1.3	Moments de l'adaptation	35
3.2	Fonctionnalités requises pour l'adaptabilité dynamique	36
3.2.1	Observation	36

3.2.2	Décision	37
3.2.3	Action	37
4	Architecture proposée	38
4.1	Vision globale	38
4.2	Action	40
4.2.1	Composants fonctionnels et conteneurs	40
4.2.2	Services non-fonctionnels	41
4.2.3	Association dynamique de services	43
4.3	Observation	44
4.3.1	Observation des ressources physiques	44
4.3.2	Observation des composants logiciels	52
4.4	Décision	53
4.4.1	Différents niveaux d'abstraction	54
4.4.2	Politiques système	54
4.4.3	Politiques applicatives	58
4.4.4	Liaison entre politiques système et applicatives	61
4.5	Utilisation du système	62
4.5.1	Éléments constituant le système	62
4.5.2	Travail du programmeur d'application	63
4.5.3	Conclusion	63
5	Description du prototype	64
5.1	Architecture générale	64
5.2	Framework d'observation	65
5.3	Évaluation d'expressions	65
5.3.1	Expressions terminales	67
5.3.2	Combinateurs d'expressions	67
5.3.3	Expressions liées et libres	68
5.3.4	Évaluation incrémentale d'expressions	68
5.4	Groupes de composants	69
5.5	Services, rôles et métaobjets	70
5.5.1	Le MOP	71
5.5.2	Implémentation des services et des rôles	71
5.6	Gestion des politiques	72
5.7	Le lanceur	73
5.8	Exemple d'utilisation	74
5.8.1	Le code métier	74
5.8.2	Les fichiers de configuration communs	75
5.8.3	Les politiques système et applicatives	76
5.8.4	Exécution des programmes	76
5.8.5	Conclusion	79

6 Conclusion	80
6.1 Apport de ce travail	80
6.2 Questions ouvertes	80
6.3 Travaux futurs	81
Bibliographie	i
Liste des Figures	vi
Sommaire	vii

Une infrastructure pour middleware adaptable

Pierre-Charles DAVID
(encadré par Thomas LEDOUX, École des Mines de Nantes)

Abstract

Cette étude a permis de concevoir et développer une infrastructure pour le middleware adaptable, capable d'adapter de façon la plus transparente possible (pour le programmeur d'application) les applications à des conditions changeantes d'exécution, aussi bien statiquement (configuration au déploiement) que dynamiquement (reconfiguration automatique à l'exécution).

La solution proposée est basée sur un protocole à métaobjets de type run-time permettant de séparer le code fonctionnel des services non-fonctionnels des applications et de modifier dynamiquement leurs associations. L'infrastructure développée met en oeuvre des politiques d'adaptation écrites en XML (certaines définies par le programmeur, d'autres de type système indépendantes des applications) pour décider quand et comment effectuer ces reconfigurations à la volée.

Termes généraux : adaptation, middleware, réflexion, XML