



Université Joseph Fourier

U.F.R. Informatique et
Mathématiques Appliquées



Institut National Polytechnique
de Grenoble

ENSIMAG

I.M.A.G.

École Doctorale
Mathématique et Informatique

DEA Informatique :
Systèmes et Communications

Projet présenté par

Romain Lenglet

Conteneurs adaptables pour le déploiement de composants

Effectué dans le département DTL/ASR
de France Télécom R&D

Date : 2001-06-18

Jury : J. Briat

J. Caelen

L. Bellissard

P. Déchamboux

T. Coupaye

Table des matières

I	Construction de plate-formes pour des composants répartis	8
1	Plate-formes pour des composants répartis	10
1.1	Principes de conception par objets	10
1.2	Le modèle de référence OSI RM-ODP	17
1.2.1	Points de vue	18
1.2.2	Fonctions et transparences	24
1.3	Généralités sur les modèles de composants répartis	25
1.3.1	Architecture	26
1.3.2	Séparation métier–technique	27
1.3.3	Modèles de composants industriels	29
1.4	Enterprise Java Beans	30
1.4.1	Modèle de composants	31
1.4.2	Architecture d’un conteneur EJB	33
2	Techniques de séparation de problèmes	37
2.1	Réflexivité	37
2.2	Programmation par aspects (AOP)	39
2.3	Filtres de composition	41
2.4	Programmation par sujets (SOP)	42
3	Synthèse	45
II	Construction d’un conteneur EJB adaptable et extensible	47
4	Extension du modèle EJB	48
4.1	Services techniques	48

4.2	Interactions et dépendances entre beans	49
4.3	Interfaces locales et transparence de répartition	50
4.4	Composition des beans	51
4.5	Conclusion	52
5	Proposition d'architecture d'un conteneur EJB	53
5.1	Démarche globale	54
5.2	Objet d'interposition générique	56
5.3	Utilisation de l'AOP pour le talon serveur générique	62
5.4	Gestion de l'état des sessions et des entités	64
5.5	Canaux de communication ouverts	66
5.6	Interactions avec les talons génériques	67
5.7	Propagation de contextes d'exécution	69
5.8	Déploiement des beans	74
5.9	Synthèse	76
6	Conclusion	80
A	Concepts de OSI RM-ODP	85

Table des figures

1.1	Délégation abstraite pour l'ouverture-fermeture	11
1.2	Patron de conception STRATÉGIE	11
1.3	Délégation abstraite pour l'inversion des dépendances	13
1.4	Héritage multiple pour la séparation des interfaces	14
1.5	Patron de conception ADAPTATEUR	14
1.6	Canal entre objets d'ingénierie de base	23
1.7	Architecture globale d'un conteneur EJB	34
1.8	Isolation des instances des beans	35
5.1	Architecture d'objet d'interposition serveur générique pour un session bean ou un entity bean	58
5.2	Diagramme de classes pour un EJBObject générique	59
5.3	Architecture d'objet d'interposition client générique pour un session bean ou un entity bean	60
5.4	Architecture pour la composition des services techniques dans un talon serveur générique	64
5.5	Architecture pour la gestion de l'état des sessions et des entités	66
5.6	Inversion des dépendances entre canaux et objets d'interpositions	67
5.7	Interactions dans les objets d'interposition pour l'exécution d'une interrogation	69
5.8	Communication par propagation implicite de contexte d'exécution	70
5.9	Proposition d'architecture pour la désérialisation de contextes d'exécution	72
5.10	Architecture globale de notre conteneur	77

Remerciements

Je tiens à remercier ici Pascal Déchamboux et Thierry Coupaye de m'avoir proposé de réaliser ce projet qui me passionne, de m'avoir guidé pendant sa réalisation, et de m'avoir offert de le poursuivre en thèse pendant les années à venir. En particulier, je suis très reconnaissant à Thierry pour ses nombreuses (re)lectures et remarques constructives.

Je remercie également Luc Bellissard d'avoir accepté de participer à mon jury de DEA.

Je souhaite remercier Mikaël pour son soutien, ses conseils, et ses lectures de ce rapport.

Je remercie aussi Sonia, pour sa relecture de ce rapport dans l'urgence, et pour son enthousiasme et ses encouragements.

Enfin, je suis reconnaissant à Aree simplement d'avoir été présente à mes côtés.

Introduction

Ce rapport présente le travail que j'ai réalisé au cours de mon projet de DEA Informatique, Systèmes et Communication, au sein du département DTL/ASR (Architecture des Systèmes Répartis) de France Télécom R&D. Les travaux de ce département portent globalement sur la spécification et la construction des couches logicielles pour la mise en oeuvre d'architectures réparties.

Ce stage de DEA se place dans le domaine de la construction d'applications réparties par composants. Il est plus particulièrement centré autour des Enterprise Java Beans (EJB), qui sont un modèle de composants répartis basé sur le langage Java. Comme d'autres modèles de composants répartis, le modèle EJB sépare la programmation des composants, les "beans" dans les EJB, de la mise en oeuvre de services techniques tels que la répartition, la gestion de transactions ou la sécurité, de manière transparente pour les programmeurs.

Un point important du modèle EJB est qu'il fait apparaître la notion de "conteneur", c'est-à-dire de support d'exécution qui met en oeuvre des services techniques pour les composants. Notamment, les EJB spécifient un ensemble de contrats qui permettent à un bean d'être exécuté dans des conteneurs différents, et à un conteneur d'exécuter des beans différents. Mais l'un des aspects négatifs des EJB est que cette notion de conteneur est incomplète : un conteneur est perçu comme une "boîte noire". Un autre problème est que le modèle EJB est figé et n'est pas extensible. Notamment, l'ensemble des services techniques mis en oeuvre dans un conteneur est restreint, et le modèle EJB ne spécifie pas de mécanisme d'extension pour les conteneurs.

L'objectif de mon travail de DEA est d'abord de préciser les problèmes liés aux modèles de composants répartis actuels, et particulièrement des EJB, pour ensuite proposer une architecture de conteneur EJB ouverte et extensible qui résoud ces problèmes, en utilisant des techniques appropriées aux problèmes rencontrés.

La première partie de ce document présente la problématique de la construction des systèmes à objets en général, et des systèmes à objets répartis en particulier. Nous décrivons les propriétés souhaitées de tels systèmes, avec notamment un ensemble de principes de conception, et un canevas standard de spécification des systèmes répartis ouverts, RM-ODP. Ces outils nous permettent d'évaluer l'intérêt des modèles de composants répartis, notamment les EJB, pour la construction de tels systèmes. Nous montrons également les limitations de ces modèles, et les problèmes que pose la conception des conteneurs. Nous présentons ensuite un ensemble de techniques de séparation et de composition de "problèmes", qui permettent de résoudre certains de ces problèmes de conception.

Nous proposons dans la seconde partie une architecture de conteneur EJB ouverte et extensible. Nous nous attachons particulièrement à maîtriser les dépendances entre les parties de ce conteneur, afin de le rendre "ouvert-fermé" en ce qui concerne la mise en oeuvre des services techniques et des canaux de communication. Nous comparons notre approche avec celle d'un conteneur EJB existant, JOnAS, et nous montrons que notre conteneur est plus ouvert et plus flexible que JOnAS.

Première partie

Construction de plate-formes pour des composants répartis

Introduction

Lors du développement d'un système, le principal problème rencontré est la gestion des dépendances entre les parties de ce système. En effet, ces dépendances ont une influence sur la capacité du système à évoluer et à être étendu. Les caractéristiques des systèmes répartis apportent des difficultés supplémentaires pour leur construction, qui rendent l'architecture de ces systèmes plus complexe, avec des dépendances plus difficilement maîtrisables.

La programmation par objets a pour objectif de résoudre certains de ces problèmes. Ses principes de base sont, entre autres, l'encapsulation, l'héritage et le polymorphisme [MK90]. Ces principes servent à structurer le logiciel au plus bas niveau (au niveau du code), mais ne sont pas suffisants pour guider la conception d'une application.

Une solution serait d'utiliser systématiquement des patrons de conception réutilisables (*design patterns*) [GHJV94], qui décrivent des abstractions d'architectures (ensembles d'objets et de leurs interactions). L'avantage est que de nombreux patrons sont bien connus (e.g. OBSERVATEUR, ADAPTATEUR...), et sont utiles pour communiquer facilement les concepts sous-jacents à une architecture. Mais les patrons de conceptions ne forment pas un ensemble suffisamment complet et cohérent pour guider la conception complète d'applications.

Il faut donc compléter ces principes de base par des principes généraux de conception, décrits dans la section 1.1, et des techniques de séparation de problèmes, décrites dans le chapitre 2, permettant d'appliquer systématiquement de tels principes.

La construction de logiciel réparti pose de nouveaux problèmes, comme la répartition, la concurrence et l'hétérogénéité.

La section 1.2 présente la problématique des plate-formes logicielles réparties et un modèle de référence pour leur construction (OSI RM-ODP). Les sections 1.3 et 1.4 présentent des spécifications de plate-formes réparties existantes et implantables, notamment les Enterprise Java Beans. Le chapitre 2 montre l'intérêt des techniques de séparation de problèmes pour la construction de telles plate-formes logicielles.

Chapitre 1

Plate-formes pour des composants répartis

1.1 Principes de conception par objets

Nous pouvons trouver dans la littérature plusieurs principes qui guident la conception d'applications par objets [GM01]. Ces principes s'appuient sur les principes de base de la programmation par objets, et permettent de concevoir du logiciel de bonne qualité.

Dans cette section, nous considérons qu'un *module* logiciel correspond généralement à une classe d'objets (d'après [Mey97]), bien que les principes décrits puissent s'appliquer à des modules de granularité différente (e.g. paquetage, méthode, etc.). De même, dans cette section, une interface est définie comme une classe abstraite qui définit seulement des opérations, sans donner leur implantation.

Par définition (selon le Petit Robert), un principe est une proposition non démontrée, mais vérifiée dans ses conséquences. Cette section décrit ces principes de conception en montrant informellement leur intérêt du point de vue du concepteur d'un logiciel.

Principe d'ouverture-fermeture

Tout module doit être à la fois ouvert et fermé.

(Open-Closed Principle) [Mey97]

Un module est dit *ouvert (aux extensions)* s'il est possible de l'étendre. Un module est dit *fermé (aux modifications)* si sa description est stable et bien définie, et s'il est disponible pour être utilisé par d'autres modules. L'ouverture est nécessaire, car un module est souvent amené à évoluer pour être adapté à de nouveaux besoins non prévus initialement. La fermeture est aussi nécessaire, car un module ne peut pas être utilisé par d'autres modules si sa définition n'est pas stable. En effet, une modification du code d'un module peut induire une

modification de sa description. Cela peut donc entraîner une modification des modules qui en dépendent, de manière récursive car les dépendances peuvent être considérées comme transitives. Donc une instabilité de la description d'un module entraîne une maintenance coûteuse. Ceci peut être évité en conservant les modules fermés. Nous reformulons le principe d'ouverture-fermeture dans ce sens :

Tout module doit pouvoir être étendu, et ces extensions doivent être introduites sans modifier le code existant, donc seulement en ajoutant du code.

Un mécanisme permettant d'appliquer ce principe à un module est la *délégation abstraite* (fig. 1.1). Une interface (fixe) est introduite, qui décrit de manière abstraite le comportement du module. Les implantations (variables) du module implantent cette interface. Les modules extérieurs ne dépendent que de l'interface fixe, et pas des implantations variables, donc le module est ouvert-fermé.

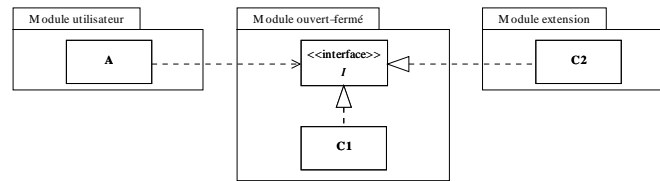


FIG. 1.1 – Délégation abstraite pour l'ouverture-fermeture

De nombreux patrons de conception appliquent ce principe. Par exemple, le patron STRATÉGIE (fig. 1.2) permet de définir une famille d'algorithmes, de les encapsuler séparément, et de les rendre interchangeables. L'abstraction de la famille d'algorithmes est matérialisée par une interface qui définit toutes les opérations de cette famille, et qui est implantée par chaque algorithme spécifique. Le code utilisant la famille d'algorithme dépend seulement de l'interface, dont la description est fixe. Donc l'architecture construite est ouverte-fermée à l'ajout de nouveaux algorithmes.

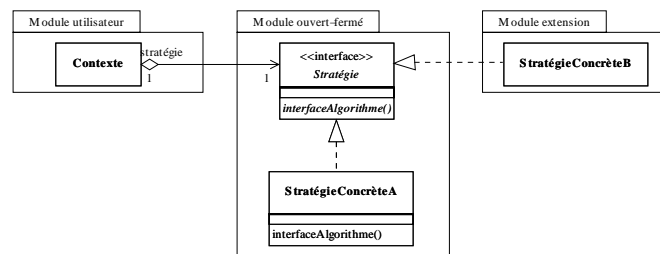


FIG. 1.2 – Patron de conception STRATÉGIE

Il faut cependant remarquer qu'une application généralisée de ce principe résulte en une plus grande complexité du logiciel. Il faut donc n'appliquer ce principe que lorsque cela simplifie la conception et permet de rendre flexibles des points stratégiques du logiciel.

Principe d'auto-documentation

Le concepteur d'un module doit s'efforcer d'inclure toute information à propos du module dans le module lui-même.

(Self-Documentation Principle) [Mey97]

L'application de ce principe favorise la *compréhension modulaire* de ce module, c'est-à-dire qu'un humain peut comprendre facilement ce module sans connaître les autres modules, ou au pire en en connaissant seulement quelques uns. En effet, si le code du module et sa documentation sont maintenus séparément, il y a un risque de désynchronisation entre ces deux produits, et donc de difficulté de compréhension du module. Il n'est question ici que de la documentation interne du module, pas de son manuel d'utilisation qui peut être livré séparément du module.

Dans le contexte de la programmation dans le langage Java, par exemple, une classe doit contenir des commentaires dans les fichiers sources pour chaque attribut et opération de la classe, par exemple au format Javadoc [GJS96].

Principe de substitution de Liskov

Les programmes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir.

(Liskov Substitution Principle) [LW94][Mar96d]

Ce principe impose que les classes héritant d'une classe de base doivent avoir le même comportement, observé par les programmes utilisateurs, que la classe de base pour les opérations qu'elle définit. Ce principe étend donc les mécanismes de sous-typage classiques de Java ou d'autres langages à objets, qui n'imposent pas de compatibilité de comportement. Un tel mécanisme est par exemple mis en oeuvre dans la conception par contrat [Mey97], en utilisant des préconditions et postconditions pour les opérations, et des invariants pour les types.

Ce principe ne peut pas être systématiquement appliqué en utilisant des patrons de conception. Par exemple, si le patron FABRIQUE ABSTRAITE est utilisé, le programme client dépend seulement de l'interface de produit abstrait, créé par la fabrique abstraite. Il ne dépend pas des classes de produit concret implantant cette interface, et générés par les fabriques concrètes. Donc le programme client peut utiliser de nouvelles implantations de produit concret sans le savoir. Mais dans ce cas, le principe de substitution de Liskov n'est réellement appliqué que si chaque implantation de produit concret a le comportement défini dans l'interface de produit abstrait. Ce type de contrainte n'est pas exprimé dans les définitions de patrons de conception.

Principe d'inversion des dépendances

Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous doivent dépendre d'abstractions. Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions.

(Dependency Inversion Principle) [Mar96a]

Classiquement, les logiciels sont construits de la manière suivante : les modules de haut niveau dépendent directement des modules de bas niveau. Cela pose deux problèmes :

- les modules de haut niveau doivent être modifiés lorsque les modules de bas niveau sont modifiés,
- les modules de haut niveau ne sont pas réutilisables sans réutiliser aussi les modules de bas niveau.

Pour résoudre ces problèmes, ce principe stipule que les modules de bas niveau doivent se conformer à des interfaces définies et utilisées par les modules de haut niveau : les dépendances sont donc inversées.

Pour appliquer ce principe, le mécanisme de délégation abstraite introduit pour le principe d'ouverture-fermeture (fig. 1.3) peut être utilisé : une interface fixe est spécifiée, dont dépend le module de haut niveau, et qui est implantée par le module de bas niveau. Ainsi, les deux modules dépendent d'une interface fixe, donc le module de haut niveau n'est plus impacté par les changements du module de bas niveau. Il est plus facilement réutilisable avec d'autres implantations de modules de bas niveau.

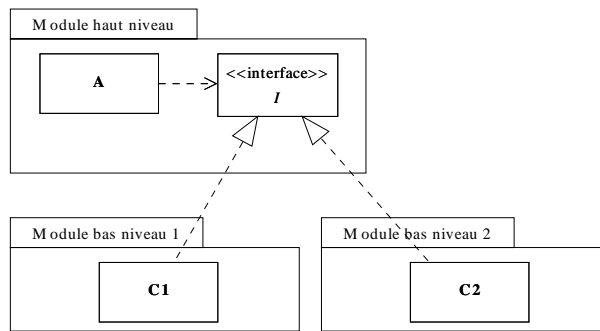


FIG. 1.3 – Délégation abstraite pour l'inversion des dépendances

Ce principe est différent du principe d'ouverture-fermeture, car l'interface fixe est spécifiée par le module utilisateur, et non par le module utilisé.

Principe de séparation des interfaces

Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas.

(Interface Segregation Principle) [Mar96c]

Ce principe stipule que chaque fonctionnalité (ou service) distincte doit être spécifiée dans une interface séparée. Si un module implante simultanément plusieurs fonctionnalités non reliées entre elles, chaque fonctionnalité doit être accessible seulement à travers une interface qui lui est spécifique. En effet, si un module n'offre qu'une seule interface pour toutes les fonctionnalités qu'il offre aux autres

modules, ceux-ci voient une interface trop riche et complexe, et peuvent être impactés par des changements de l'interface, même s'ils ne se servent que d'une partie de cette interface.

Il y a deux moyens d'appliquer ce principe :

- l'héritage multiple (fig. 1.4) : chaque fonctionnalité est spécifiée dans une interface, qui est implémentée par les modules serveurs qui offrent cette fonctionnalité. Les modules clients ne dépendent que des interfaces des fonctionnalités qu'ils utilisent.
- le patron de conception ADAPTATEUR (fig. 1.5) : lorsque l'héritage multiple n'est pas possible, le patron de conception ADAPTATEUR peut être utilisé. Dans ce cas, le module serveur offre toutes les fonctionnalités dans une interface unique, et à chaque fonctionnalité correspond une classe d'adaptation du module (un adaptateur), qui offre une seule fonctionnalité et qui délègue ses traitements au module serveur. Les modules clients dépendent seulement des adaptateurs, et ne voient pas ainsi toutes les fonctionnalités simultanément.

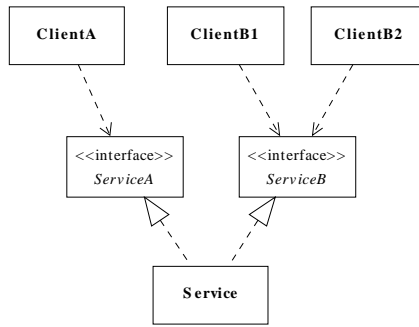


FIG. 1.4 – Héritage multiple pour la séparation des interfaces

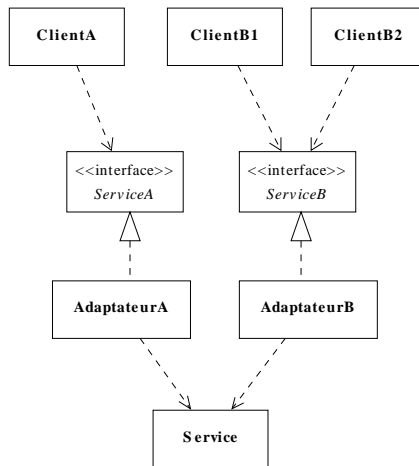


FIG. 1.5 – Patron de conception ADAPTATEUR

Principe de fermeture commune

Les parties du logiciel impactées par les mêmes changements doivent être placées dans un même module.

(Common Closure Principle) [Mar96b]

Il est difficile de fermer (au sens donné pour le principe d'ouverture-fermeture) complètement une application : la flexibilité introduite n'est pas forcément exploitée, et l'application produite est complexe. Il faut donc faire des choix concernant les points stratégiques à "ouvrir-fermer" dans une application. Cela implique qu'il reste souvent dans une application des parties qui sont impactées par des changements dans d'autres parties.

Pour réduire l'impact de ces changements, le principe de fermeture commune stipule que les parties d'une application qui sont simultanément impactées par les mêmes changements doivent être regroupées. Ainsi, l'impact d'un changement est minimisé.

Principe des dépendances acycliques

Les dépendances entre modules doivent former un graphe acyclique.

(Acyclic Dependencies Principle) [Mar96b]

D'après le principe de fermeture commune, sont regroupées dans un même module toutes les parties du code qui sont impactées par un même changement. Ainsi, une modification dans un module peut impacter tout le code dans ce module, mais impactera aussi les modules qui en dépendent si sa fermeture est violée (i.e. si une de ses interfaces est modifiée). De plus, les dépendances entre modules sont transitives. Donc ces dépendances entre modules conditionnent et temporisent la propagation des changements dans l'application, dans la mesure où une modification se propagera d'autant plus que les relations de dépendance sont nombreuses et fortes.

Ainsi, dans le cas d'une rupture de fermeture d'un module, la présence de cycles de dépendance peut être problématique. En effet, si deux modules dépendent l'un de l'autre, un changement dans l'un des modules entraîne des changements dans l'autre, dans les deux sens, de sorte que ces modules doivent évoluer ensemble et être livrés ensemble lors d'une modification dans l'un d'entre eux. Par extension, tous les modules formant un cycle de dépendance doivent évoluer ensemble et être livrés ensemble. Ceci nous fait perdre l'avantage de la temporisation des modifications, apporté par le découpage en modules, et qui permet de ne livrer qu'un nombre minimal de modules après une modification.

Un moyen de supprimer un cycle dans un graphe de dépendances entre des modules est d'appliquer le principe d'inversion des dépendances sur une des dépendances du cycle. En appliquant systématiquement ce principe à tous les cycles du graphe, itérativement, nous obtenons un graphe orienté acyclique qui vérifie le principe des dépendances acycliques. Tous les modules formant un cycle peuvent aussi être regroupés dans un même module, en appliquant le principe de fermeture commune.

Principe de relation dépendance-stabilité

Un module doit dépendre uniquement de modules plus stables que lui.

(Stable Dependencies Principle) [Mar97]

La *stabilité* d'un module peut être mesurée comme l'inverse de la fréquence de ses changements au cours du temps. Les changements les plus importants (parce que les plus coûteux) sont ceux qui concernent plusieurs modules à la fois. La stabilité d'un module est donc liée :

- au nombre de modules dont il dépend : plus ce nombre est grand, plus le module peut être impacté par des modifications de ces modules, et donc moins il est stable,
- au nombre de modules qui dépendent de ce module : plus ce nombre est grand, plus les modifications de ce module sont coûteuses, car plus de nombreux modules sont impactés, donc moins de modifications sont apportées à ce module, donc plus il est stable.

Le principe de relation dépendance-stabilité permet donc de vérifier que les modules les moins stables impacteront le moins de modules possibles, et seulement des modules encore moins stables. Cela minimise l'impact des changements les plus fréquents, et maximise la stabilité globale.

Principe de stabilité des abstractions

Les modules les plus stables doivent être les plus abstraits. Les modules instables doivent être concrets. Le degré d'abstraction d'un module doit correspondre à son degré de stabilité.

(Stable Abstractions Principle) [Mar97]

Ce principe découle des principes d'ouverture-fermeture et d'inversion des dépendances.

Le principe d'ouverture-fermeture montre que les interfaces servent à stabiliser (fermer) des parties stratégiques d'une application. Ainsi, ces parties et leurs interfaces forment les modules les plus stables. Ces modules sont aussi les plus abstraits, car les interfaces ne contiennent pas de détails.

D'après le principe d'inversion des dépendances, si un module est concret (contient des détails), il ne contient pas d'interfaces dont dépendent d'autres modules. Il y a donc peu de dépendances vers ce module : il peut donc être instable.

Conclusion

Les principes de conception présentés dans cette section ont pour objectifs de garantir la flexibilité et l'évolutivité des applications, en introduisant des abstractions et en gérant les dépendances entre modules. Ils ne donnent pas de modèles d'architecture, comme les patrons de conception réutilisables. Mais ils guident la construction de ces architectures et leur composition, et sont donc complémentaires des patrons de conception. Nous avons aussi montré que ces

patrons de conception permettent d'appliquer localement certains de ces principes.

Cependant, l'application de ces principes est coûteuse, et rend les applications plus complexes. Il ne faut donc les appliquer qu'aux parties stratégiques d'une application. Ceci impose donc une analyse avancée des applications pour déterminer ces parties stratégiques.

1.2 Le modèle de référence OSI RM-ODP

Le standard ODP (Open Distributed Processing) [ISO95c][Ste95] de l'ISO a pour but de gérer les problèmes liés aux systèmes répartis. Selon ce standard, les caractéristiques de tels systèmes sont principalement :

- *répartition* : les composants d'un système réparti sont répartis dans l'espace, et leurs interactions peuvent être locales ou réparties,
- *concurrency* : tout composant d'un système réparti peut s'exécuter en parallèle avec tout autre composant,
- *absence d'état global* déterminé précisément,
- *défaillances partielles* : les composants peuvent être défaillants indépendamment les uns des autres,
- *asynchronisme des communications* : il n'existe pas d'horloge globale au système,
- *hétérogénéité* des matériels, systèmes d'exploitation, protocoles, etc.,
- *évolution* des systèmes pour prendre en compte de nouveaux types d'applications et l'évolution des supports d'exécution.

Pour prendre en compte ces caractéristiques, les systèmes répartis doivent avoir les propriétés suivantes :

- *ouverture*, pour permettre la *portabilité* et l'*interopérabilité* (ou *compatibilité*) des applications,
- *intégration*, pour pouvoir incorporer dans un même système des systèmes et ressources différents sans nécessité de développement supplémentaire ; à rapprocher de la *réutilisabilité* (capacité d'un élément logiciel à servir à construire de nombreuses applications différentes [Mey97]),
- *flexibilité* et *extensibilité*, pour supporter l'évolution des systèmes, y compris leur reconfiguration dynamique pour les adapter aux changements d'environnement ou de spécification,
- *modularité*, c'est-à-dire le découpage des systèmes en parties autonomes mises en relation, pour permettre la flexibilité,
- *fédération*, pour pouvoir combiner des systèmes de domaines techniques et administratifs différents,
- *administrabilité*, pour observer, contrôler et gérer les ressources d'un système,
- fourniture de *qualité de service*,
- *sécurité* d'accès aux applications et aux données,
- *transparence*, pour masquer aux applications les détails et différences des mécanismes utilisés pour résoudre les problèmes liés à la répartition.

L'ouverture est une propriété globale des systèmes ODP : un système est dit *ouvert*, dans ODP, s'il se conforme aux standards ODP. Les propriétés d'intégration, de flexibilité et de modularité sont des propriétés souhaitées dans les

systèmes en général, y compris en environnement non réparti. Elles peuvent donc par exemple être obtenues en appliquant des principes de conception par objets, puisque ODP se base sur le concept d'objet. Nous présentons informellement dans cette section comment ODP permet d'appliquer ces principes.

Les propriétés comme la transparence (voir section 1.2.2) sont spécifiques aux systèmes répartis. Le but du standard ODP est le développement d'un canevas (*framework*) pour la spécification de systèmes, et des composants d'infrastructure correspondants, pour la construction de systèmes ayant toutes ces propriétés. Le Modèle de Référence d'ODP (RM-ODP) fournit ce canevas, qui est utilisé pour structurer les standards de spécification, et garantir leur cohérence. ODP définit deux types de standards de spécification (un standard peut être des deux types à la fois) :

- *component standard* : spécifie une seule fonction, e.g. la fonction de transaction,
- *component composition standard* : spécifie un modèle d'assemblage de composants pour réaliser complètement un objectif du système, e.g. la fourniture complète d'une transparence (voir section 1.2.1.4).

RM-ODP étant un canevas de spécification de systèmes répartis, il peut être considéré comme un modèle pour les spécifications comme CORBA et EJB [Cha99] (présentés dans la section 1.4), qui à leur tour sont directement implantables dans des supports d'exécution répartis. Pour l'implantation d'un tel support, il faut donc faire des choix spécifiques à deux étapes :

- lors de la rédaction de la spécification, en précisant les points de conformité de la spécification, et en choisissant et en précisant les transparences à mettre en oeuvre,
- lors de l'implantation, en choisissant les composants et l'architecture pour mettre en oeuvre les fonctions, et vérifier les points de conformité.

RM-ODP est divisé en quatre parties :

- Part I - Overview : introduit le standard, et son utilité. Cette partie est non normative.
- Part II - Foundation : définit les concepts utilisés dans les autres parties du standard.
- Part III - Architecture : spécifie les caractéristiques nécessaires pour qualifier un traitement réparti d'ouvert. Ce sont les contraintes que doivent respecter les standards ODP.
- Part IV - Architectural semantics : contient une formalisation des concepts de modélisation définis dans la partie II.

La partie II [ISO95a] est présentée en Annexe A. Les sections suivantes décrivent la partie III [ISO95b].

1.2.1 Points de vue

Pour simplifier la compréhension des systèmes répartis, RM-ODP introduit la notion de point de vue (*viewpoint*). Chaque point de vue est un modèle spécifiant un aspect des systèmes. Tous les points de vue décrivent tous les systèmes

complètement, mais avec des abstractions différentes. RM-ODP introduit cinq points de vue :

- métier (*enterprise viewpoint*)
- information (*information viewpoint*)
- traitement (*computational viewpoint*)
- ingénierie (*engineering viewpoint*)
- technologie (*technology viewpoint*)

Chaque point de vue est décrit dans RM-ODP avec un vocabulaire propre (*viewpoint language*) qui s'appuie sur les concepts définis dans la partie II de ce standard, et les raffine.

Une *spécification* est la description d'un système dans un point de vue donné. Une spécification doit donc être décrite dans le langage correspondant, et donner des points de conformité correspondant aux points de références de ce point de vue. De plus, RM-ODP donne peu de contraintes de cohérence entre les points de vue, mais les spécifications d'un même système dans des points de vue différents doivent être cohérentes et ne pas donner d'assertions contradictoires. Une spécification complète d'un système doit donc donner des correspondances entre les différents points de vue.

Cette section présente ces cinq points de vue. Mais dans notre problématique de construction de plate-formes réparties, nous nous intéressons particulièrement aux points de vue traitement et ingénierie.

1.2.1.1 Point de vue métier

Une spécification métier doit définir, dans le langage métier : l'objectif, le champ d'application et les politiques d'un système. Cette définition est à rapprocher des cas d'utilisation UML [RJB99]. Les systèmes sont décrits en termes :

- d'objets métier,
- de rôles remplis par ces objets métiers pour atteindre les objectifs,
- de politiques concernant les relations entre ces objets, leur contrats, etc.

Les points de conformité métier sont un ensemble donné d'objectifs et de politiques qui doivent être respectés par les systèmes implantés.

1.2.1.2 Point de vue information

Une spécification d'information définit la sémantique des informations et de leur traitement dans un système. Une telle spécification définit donc l'état de l'ensemble des objets d'un système, et l'évolution de ces états, en utilisant des schémas qui peuvent être de trois sortes :

- *schéma invariant* : un ensemble de prédicats sur les informations, qui doivent être toujours vérifiés,
- *schéma statique* : spécification de l'état d'un ou plusieurs objets à un moment donné, soumis aux contraintes des schémas invariants,

- *schéma dynamique* : spécifications des changements d'état possibles d'un ou plusieurs objets, soumis aux contraintes des schémas invariants. Les schémas dynamiques décrivent aussi la création et la destruction d'objets d'information.

Ces schémas décrivent une partie des systèmes ODP sous la forme d'objets d'information. Ces objets d'information, comme dans tous les points de vue, sont encapsulés : leur état ne peut changer qu'à l'occurrence d'actions impliquant ces objets. Les objets d'information peuvent être atomiques ou composites (composés d'autres objets d'information). Les objets d'information composites étant encapsulés, un objet d'information composant d'un composite ne peut pas être composant d'un autre composite.

Les points de conformité d'information, donnés dans une spécification d'information, sont un ensemble donné de schémas auxquels doivent se conformer les systèmes implantés. Les interactions aux points de conformité de traitement et d'ingénierie doivent être cohérentes avec ces schémas d'information.

1.2.1.3 Point de vue traitement

Une spécification de traitement définit la décomposition fonctionnelle d'un système en objets de traitements (*computational object*) qui interagissent à leurs interfaces.

Dans ce point de vue, à la fois les applications et les fonctions ODP (services techniques) consistent en des configurations d'objets qui interagissent. Une particularité d'ODP est donc de considérer les services techniques au même niveau d'abstraction que les applications, et non comme des couches logicielles séparées où les couches de niveau supérieur dépendent des couches de niveau inférieur (e.g. l'architecture standard OSI de l'ISO, pour les protocoles de communication). En effet, même si la séparation en couches permet de séparer les problèmes et de ne considérer qu'une partie des problèmes dans chaque couche, cela empêche d'avoir une vue homogène de tous les aspects du système.

Dans le point de vue traitement, il existe trois modèles d'interaction entre objets de traitements :

- *signal* : interaction atomique, communication unidirectionnelle entre deux objets,
- *opération* : il y a deux types d'opérations :
 - *annonce* (*announcement*) : une interaction (invocation) avec un objet serveur, initiée par un objet client qui requiert l'exécution d'une fonction par cet objet serveur,
 - *interrogation* : consiste en une invocation initiée par un objet client vers un objet serveur, suivie d'une interaction (terminaison) initiée par l'objet serveur vers l'objet client en réponse à l'invocation,
- *flot* (*flow*) : abstraction d'une séquence d'interactions, résultant en la transmission d'information entre un objet producteur vers un objet consommateur.

Les opérations et les flots peuvent être décrits en termes de signaux, à un niveau d'abstraction inférieur. En effet, chaque invocation ou terminaison peut être décrite comme un signal, donc chaque opération peut être décrite comme

une séquence de signaux. De même un flot est une séquence d'interactions qui peuvent être décrites par des signaux, donc un flot peut être décrit par des signaux.

Chaque interaction fait partie d'une interface unique dans chaque objet de traitement participant à l'interaction. Les interfaces d'un objet constituent donc une partition des interactions de cet objet. De plus, les interactions n'ont lieu qu'entre les interfaces des objets. Ainsi, les objets ne dépendent que d'interfaces, ce qui facilite l'application systématique du principe d'ouverture-fermeture au niveau des objets. Et toutes les interactions initiées par un objet sont identifiées dans des interfaces, ce qui permet de déterminer systématiquement les services qu'il requiert d'autres objets.

Les objets peuvent avoir des interfaces de trois types, correspondant à ces trois types d'interactions : signal, opération et flot (*stream*). Chaque objet participant à une interaction doit porter une interface correspondant au type d'interaction, et à la causalité de l'interaction dans cette interface. Cette causalité peut être *initiateur* ou *répondeur* pour une interface de type signal ou opération, ou *producteur* ou *consommateur* pour une interface de type flot.

Les interactions entre les objets de traitement se font dans des *contextes contractuels* appelés liens (*bindings*). Ces liens sont mis en oeuvre explicitement ou implicitement. Les liens explicites sont établis par des actions d'objets. Un lien implicite est mise en oeuvre dans le cas où un objet client initie une interaction avec une interface d'opération d'un objet serveur, à laquelle le client n'est pas lié explicitement. Dans ce cas, si le client ne possède pas une interface complémentaire à celle du serveur à laquelle il est lié, le lien implicite crée pour le client une interface complémentaire à celle du serveur, lie cette interface à celle du serveur, invoque l'interface serveur avec l'interface client créée, et éventuellement détruit l'interface client après l'interaction. La notion de lien implicite est utile pour décrire les langages à objets, comme Java, qui n'ont pas de notion de lien explicite, car ils n'ont pas de notion d'interface initiateur.

Les objets et interfaces de traitement sont décrits et instanciés à l'aide de patrons. Un patron d'objet de traitement est un patron d'objet qui comprend l'ensemble des patrons d'interfaces de traitement que les objets peuvent instancier, avec une spécification de leur comportement et de leur contrat d'environnement. Un patron d'interface de traitement contient la signature des interactions contenues dans les interfaces qu'elle permet d'instancier, ainsi qu'une spécification du comportement et un contrat d'environnement de ces interfaces. Ce contrat d'environnement est appliqué dans les contextes contractuels auxquels participent ces interfaces. Le comportement d'un objet est donc contraint par l'intersection des comportements prescrits par chaque contexte contractuel et dans son patron d'objet.

Le point de vue traitement ne spécifie pas comment concevoir les objets, et en particulier comment les interactions d'un objet sont partitionnées dans ses interfaces. Il faut donc par exemple appliquer le principe de séparation des interfaces [Mar96c], afin que chaque patron d'interface contienne seulement les signatures des interactions d'une fonctionnalité particulière.

Les points de conformité de traitement, donnés dans une spécification de traitement, sont un ensemble de patrons d'objets et d'interfaces auxquels les interfaces et objets doivent se conformer.

1.2.1.4 Point de vue ingénierie

Une spécification d'ingénierie définit les mécanismes et les fonctions requises pour supporter des interactions entre des objets répartis, dans un système ODP [Ste95].

Un *canal* (*channel*) est une configuration d'objets répartis qui permet de supporter de manière transparente les interactions réparties entre objets d'ingénierie de base. Cette configuration comporte des objets des types suivants :

- *talon* (*stub*) : interprète et transforme les interactions véhiculées par le canal. Un talon possède au moins deux interfaces de communication : une pour interagir avec un objet d'ingénierie de base, et une autre pour interagir avec un lieu. C'est le seul type d'objets dans un canal qui interagit directement avec les objets d'ingénierie de base. Il peut être spécifique à l'instance d'interface d'ingénierie à laquelle il est lié, au type d'interface d'ingénierie, ou bien être générique.
- *lieur* (*bindér*) : maintient un lien (*binding*) réparti entre des interfaces d'objets d'ingénierie. Un lieu possède au moins deux interfaces de communication : une pour interagir avec un talon, et une autre pour interagir avec un objet de protocole. Le rôle d'un lieu est d'assurer l'intégrité de bout-en-bout du canal. Son rôle est par exemple de créer, configurer et détruire un canal, et de réparer automatiquement les liens répartis qui ont été cassés.
- *objet de protocole* (*protocol object*) : communique avec d'autres objets de protocole dans le même canal, pour assurer les interactions entre des objets d'ingénierie. Un objet de protocole possède au moins deux interfaces de communication : une pour interagir avec un lieu, et une autre pour interagir avec un autre objet de protocole ou un intercepteur.
- *intercepteur* (*interceptor*) : se place à une frontière entre des domaines donnés (voir annexe A), et réalise des vérifications et transformations sur les interactions entre ces domaines. Un intercepteur possède au moins deux interfaces de communication pour interagir avec des objets de protocole. Un intercepteur peut par exemple servir à convertir les protocoles, entre des objets de protocole de type différent.

Donc dans un canal sont associés à chaque objet d'ingénierie de base : un talon, un lieu et un objet de protocole, liés entre eux. Lorsque deux objets d'ingénierie de base répartis sont liés, les objets de protocole qui leur sont associés dans un canal sont liés. Un intercepteur peut être placé entre deux objets de protocole. Un exemple d'architecture d'un canal est donné dans la fig. 1.6.

Les talons, lieux, etc. peuvent être des composites, et à leur tour être composés respectivement de talons, lieux, etc. Tous les objets dans un canal peuvent avoir des interfaces de contrôle permettant de contrôler leur comportement.

Une interface d'objet d'ingénierie de base est localisée par une *référence d'interface d'ingénierie*. Cette référence contient l'information permettant d'établir des liens vers cette interface d'ingénierie, en créant des canaux et en initialisant des lieux appropriés.

Le langage d'ingénierie de RM-ODP spécifie un découpage hiérarchique des systèmes ODP, et spécifie les fonctionnalités aux différents niveaux d'abstractions :

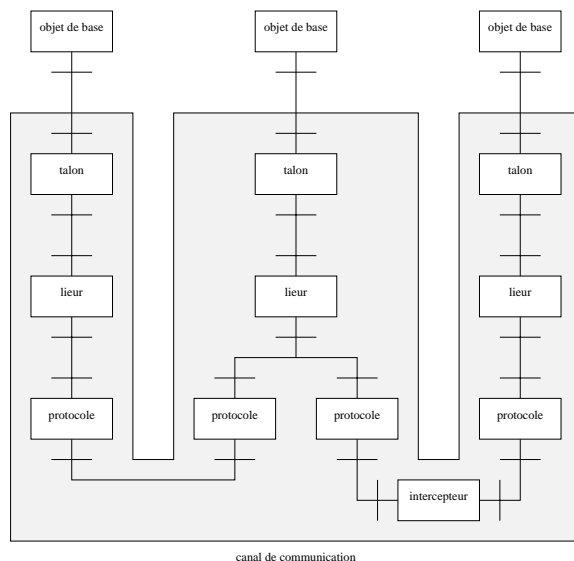


FIG. 1.6 – Canal entre objets d’ingénierie de base

- *grappe (cluster)* : une configuration d’objets d’ingénierie de base formant une unité pour leur activation et désactivation, leur migration et leur récupération après panne. Une grappe est gérée par un *gestionnaire de grappe (cluster manager)*, qui a notamment pour rôle d’instancier cette grappe.
- *capsule* : une configuration d’objets d’ingénierie de base formant une unité pour leur exécution et leur stockage. Un processus ou une machine virtuelle sont des exemples de capsules. Une capsule gère donc l’exécution d’un ensemble de grappes. Une capsule est gérée par un *gestionnaire de capsule (capsule manager)* auquel sont liées toutes les grappes de la capsule. Une grappe ne peut appartenir qu’à une seule capsule. Une capsule est instanciée par le noyau qui supporte son exécution.
- *noyau (nucleus)* : un objet d’ingénierie qui coordonne l’exécution, le stockage et les communications d’objets d’ingénierie sur le même nœud que lui. Un noyau coordonne donc l’exécution de capsules, qui interagissent avec lui pour mettre en oeuvre des services. Un système d’exploitation est un exemple de noyau.
- *nœud (node)* : une configuration d’objets d’ingénierie qui forment une unité pour leur localisation dans l’espace, et qui met en oeuvre un ensemble de fonctions de traitement, de stockage et de communication. Une machine matérielle avec un système d’exploitation forment un exemple de nœud. Nous pouvons remarquer que dans RM-ODP, un nœud (i.e. une machine matérielle) est indissociable d’un noyau (i.e. un système d’exploitation).

Les objets d’ingénierie de base peuvent être liés entre eux, dans une même grappe ou dans des grappes différentes. Des objets dans une même grappe peuvent interagir à travers des liens directs. Mais s’ils sont dans des grappes différentes, ils ne peuvent interagir qu’à travers des liens distants mis en oeuvre par des canaux.

Une spécification d'ingénierie d'un système ODP doit fournir des points de référence de programmation pour les interactions entre les objets d'ingénierie du système, et en particulier entre les objets consistant les canaux, et les gestionnaires des grappes et capsules et les noyaux. De même, une spécification d'ingénierie doit fournir des points de référence pour toutes les interactions entre les objets de protocole dans les canaux. Les points de référence spécifiés sont donc des points stratégiques des systèmes spécifiés. Nous pouvons notamment les considérer comme des points de "fermeture" des systèmes, qui permettent éventuellement d'appliquer le principe d'ouverture-fermeture.

1.2.1.5 Point de vue technologie

Une spécification de technologie définit comment les spécifications (e.g. de traitement, d'ingénierie, etc.) d'un système ODP sont implantées, au plus bas niveau d'abstraction, en spécifiant une configuration d'objets de technologie et leurs interfaces.

Le langage de technologie définit notamment la notion de *standard implantable* (*implementable standard*) comme un patron d'objet de technologie. L'*implantation* est un processus d'instantiation d'un standard implantable, dont la validité peut être testée grâce aux points de conformance définis dans le standard implantable.

Une spécification de technologie spécifie que des objets de technologie donnés sont des instances de standards implantables donnés. Une spécification de technologie est fortement dépendante d'autres spécifications dans d'autres points de vue, car les objets de technologie spécifiés correspondent à des objets ou des termes atomiques dans ces spécifications. Une spécification de technologie est donc produite au terme du processus de spécification complète d'un système ODP, après que les spécifications dans les autres points de vue aient été produites.

1.2.2 Fonctions et transparences

Un des objectifs des systèmes ODP, qui sont décrits avec les cinq points de vue présentés ci-dessus, est de mettre en oeuvre les transparences suivantes :

- *transparence d'accès* (*access transparency*) : masque les différences de représentation des données transmises lors des interactions, pour permettre l'interopérabilité des objets.
- *transparence de défaillances* (*failure transparency*) : masque à un objet ses défaillances ou les défaillances d'objets avec lesquels il interagit.
- *transparence de localisation* (*location transparency*) : masque l'utilisation d'information de localisation dans l'espace, lors de l'identification et de la création de liens entre des interfaces. Les objets peuvent donc interagir entre eux sans dépendre de leur localisation.
- *transparence de migration* (*migration transparency*) : masque à un objet la capacité du système à modifier la localisation de cet objet.
- *transparence de persistance* (*persistence transparency*) : masque à un objet la désactivation et l'activation d'autres objets, ou de lui-même. L'activation est le mécanisme de récupération de l'état d'un objet à partir d'un support persistant, la désactivation est la conservation de cet état sur ce support.

- *transparence de réimplantation (relocation transparency)* : masque à des interfaces le “déménagement” d’une interface à laquelle elles sont liées. Par exemple, si un objet lieur (*binders*) dans un canal détecte que le lien distant est cassé, il peut reconstruire le lien vers une autre interface similaire, de manière transparente aux interfaces qui interagissent par le canal.
- *transparence de duplication (replication transparency)* : masque l’utilisation d’un groupe d’objets ayant des comportements compatibles, pour supporter une seule interface.
- *transparence de transaction (transaction transparency)* : masque la coordination de plusieurs activités (flux d’exécution, ou *threads*) sur une même configuration d’objets, pour garantir leur cohérence.

D’autres transparences peuvent être mises en oeuvre par les systèmes : cette liste n’est pas exhaustive.

Les systèmes qui mettent en oeuvre certaines de ces transparences présentent la propriété de transparence. Cette propriété de transparence, comme les autres propriétés souhaitées des systèmes ODP (e.g. intégration, administrabilité, sécurité...), est implantée dans des fonctions de ces systèmes. Ces fonctions sont des services à caractère technique, par opposition aux services fonctionnels qui sont fournis par l’application et qui sont décrits dans le point de vue métier. Parmi les fonctions définies dans RM-ODP, nous pouvons citer :

- administration des noeuds, grappes et capsules,
- points de reprise, reprise après pannes,
- activation et réactivation des objets,
- duplication d’objets,
- migration d’objets,
- gestion de transactions,
- persistance,
- négociation,
- gestion des activités (*threads*),
- sécurité (contrôle d’accès, authentification...).

Ces fonctions concernent à la fois les points de vue traitement et ingénierie. Toutes ces fonctions sont implantées dans des objets, dans le système lui-même. Il n’y a donc pas de modèles séparés pour concevoir la fonctionnalité de l’application et les services techniques, et ceux-ci sont mis en oeuvre et encapsulés dans des objets distincts.

1.3 Généralités sur les modèles de composants répartis

Le modèle de référence RM-ODP, décrit dans la section 1.2, n’est pas directement implantable dans un système. Mais il sert à décrire et à raisonner sur des spécifications qui peuvent être implantées pour construire des systèmes répartis ouverts. Les modèles de composants répartis sont un exemple de telles spécifications.

Le concept de composant est une évolution du concept d’objet, pour étendre sa portée du domaine de la programmation, à celui de la conception d’applications

à une plus large échelle et à un plus haut niveau d'abstraction [Cha99]. Il existe plusieurs notions de composants, qui ont des objectifs différents. Par exemple, les modèles de *composants métier* servent à la construction d'applications d'entreprise, et ont globalement deux objectifs :

- séparer les différentes parties fonctionnelles des applications, pour les encapsuler dans des composants métier distincts,
- séparer l'implantation des composants du point de vue métier, de celle des fonctions techniques dans les autres points de vue.

Les systèmes applicatifs sont donc des configurations de composants. La notion de composant peut aussi être étendue pour décrire aussi les fonctions techniques (hors du point de vue métier) des systèmes, comme dans RM-ODP.

Les sections 1.3.1 et 1.3.2 décrivent les principaux objectifs des modèles de composant métier en général, en s'appuyant sur les principes de la conception par objets (voir section 1.1), et sur RM-ODP (voir section 1.2). La section 1.3.3 décrit et compare des modèles de composants métier répartis existants.

1.3.1 Architecture

L'*architecture* d'un système basé sur des composants est une description de haut niveau de ce système, notamment de sa décomposition en composants, de la fonctionnalité de ces composants, et de leurs interactions [SDK⁺95]. Cette définition correspond à celle donnée dans RM-ODP (voir en annexe A). Un composant est donc un module, qui est une unité de base pour raisonner sur un système à un haut niveau d'abstraction.

Le premier objectif principal des modèles de composants est de séparer l'architecture des systèmes, de l'implantation des composants qui constituent ces systèmes. En effet, les programmeurs d'un composant ne doivent pas se préoccuper de l'architecture globale de l'application, et l'assemblage des composants ne doit manipuler que des abstractions et ne pas se préoccuper des détails d'implantation. Donc la configuration d'une application, c'est-à-dire l'assemblage des composants entre eux, est réalisée hors du code des composants.

Les composants sont plus abstraits que les objets, et ont un plus gros grain. Il est même possible de programmer un composant sans utiliser un langage à objets [Cha99]. Mais le concept de composant reste une évolution du concept d'objet, dans la mesure où il applique certains principes des modèles d'objets [MK90], comme l'encapsulation. Certains modèles de composants permettent aussi de les composer de manière récursive [SDK⁺95], et de distinguer deux types de composants : les *composants primitifs*, non décomposables, et les *composites*, qui peuvent être décomposés en une configuration de composants. Cela correspond à la notion de composition des objets dans RM-ODP.

Une conception à gros grain permet de mieux gérer les dépendances entre les parties d'une architecture. En effet, les composants mettent en oeuvre complètement une fonctionnalité d'une application, et présentent chacun une forte cohésion. Les composants dépendent d'autres composants, mais ces dépendances sont identifiées explicitement et sont peu nombreuses, donc les composants sont peu couplés. Les composants utilisés dans une architecture ont une description

stable et bien définie, et sont donc fermés : cela permet d'appliquer systématiquement le principe de fermeture commune [Mar96b] à une architecture. Les composants peuvent désigner des objets de différentes natures, par exemple du code binaire, comme dans le modèle COM [MC95], ou du code source, ou même ne pas avoir de lien direct avec du code.

Un lien entre deux composants d'une configuration est généralement appelé *connecteur*. Deux composants ne peuvent interagir que s'ils sont liés par un même connecteur : un connecteur correspond donc à un lien (*binding*) dans RM-ODP. Les composants étant fermés, ces connecteurs sont des points stratégiques pour modifier une architecture. Ils permettent d'appliquer le principe d'ouverture-fermeture [Mey97], en permettant de connecter des composants différents, et des nouveaux composants, sans modifier le code des composants. Un connecteur peut être matérialisé ou pas au niveau de l'implantation, et effectuer ou pas des traitements sur les interactions entre des composants. Ces liens matérialisent les dépendances entre les composants. Ainsi, ces dépendances sont explicites et faciles à maîtriser pour appliquer certains principes de conception, comme le principe des dépendances acycliques [Mar96b].

Une architecture est définie en général de manière déclarative, hors du code des composants, dans un langage de description d'architecture (ADL, Architecture Description Language), qui formalise les notions de configuration, de composant et de connecteur. Mais d'autres concepts peuvent être formalisés dans un ADL [MT97], comme par exemple la notion d'interface.

Une *interface* est un ensemble de points d'interactions d'un composant ou d'un connecteur. Une interface peut spécifier un service fourni par un composant ou un connecteur, ou un service requis pour sa mise en oeuvre. Ce concept est donc similaire au concept d'interface du langage de traitement de RM-ODP. Lorsque des composants interagissent à travers un connecteur, leurs interfaces sont liées à celles du connecteur.

Les composants ne dépendent pas directement d'autres composants : ils dépendent d'interfaces d'autres composants ou d'interfaces de connecteurs, suivant le modèle de composant utilisé. Ils dépendent donc d'abstractions, qui sont stables car les composants et connecteurs sont fermés quand ils sont utilisés dans une architecture. Un modèle de composants permet donc d'appliquer systématiquement le principe de relation dépendance-stabilité [Mar97].

Certains ADLs sont accompagnés d'outils spécifiques pour générer du code pour les liens entre composants et connecteurs, offrir des vues multiples d'une architecture, ou pour analyser les architectures. Une partie du travail de construction d'une configuration est donc automatisée.

1.3.2 Séparation métier–technique

L'autre objectif principal des modèles à composants métier est de découpler la fonctionnalité des applications et les fonctions techniques – ou *services techniques* – qui mettent en oeuvre des transparences, comme la transparence de transaction [Cha99][ISO95b].

La mise en oeuvre des fonctions techniques dans une configuration de composant doit être paramétrable, et pouvoir s'adapter au rôle de chaque composant dans

une configuration. C'est pourquoi un composant encapsule à la fois son code fonctionnel, et des propriétés permettant de paramétrer son environnement. Un composant est donc auto-descriptif, et peut décrire lui-même les fonctionnalités qu'il met en oeuvre, les fonctionnalités qu'il requiert d'autres composants (ses dépendances), et ses propriétés techniques. Ce mécanisme d'auto-description permet l'application systématique du principe d'auto-documentation [Mey97] : l'information sur une configuration de composants est fournie avec ces composants.

Les services techniques sont mis en oeuvre par un support d'exécution spécifique, séparé des composants, et qui peut être appelé *conteneur*, qui est intégré dans un *serveur d'application*. Ce support d'exécution est essentiellement décrit dans les points de vue traitement et ingénierie de RM-ODP. Un modèle de composant spécifie les contrats d'environnement des composants, pour les interactions entre les composants et le support d'exécution. Ces spécifications peuvent par exemple spécifier des interfaces de programmation (APIs), pour les services offerts par le support d'exécution, ou pour que ce support interroge les composants sur leurs propriétés. Un modèle de composant spécifie donc des abstractions dont dépendent les composants et les supports d'exécution. Ces abstractions permettent donc d'appliquer le principe d'ouverture-fermeture [Mey97] : un support d'exécution peut exécuter n'importe quel composant respectant les spécifications, sans modification du code du support, et un composant peut être exécuté sur n'importe quel support qui respecte ces spécifications, sans modifier le code de ce composant.

Le processus de *déploiement* d'un système comprend plusieurs phases [CE00]. Par exemple, la phase d'installation met en oeuvre une configuration de composants sur un support donné, et paramètre les services techniques. Certaines de ces phases peuvent être partiellement automatisées par des outils spécifiques au support, qui permettent par exemple d'instancier un nouveau composant, de détruire un composant, de paramétrer les services techniques pour un composant, de paramétrer les connecteurs dans un système à partir d'une description dans un ADL, etc. Ainsi, ces outils masquent la technicité du support d'exécution, et offrent un haut niveau d'abstraction aux utilisateurs. Généralement, les spécifications de modèles de composants décrivent aussi les informations devant être fournies lors du déploiement et leur format, avec par exemple une définition d'un ADL. Les outils de déploiement respectant une même spécification offrent donc une interface homogène aux utilisateurs, ce qui facilite la portabilité des systèmes sur des supports différents.

Nous pouvons distinguer deux grandes classes de modèles de composants, suivant si les composants peuvent être répartis ou non. Un exemple de modèle de composants métier non répartis est le modèle Java Beans. Nous ne nous intéressons par la suite qu'aux modèles de composants répartis.

Ce qui distingue ces deux classes de modèles est l'ensemble des transparences offertes, et donc des services techniques mis en oeuvre dans le support d'exécution. Par exemple, la transparence de répartition n'est offerte que par les modèles de composants répartis. Les transparences pouvant être offertes par un support sont par exemple celles spécifiées dans RM-ODP (voir section 1.2.2).

1.3.3 Modèles de composants industriels

San Francisco d'IBM (SF) [And96], les Enterprise Java Beans (EJB) de Sun Microsystems [DmYK01], le modèle de composants CORBA (CCM) [ea99] et COM de Microsoft [MC95] sont les quatre modèles de composants industriels les plus courants. Ils sont bâtis sur les mêmes principes généraux, qui sont exposés dans cette section, et ils convergent rapidement (intégration SF-EJB, EJB-CCM, prise en compte des communications asynchrones par CCM puis EJB puis COM+, ...).

SF, EJB, CCM et COM sont des modèles de *composants applicatifs ou métiers*. Les composants représentent des processus (transferts bancaires, gestion d'ordres) ou des entités (compte bancaires, employés, clients, ...) utilisés couramment dans différents domaines d'activité d'une entreprise (finance, paie, gestion de stocks, de commandes, ...). Les composants sont déployés dans des structures d'accueil : les *conteneurs*¹. Les conteneurs ont deux rôles principaux :

- ils gèrent le cycle de vie (i.e. création, accès, activation, passivation, destruction, ...) des composants qu'ils contiennent,
- et ils fournissent à ces composants automatiquement et de manière transparente un certain nombre de services techniques (e.g. persistance, comportement transactionnel, concurrence, sécurité, etc.) par interception des interactions avec ces composants.

Le paramétrage de ces services techniques est effectué de manière déclarative, au moment du déploiement, au travers de *descripteurs de déploiement*². Les différents modèles décrivent principalement des *modèles de programmation* qui spécifient en réalité des *contrats* entre les composants et les conteneurs. Ils existent un certain nombre de *types de composants* prédéfinis dans chaque modèle (*sessions, entités, procédés, messages, ...*). Une *granularité tacite* est associée aux types de composants : par exemple, les entités sont de "petits composants", tandis que les procédés sont de "plus gros composants". Les contrats dépendent de ces types de composants. Ils spécifient notamment quels services techniques sont associés à chaque type de composants.

Il existe, bien sûr, de nombreuses différences entre les différents modèles, que nous ne décrivons pas toutes ici. Ces différences concernent les points abordés ci-dessus :

- types de composants considérés :
 - sessions sans états dans COM,
 - sessions, entités, messages dans EJB,
 - entités, dépendants, commandes dans SF,
 - sessions, entités, processus, services dans CCM.
- services techniques associés à chacun de ces types :
 - nommage, cycle de vie, transactions, sécurité dans COM,
 - cycle de vie, nommage, persistance, transactions, concurrence, sécurité dans EJB,
 - idem + notification d'événements dans CCM,

¹On peut considérer l'infrastructure transactionnelle MTS comme le conteneur des composants COM+.

²On parle également d'*aspects non fonctionnels*.

- idem + requêtes dans SF.
- modèles de programmation associés.

Parmi les différences notables, signalons que SF et COM, à la différence de EJB et CCM, ne sont pas des spécifications mais des produits (SF par exemple propose plus de 1100 composants directement utilisables). Signalons également que CCM et COM sont les seuls modèles à supporter des composants offrant des *interfaces multiples* (dont une par défaut, `QueryInterface` de `Unknown`, dans COM qui permet de naviguer entre les multiples interfaces des composants).

Si nous comparons les modèles de composants industriels aux modèles issus du domaine de l'architecture logicielle et des ADLs (voir section 1.3.1), force est de constater l'existence d'un fossé certain en défaveur des modèles industriels. La notion de connecteur est absente des modèles industriels. La notion de configuration, dans le sens donné par RM-ODP (voir annexe A), est également simplement inexistante (SF, EJB, COM) ou embryonnaire : il existe bien un concept d'*agencement* dans CCM mais il n'est pas formellement défini, et un agencement, (i.e. une composition de composants) n'est pas un composant. Nous touchons ici sans doute à l'une des plus grosses lacunes des modèles industriels. Ces modèles sont "plats" du point de vue de la composition : il n'y a pas de composition multi-niveaux. Un composant ne peut pas être lui-même décomposé en sous-composants.

SF, EJB, CCM et COM sont des modèles de composants métiers utilisables industriellement par des programmeurs d'applications. Ils sont bien adaptés à ce pour quoi ils sont prévus : la programmation, le déploiement et la réutilisation des aspects fonctionnels des systèmes répartis.

La section suivante décrit plus en détails l'un de ces modèles : les EJB.

1.4 Enterprise Java Beans

Les spécifications Enterprise Java Beans version 2.0 [DmYK01] décrivent un modèle de composants métier répartis, appelés *beans*, destinés à construire des applications réparties. Ces spécifications mettent l'accent sur la séparation du code des composants (le code métier), de leur assemblage et de leurs aspects techniques. Ces aspects techniques, précisés dans les spécifications EJB, sont décrits de manière déclarative, et sont mis en oeuvre par un support d'exécution spécifique, le *conteneur EJB (container)*.

L'objectif principal des EJBs étant la portabilité des composants sur des conteneurs différents, ces spécifications décrivent principalement :

- le modèle de construction des composants métier,
- les contrats (spécifications de comportement et d'interfaces de programmation) entre les composants et les conteneurs.

La section 1.4.1 décrit le modèle des beans, et la section 1.4.2 décrit la mise en oeuvre des services techniques par un conteneur pour ces beans.

1.4.1 Modèle de composants

Les spécifications EJB distinguent les composants décrivant les procédures et les entités métier, et proposent trois types de beans correspondants :

- les *session beans* modélisent des procédures métier, invoquées par les clients par invocation de méthodes à distance (RMI, Remote Method Invocation),
- les *entity beans* modélisent des entités métier persistantes, manipulées par RMI, et servent à spécifier les applications dans le point de vue information (voir section 1.2.1.2),
- les *message-driven beans* modélisent des procédures métier, invoquées par les clients par envoi asynchrone de messages (signaux).

Ces différents types de composants permettent une description des systèmes dans les points de vue métier, traitement (avec les session beans et message-driven beans) et information (avec les entity beans).

Ces types de bean diffèrent par les aspects suivants :

- le mode de programmation. Par exemple, les classes de beans doivent implanter des interfaces spécifiques à leur type de bean : la classe d'un entity bean doit par exemple implanter l'interface `EntityBean`.
- les types d'interactions avec les clients. Par exemple, les message-driven beans interagissent par des signaux (messages JMS, *Java Message Service*), et les autres types de beans par des opérations (appels de méthodes).
- les états des objets abstraits manipulés par les clients. Les message-driven beans n'ont pas d'état et sont toujours accessible directement par les clients, alors que les clients doivent explicitement créer des sessions et des entités, qui portent une identité unique, pour interagir avec un session bean ou un entity bean. Le cycle de vie de ces vues client est donc différent.
- le cycle de vie des instances de bean, qui est indépendant des vues client des beans, et qui est géré par le conteneur.
- la concurrence d'accès aux beans. Par exemple, plusieurs clients peuvent accéder simultanément à une même entité, mais pas à une même session.
- les services techniques mis en oeuvre par le conteneur. Par exemple, seules les entités sont persistantes, et les entités et les sessions ont plus de possibilités de paramétrage des modes d'exécution transactionnelle que les message-driven beans.

Le choix du type de bean est fait avant le développement d'un composant, et ne peut être modifié par la suite, car la classe du bean implante une interface spécifique à ce type de bean. Après son développement, un bean ne peut donc pas être étendu pour supporter de nouveaux types d'interactions, ou de nouveaux services (e.g. un message-driven bean ne peut jamais être persistant), sans modifier son code pour changer le type du bean : ceci viole le principe d'ouverture-fermeture (voir section 1.1), mais simplifie beaucoup le développement de ces beans et l'infrastructure logicielle.

La spécification des aspects techniques se fait en-dehors du code, de manière déclarative, dans un *descripteur de déploiement*, qui est un fichier dans un format XML (*eXtensible Markup Language*), dont la syntaxe et la sémantique sont définis dans les spécifications EJB. Un descripteur de déploiement est utilisé dans deux contextes :

- pour chaque bean : pour spécifier les propriétés techniques du composant,
- pour l’application entière : pour spécifier l’assemblage des composants, et les propriétés globales de l’application, comme les rôles de sécurité, etc.

Un bean fourni par un développeur est donc constitué de :

- une ou plusieurs classes en Java qui implament le bean, dont une classe principale qui sert à instancier les *instances de bean* proprement dites, et qui implante l’interface du type de bean (e.g. `EntityBean`, `SessionBean...`),
- dans le cas d’un entity bean ou d’un session bean, un ensemble de classes d’interfaces qui servent au conteneur à implanter la vue cliente du bean, et qui héritent des interfaces `EJBObject`, `EJBLocalObject`, `EJBHome` ou `EJBLocalHome`,
- un descripteur de déploiement qui indique les valeurs initiales par défaut des propriétés techniques du bean.

Les interfaces client (i.e. héritant de `EJBObject`, etc.) correspondent à une interface d’ingénierie des *sessions* (resp. des *entités*), qui sont les objets abstraits qui sont manipulées par les clients des session beans (resp. des entity beans). Ces objets abstraits ont une identité et un état propre. Nous pouvons considérer que le rôle du conteneur est de faire correspondre ces entités abstraites avec les instances du bean, et notamment de faire correspondre leurs cycles de vie.

Le modèle de composants des EJB est un modèle “plat” : il n’est pas possible de décomposer un bean comme une configuration d’autres beans. Tous les beans sont donc à un même niveau d’abstraction.

Un des principaux apports des dernières spécifications EJB (2.0), par rapport aux spécifications précédentes (1.1) est la possibilité d’exprimer l’état des entités comme des configurations persistantes d’objets. Ces objets persistants peuvent être d’autres entités ou des valeurs persistantes simples (e.g. des objets Java sérialisables). Ces configurations forment un *schéma abstrait de persistance*, qui est spécifié de manière déclarative dans les descripteurs de déploiement, indépendamment du support utilisé à l’exécution pour la persistance de ces configurations d’objets. Il est ensuite possible d’effectuer des requêtes sur ces schémas, dans le langage EJB QL, qui est indépendant des supports de persistance, et qui est une extension de SQL pour supporter les collections d’objets et la navigation des liens dans ces configurations d’objets. Lors de l’exécution du système, les schémas abstraits sont matérialisés par des services de persistance spécifiques au conteneur, et les requêtes EJB QL sont traduites dans un langage de requête spécifique à ce service de persistance. Il est à noter que ces configurations d’objets persistants ne sont pas encapsulées dans les entités, car elles peuvent être partagées par plusieurs entités dans le même schéma. Les entités ne sont donc pas exprimées comme des composites : le modèle reste “plat”.

L’ensemble des éléments constituant un bean est empaqueté dans une archive compressée dans un format bien spécifié (“`ejb-jar`”). Regrouper ensemble le code des beans et la description de leurs propriétés permet d’appliquer le principe d’auto-documentation (voir section 1.1).

La phase d’assemblage d’un système complet correspond à la production d’une spécification métier, dans RM-ODP (voir section 1.2.1.1). Elle consiste à choisir un ensemble de beans, et à produire un descripteur de déploiement décrivant l’assemblage et les propriétés de tous les beans, adaptés à l’application. Il n’est

donc pas nécessaire de coder pour assembler une application. Ce descripteur global est ensuite utilisé pour déployer l'application dans un conteneur donné, à l'aide des outils spécifiques à ce conteneur.

1.4.2 Architecture d'un conteneur EJB

Le rôle principal d'un conteneur EJB est de fournir des services techniques (non-métier) aux applications à base de beans qu'il exécute. Ces services sont fournis de manière transparente, par *interception* des interactions entre les clients et les beans. Cette interception se fait par l'*interposition d'objets* spécifiques, entre les clients et les beans. Ainsi, les clients n'interagissent jamais directement avec les beans, mais interagissent avec les objets d'interposition, qui à leur tour interagissent avec les instances de beans. Dans le point de vue ingénierie, ces objets d'interposition sont des talons serveur qui implantent les fonctions du conteneur.

Pour les session beans et les entity beans, ces objets d'interposition matérialisent les objets abstraits (sessions et entités) que manipulent les clients à travers les interfaces client du bean. Ils encapsulent donc l'état de ces objets abstraits, et l'état des instances de beans qu'ils manipulent est aussi géré par le conteneur. Les clients interagissent avec les message-driven beans à travers des files de messages respectant le standard JMS (Java Messaging Service). Ces files de messages interagissent avec un objet d'interposition du conteneur, qui met en oeuvre les services non-fonctionnels avant de délivrer les messages aux instances des beans. Les types d'interaction et les diagrammes d'état sont donc différents suivant les types de beans, et les objets d'interposition sont donc aussi spécifiques au type des beans dont ils interceptent les interactions.

Lorsqu'un client interagit avec un session bean ou un entity bean, il doit gérer explicitement le cycle de vie des objets abstraits qu'il manipule. Pour cela, il doit interagir avec l'objet d'interposition qui implante l'interface `EJBHome` de ce bean. Cette interface permet au client, entre autres, de créer une session ou une entité, et de créer un lien réparti (*binding*), à travers un canal, vers l'interface de l'objet d'interposition matérialisant cet objet abstrait. A chaque objet abstrait manipulé par un client correspond donc au moins un objet d'interposition implantant l'interface `EJBObject` du bean. Il peut ensuite détruire cet objet abstrait en interagissant avec l'interface `EJBHome` du bean, ou avec l'interface `EJBObject` de l'objet abstrait, en appelant des méthodes spécifiques.

Dans les dernières spécifications EJB 2.0 [DmYK01], a été introduite la notion d'interface locale. Ainsi, un session bean ou un entity bean peut offrir deux interfaces supplémentaires : `EJBLocalHome` qui correspond à `EJBHome`, et `EJBLocalObject` qui correspond à `EJBObject`. De la même manière, ces interfaces sont implantées par des objets d'interposition du conteneur, mais ne peuvent être utilisées que pour des interactions locales, entre des beans déployés dans le même conteneur. Ainsi, ces interfaces offrent des optimisations pour les interactions locales, et notamment permettent au conteneur de ne pas mettre en oeuvre de liens distants avec les clients interagissant avec ces interfaces, mais seulement des liens locaux.

La figure 1.7 illustre l'architecture globale d'un conteneur EJB, et les différents objets d'interposition qu'il fournit.

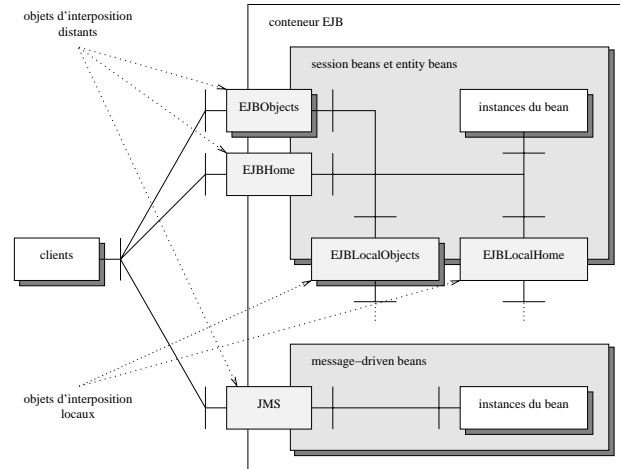


FIG. 1.7 – Architecture globale d’un conteneur EJB

Les spécifications EJB définissent les services techniques qui doivent être mis en oeuvre par chaque conteneur :

- *nommage* des beans (à travers un service de nommage JNDI, *Java Naming and Directory Interface*), et des objets abstraits (sessions et entités) manipulés par les clients (i.e. gestion des “clés” identifiant ces objets),
- *persistance* des entités et activation / passivation des sessions avec états et des entités,
- *gestion des transactions* pour tous les types de beans, avec des modes transactionnels différents suivant les types des beans, et suivant le paramétrage spécifié dans le descripteur de déploiement,
- *gestion des ressources*, notamment des ensembles d’instances de beans, des connexions aux bases de données, etc.
- *sécurité*, avec par exemple l’authentification des clients et des permissions pour interagir avec les beans,
- *répartition*, avec la mise en oeuvre de canaux répartis (e.g. CORBA) pour les interactions entre les clients et les interfaces EJBObject et EJBHome des session beans et entity beans, et de canaux JMS pour les interactions avec les message-driven beans. Les spécifications EJB 2.0 imposent que les conteneurs EJB supportent des canaux de type CORBA pour les interactions avec les session beans et entity beans.
- *gestion de la concurrence* d’accès, par exemple pour empêcher que deux clients interagissent simultanément avec une même session.

Ces contrats sont mis en oeuvre lorsqu’un client interagit avec un bean, à travers les objets d’interposition, ou lorsqu’une instance de bean accède à des ressources (e.g. une base de données). Ainsi, le conteneur intercepte toutes les interactions avec le code d’un bean : cette isolation permet de garantir la portabilité des composants sur des conteneurs différents. Les objets d’interposition d’un conteneur sont donc des points stratégiques de son architecture, car ils concentrent la mise en oeuvre des tous les services techniques, lors de toutes les interactions avec les beans. La figure 1.8 illustre cette isolation des instances de beans.

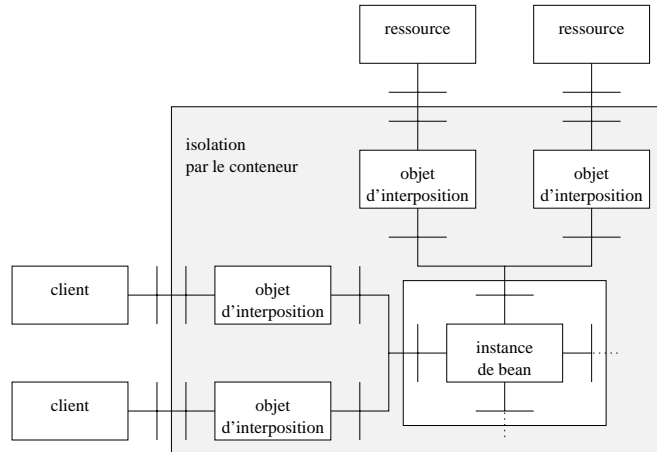


FIG. 1.8 – Isolation des instances des beans

Pour chacun des services techniques, des contrats entre le conteneur, les clients et les beans sont définis dans les spécifications. Ces contrats sont composés d'interfaces spécifiques aux services, et de diagrammes de séquence UML qui décrivent des traces de comportement qui doivent être observées à l'exécution. Les interfaces des services techniques sont des standards de Java, comme JNDI pour le nommage, JTA (*Java Transaction API*) pour la gestion des transactions, etc. et sont donc stables.

Ces contrats permettent de bien séparer la fonctionnalité des beans de l'implantation des services techniques, et donc cette implantation peut varier suivant les conteneurs EJB sans que le code des beans soit impacté. Ces contrats peuvent donc être considérés comme des points de conformance, dans le point de vue ingénierie de RM-ODP.

Il est à noter que l'ensemble des services techniques spécifiés dans les spécifications EJB est fixe et bien défini, et que ces spécifications ne décrivent aucun mécanisme permettant d'étendre cet ensemble de manière portable. De même, il n'est pas spécifié comment mettre en oeuvre des services techniques "avancés" tels que la migration de beans, comme décrit dans RM-ODP (voir section 1.2.2).

L'objectif principal des EJB étant la portabilité des beans, l'architecture interne des conteneurs n'est pas spécifiée, de même que l'organisation des services techniques, et elle est spécifique à chaque éditeur de logiciel qui produit un conteneur EJB. En effet, ceci n'a pas d'incidence sur la portabilité tant que le conteneur vérifie bien les points de conformance.

Les propriétés techniques pour chaque bean sont spécifiées dans les descripteurs de déploiement de manière déclarative, et correspondent à des paramètres pour ces services techniques. Ainsi, le code d'un bean est relativement indépendant du paramétrage des services techniques pour ce bean, et ceux-ci peuvent varier en général sans que le code du bean soit impacté. Ceci est vrai pour des services techniques comme la sécurité qui n'ont pas d'influence sur le mode de programmation. En revanche, le paramétrage de certains services techniques, comme la

gestion de concurrence d'accès à une instance de bean, peuvent être couplés à l'implantation du bean. Par exemple, si le code d'une classe de bean n'est pas réentrant, configurer le service de gestion de concurrence pour autoriser la réentrance donnerait des résultats incorrects, lors de son exécution.

Ainsi, les paramètres spécifiés par le développeur d'un bean dans son descripteur de déploiement, et la documentation du bean, guident l'assembleur d'un système dans la spécification des paramètres de chaque bean, sans pour autant donner des détails sur l'implantation du bean. Le code d'un bean reste donc toujours fermé, même si les développeurs peuvent limiter le paramétrage des services techniques pour ce bean à l'implantation.

Les clients d'un bean, qui peuvent être d'autres beans, ne dépendent que des interfaces distantes ou locales de ce bean, et pas de son implantation, car ils ne peuvent interagir avec ce bean qu'à travers ces interfaces. Ainsi, ces clients ne dépendent que d'abstractions, et peuvent être assemblés dans des configurations différentes, avec des beans qui implantent différemment ces interfaces. Les dépendances d'un bean sont spécifiées par des références ("ejb-ref") dans le descripteur de déploiement de ce bean. Ces références ont un nom logique unique dans le bean, et sont liées au type des interfaces dont dépend le bean. Ces références sont ensuite résolues lors de l'assemblage du système global, en faisant correspondre ces références logiques à des beans réellement déployés dans le système. Nous pouvons donc considérer que le langage des descripteurs de déploiement des systèmes EJB est un langage de description d'architecture (ADL) "limité", qui sépare l'assemblage des composants de leur implantation (voir section 1.3.1).

Une fois qu'un système EJB est assemblé, et que son descripteur de déploiement a été spécifié, le système est déployé dans le conteneur EJB. Cette phase de déploiement est spécifique au conteneur, car elle s'effectue en utilisant les outils spécifiques à ce conteneur, pour :

- raffiner le descripteur de déploiement, par exemple pour paramétrer les implantations de services techniques spécifiques au conteneur, ou spécifier les ressources à utiliser (e.g. le SGBD à utiliser pour l'activation/passivation des instances de bean...),
- planter et instancier les objets d'interposition pour les beans déployés,
- initialiser les contextes d'exécution des instances de beans, notamment pour créer les références entre beans,
- instancier les services, les canaux de communications, et enfin les beans.

La phase de déploiement est donc généralement automatisée par les outils des conteneurs, mais n'est pas du tout précisée dans les spécifications EJB 2.0. Les fonctionnalités offertes par ces outils diffèrent donc entre les conteneurs.

Chapitre 2

Techniques de séparation de problèmes¹

Nous avons présenté dans les sections précédentes le concept de composant, et montré comment il permet d'appliquer des principes de conception de manière systématique. Nous avons cependant montré, en particulier pour le modèle EJB, que tous les services techniques des supports d'exécution de composants sont mis en oeuvre simultanément dans les mêmes modules, les objets d'interposition dans le modèle EJB. Ces services techniques sont donc répandus dans tous les objets d'interposition, et sont mélangés dans chaque objet d'interposition. Il peut donc être difficile de séparer ces services techniques et de les encapsuler séparément, en utilisant seulement les paradigmes de la conception par objets. De plus, il peut être difficile de modifier ou d'ajouter un de ces "problèmes" (*concerns*) sans modifier le code existant : le code n'est pas complètement fermé.

L'idéal serait d'appliquer le principe d'ouverture-fermeture, en n'implantant dans un module qu'un seul "problème", comme cela est proposé dans le langage d'ingénierie de RM-ODP. Mais cela ne peut pas être réalisé systématiquement : certains problèmes sont difficiles à encapsuler dans un seul module (e.g. la gestion de la concurrence d'accès à un module se traduit par la présence de code dans ce module), et ces principes ne permettent pas une séparation et composition systématique des problèmes, y compris de problèmes non prévus initialement. Il faut donc utiliser des techniques spécifiques pour obtenir une flexibilité accrue pour la séparation et la composition des problèmes d'une application. Les sections suivantes décrivent certaines de ces techniques.

2.1 Réflexivité

La *réflexivité*, ou *réflexion*, est une démarche qui consiste à considérer un même système à deux niveaux d'abstractions :

- un *niveau de base*,

¹"séparation de problèmes" est une traduction de l'expression anglo-saxonne "separation of concerns"

– un *niveau méta*, ou *méta-niveau*.

Le niveau de base manipule les entités du domaine d’application premier du système, les objets, et décrit la fonctionnalité de l’application.

Le méta-niveau manipule des abstractions de ces entités du niveau de base. Ce niveau *réifie* (*matérialise*) ces abstractions dans des entités appelées *méta-objets*. Ces méta-objets permettent de raisonner sur les objets du niveau de base.

Ces méta-objets peuvent manipuler des structures du langage de programmation du niveau de base, comme les méthodes ou les classes. Par exemple, le langage Java fournit des classes (Method, Field, Class, etc.), qui permettent de raisonner sur la structure des objets Java, pendant l’exécution.

Mais ces méta-objets peuvent aussi manipuler des entités du support d’exécution du système. Par exemple, les systèmes Solaris et Linux permettent de charger et décharger explicitement, et pendant l’exécution, des bibliothèques de fonctions, à l’aide de fonctions en C (définies dans `dlfcn.h`), et de manipuler ces bibliothèques, notamment pour appeler les fonctions qu’elle contient. Or, cette notion de bibliothèque n’est pas spécifique aux langages de programmation, mais au système d’exploitation. Autre exemple, le langage Java fournit une classe Thread, qui réifie la notion d’activité, qui est spécifique à la machine virtuelle, c’est-à-dire au support d’exécution. Dans ce cas, nous pouvons considérer que le niveau de base est le support d’exécution, et que le méta-niveau est le système qui l’exécute.

Il est possible qu’un méta-niveau ait lui-même un méta-niveau (un “méta-méta-niveau”) : “tour de méta-niveaux” peut ainsi être formée.

Les interactions entre un niveau de base et son méta-niveau forment un *protocole méta*, ou *protocole à méta-objets* (*MOP*, *Meta-Object Protocol*), et s’appuient sur deux démarches :

- l’*introspection* permet au système de réifier des abstractions de son niveau de base dans des méta-objets, qui font aussi partie de ce système. L’*introspection* permet donc à un système de raisonner sur lui-même pendant son exécution.
- l’*intercession* permet aux méta-objets de modifier pendant l’exécution du système les propriétés des objets du niveau de base dont ils sont une abstraction. Donc des interactions avec des méta-objets, qui font partie du système, permettent de modifier les propriétés du système. L’*intercession* permet donc à un système de modifier lui-même ses propriétés pendant son exécution.

Le lien entre des méta-objets et les objets de base dont ils sont une abstraction, est appelé *lien méta*. Les interactions dans un protocole à méta-objets s’appuient donc sur des liens méta. Les interactions entre des objets dans un même niveau d’abstraction se font sur des liens “non-méta”.

Nous pouvons envisager d’appliquer la réflexivité à un modèle d’objets comme RM-ODP, en spécifiant pour chaque objet son niveau d’abstraction (de base ou méta) par rapport aux objets auquel il est lié, donc pour chaque lien (*binding*) s’il est ou non un lien méta, et donc aussi pour chaque interaction si elle fait partie ou pas d’un protocole à méta-objets. La séparation d’un système en niveaux de base et méta-niveaux dépend des objectifs de cette séparation, et il existe plusieurs séparations possibles pour un système.

De même que pour la notion d'*abstraction* définie dans RM-ODP (voir annexe A), nous considérons donc que la réflexivité est une démarche qui peut être utilisée pour raisonner sur les systèmes, ou par exemple sur les techniques de séparation et de composition de problèmes. Nous pouvons aussi considérer que la réflexivité est une propriété de ces techniques : une technique de séparation de problèmes "réflexive" est une technique qui met en oeuvre une démarche de réflexivité.

2.2 Programmation par aspects (AOP)

Un *aspect* est un *problème (concern)*, c'est-à-dire une partie d'un système qui représente complètement et exclusivement une propriété donnée du système [BSL][KLM⁺97]. Nous pouvons distinguer dans un système :

- un *aspect de base*, qui décrit le système du point de vue métier,
- des *aspects techniques*, ou *aspects non-fonctionnels*, qui décrivent le système dans d'autres points de vue, notamment le point de vue ingénierie. Les fonctions techniques définies dans RM-ODP (voir section 1.2.2), comme la gestion de transaction ou la sécurité, sont des exemples d'aspects techniques.

Dans un système, les différents aspects techniques qui le composent ne sont pas toujours encapsulables dans des modules distincts, ce qui se traduit par une *dispersion* des aspects, c'est-à-dire par un éparpillement du code de chaque aspect dans le système, et par un *mélange* du code de plusieurs aspects dans un même module et avec le code de l'aspect de base.

La *programmation par aspects (AOP, Aspect-Oriented Programming)* permet de construire un système en deux phases :

1. *séparation* des aspects, c'est-à-dire la définition de ces aspects indépendamment les uns des autres,
2. *composition (weaving)* des aspects techniques avec l'aspect de base, pour former le système global.

Comme les aspects sont composés après être définis, ils dépendent peu les uns des autres, et sont donc peu couplés. Les aspects forment des modules indépendants, et sont donc plus facilement réutilisables indépendamment. Il est à noter que la programmation par aspects peut être utilisée conjointement avec la programmation par objets, mais aussi avec d'autres paradigmes de programmation (e.g. fonctionnelle, procédurale...).

Les aspects techniques se traduisent par l'ajout de traitements dans les flots d'exécution de l'aspect de base, et donc par l'ajout de code de ces aspects dans le code de l'aspect de base pour modifier ces flots d'exécution. Les points dans le code de l'aspect de base où est inséré le code des aspects techniques sont appelés *points de jonction*, et peuvent être par exemple le début d'un appel de méthode, la fin d'un appel de méthode, ou une affectation de variable.

La composition d'aspects requiert une modification du code de l'aspect de base. Mais comme cette modification est automatique et transparente pour l'aspect de base, celui-ci reste fermé. Nous pouvons donc considérer que la programmation

par aspects est un mécanisme permettant de garantir l’ouverture-fermeture d’un aspect de base à de nouveaux aspects techniques (voir section 1.1), à condition que ces aspects aient été correctement séparés.

Comme les aspects techniques sont indépendants de l’aspect de base, il est nécessaire de *configurer* ces aspects techniques lors de la composition, pour faire correspondre l’exécution d’aspects à des points de jonction précis. Il est possible de configurer un point de jonction pour y mettre en oeuvre plusieurs aspects techniques simultanément.

Les différentes techniques de mise en oeuvre de l’AOP doivent spécifier :

- les langages pour programmer l’aspect de base et les différents aspects techniques. La définition des aspects dans un même système peut se faire dans des langages d’aspects différents, si la technique d’AOP le permet.
- les points de jonction disponibles dans le langage de l’aspect de base.
- si plusieurs aspects techniques peuvent être mis en oeuvre à un même point de jonction, comment ces aspects peuvent être combinés entre eux.
- un outil logiciel, le *tisseur* (*weaver*), qui permet de combiner les langages d’aspects pour construire un système, et qui est spécifique aux langages et aux points de jonction.

Suivant la technique d’AOP utilisée, la composition des aspects peut se faire :

- en modifiant le code source de l’aspect de base, avant ou pendant la compilation, pour inclure aux points de jonction du code source correspondant aux aspects techniques.
- pendant l’exécution de l’aspect de base, par réflexion.
- en utilisant ces deux techniques simultanément (technique “hybride”).

La composition par transformation de code source permet d’obtenir automatiquement un code source mélangeant tous les aspects, tel que le produirait un programmeur sans utiliser l’AOP. Dans ce cas, le tisseur peut être un compilateur spécifique ou un pré-compilateur.

La composition par réflexion place l’aspect de base dans un niveau d’abstraction de base, et les aspects techniques dans un méta-niveau. Ainsi, les aspects techniques sont implantés dans des méta-objets qui sont liés par des liens méta aux objets de base, dont ils adaptent le comportement. Par exemple, un méta-objet peut être invoqué lorsqu’une méthode est invoquée sur l’objet de base, pour mettre en oeuvre un aspect technique. L’ensemble des points de jonctions disponibles pour configurer les aspects est limité par l’ensemble des entités réifiables dans le méta-niveau, et donc par les capacités du protocole à méta-objets. En dehors de ces points de jonctions, le comportement des objets du niveau de base est identique à leur comportement s’ils n’étaient pas liés à des méta-objets.

Il existe des techniques d’AOP qui se basent sur une réflexion à la compilation, mais elles s’apparentent à une transformation de code source.

Quelque soit la technique de composition utilisée, il peut exister des conflits lorsque plusieurs aspects sont configurés à un même point de jonction. En effet, des organisations différentes des aspects à un même point de jonction peuvent produire des résultats différents à l’exécution. Les solutions pour résoudre ces conflits sont spécifiques aux techniques, et nécessitent l’intervention d’un individu humain pour guider la composition.

La programmation par aspects ne doit pas être utilisée pour construire complètement un système, mais seulement pour séparer et composer les aspects qui ne peuvent pas être encapsulés complètement dans un module [KLM⁺97]. Ainsi, il est nécessaire de pouvoir déterminer quels aspects sont encapsulables et quels aspects doivent être “mêlés” par AOP. Mais il n’existe pas encore de théorie pour résoudre ce problème de manière générale, et il doit être résolu pour chaque cas particulier. De même, il n’existe pas de démarche générale pour séparer les aspects entre eux.

2.3 Filtres de composition

Les *filtres de composition* (*composition filters*) [ABV92] sont basés sur un modèle d’objets étendu, et ont pour objectif de concevoir des systèmes extensibles. Nous pouvons comparer le modèle de filtre de composition au modèle de traitement de RM-ODP (voir section 1.2.1.3), bien que le modèle de filtre de composition soit antérieur à RM-ODP, ou aux modèles de composants.

Dans ce modèle, tous les objets communiquent entre eux à travers leurs interfaces, qui en sont des abstractions, avec des interactions de type signal. Comme cela est décrit dans RM-ODP, ces signaux servent à composer des interactions de type opération, mais l’unité d’interaction considérée ici est le signal. Dans ce modèle, sont encapsulés dans un même objet :

- un *objet noyau* (*kernel object*), qui implante la fonctionnalité de l’objet.
- des *filtres d’entrée* (*input filters*), qui effectuent des traitements sur les signaux reçus par l’interface.
- des *filtres de sortie* (*output filters*), qui effectuent des traitements sur les signaux émis par l’interface.
- des *objets internes* (*internal objects*), qui sont des variables d’instance typées, externes à l’objet noyau.
- des *objets externes* (*external objects*), qui sont des variables d’instance typées qui référencent d’autres objets.

L’objet noyau est la *partie implantation* d’un objet, alors que les filtres et les objets internes et externes font partie de la *partie interface* de l’objet. La partie implantation peut être implantée dans n’importe quel langage à objets classique (e.g. C++, Smalltalk, etc.). La partie interface a pour rôle d’étendre le comportement de la partie implantation, en s’appuyant sur une interception des signaux reçus et émis par l’objet, à l’aide des filtres.

Ces filtres sont donc comparables aux objets d’interposition dans le modèle EJB (voir section 1.4.2), qui interceptent les interactions et étendent le comportement des composants, pour mettre en oeuvre des services techniques.

La spécification des caractéristiques des parties implantation et interface des objets se fait au niveau des classes, et non pour chaque objet individuellement. De plus, il existe une notion d’héritage, comme dans les modèles à objets classiques, ce qui permet une certaine réutilisation de ces caractéristiques.

Les signaux émis et reçus par les objets sont typés, et contiennent des paramètres. Les filtres d’entrée (resp. de sortie) forment une file. Chaque signal reçu

(resp. émis) est traité par tous les filtres de la file d'entrée (resp. de sortie). Chaque filtre comporte plusieurs parties, qui déterminent quels signaux sont considérés ou ignorés par ce filtre (e.g. en fonction du type du signal, de sa destination, etc.), les actions à effectuer si le message est accepté par le filtre, et la nouvelle destination du signal (e.g. le message est rejeté, ou transmis au prochain filtre dans la file, ou à l'objet noyau, ou à un objet externe, etc.).

Les filtres peuvent être configurés et paramétrés pour une file d'une classe donnée, mais l'ensemble des types de filtres utilisables est prédéfini et non extensible. La configuration de la partie interface, et notamment des filtres, est réalisée de manière déclarative, en dehors du code des objets.

L'objet noyau offre des *méthodes*, qui traitent les messages renvoyés par les filtres d'entrée, pour implanter la fonctionnalité de l'objet, et des *conditions*, qui sont des méthodes sans effet sur l'état de l'objet noyau, qui retournent des valeurs booléennes décrivant son état. Les conditions peuvent être utilisées dans les filtres, pour adapter leur comportement.

Ainsi, les filtres peuvent avoir un comportement paramétré par l'état de l'objet, et par son environnement. Nous pouvons donc envisager de mettre en oeuvre avec les filtres de composition un modèle d'objets proche du modèle de traitement de RM-ODP, avec par exemple l'instanciation d'"interfaces" pendant l'exécution (i.e. en acceptant de nouveaux types de messages), la notion de contexte contractuel entre les objets, etc. Les filtres de composition offrent donc une grande richesse pour étendre les objets, avec un modèle simple.

De plus, en permettant de spécifier des traitements en-dehors du code des objets noyaux, dans les filtres, les filtres de composition permettent d'étendre le comportement des objets sans modifier leur code : ils permettent donc une ouverture-fermeture des objets à de nouveaux comportements.

2.4 Programmation par sujets (SOP)

La *programmation par sujets (SOP, Subject-Oriented Programming)* [OHBS94] est liée à la programmation par objets, et a pour objectif de pouvoir définir des classes d'objets de manière décentralisée.

Un *sujet* est une "vue subjective" d'un ensemble de classes, et est constitué de classes ou de fragments de classes, qui correspondent à un contexte d'utilisation d'un système. Un sujet est donc un "problème" (*concern*), qui concerne plusieurs parties d'un système. Nous pouvons considérer qu'un sujet est similaire à un aspect (voir section 2.2) dans la programmation par aspects. En effet, un sujet n'est pas un module bien identifié faisant partie d'un système, mais il définit une partie du comportement et de l'état d'un ensemble d'objets du système à l'exécution.

Chaque sujet est défini par un groupe de développeurs, selon ses propres besoins du système : c'est donc un développement guidé par les besoins (*requirement-based development*). Ainsi, la programmation par sujets résout un des problèmes de la conception par objets, où un seul groupe de développeurs est responsable du développement d'une classe donnée, et donc décide seul des extensions à

apporter à cette classe. Cette caractéristique de “possession” des classes peut se traduire par des conflits entre les utilisateurs de la classe dans un contexte spécifique, et les développeurs. La conception par objets n’est donc généralement pas adaptée à la construction de grands systèmes, car les conflits augmentent avec la taille des systèmes. Ainsi, les systèmes construits par sujets ne sont pas monolithiques, et leur conception n’est plus centralisée comme dans la conception par objets classique.

Chaque objet d’un système est défini par des opérations et des variables d’état spécifiques aux sujets qui l’implantent. Si plusieurs sujets définissent des fragments pour une même classe d’objets, chaque fragment d’un sujet ne manipule qu’une vue subjective de l’objet, que nous pouvons qualifier d’“objet subjectif”, et ne peut pas manipuler les objets subjectifs spécifiques aux autres sujets. Ainsi, chaque sujet reste fermé. Dans un système composé de plusieurs sujets, chaque objet sera composé en réalité d’un ou plusieurs objets subjectifs, avec le même identifiant d’objet. Un objet aura donc un comportement et un état qui seront une composition des comportements et des états définis par les fragments des différents sujets.

Chaque sujet est construit avec un langage de programmation par objets et un compilateur spécifiques, et il est envisageable que des sujets différents, composés dans un même système, soient développés dans des langages différents. Chaque sujet est compilé séparément, avant d’être composé avec d’autres sujets pour former un même système. La *composition des sujets* est réalisée en composant le code compilé des sujets.

Les différents sujets dans un même système interagissent entre eux. En effet, un sujet peut demander l’exécution d’une opération par un autre sujet, ou des sujets différents peuvent partager des mêmes variables d’état. Des sujets peuvent donc modifier leurs états et comportements respectifs. Il faut donc définir des *règles de composition*, pour garantir la cohérence d’une composition.

En plus de son code compilé, chaque sujet est composé de différents éléments, qui permettent sa composition :

- un *schéma*, qui définit les classes fournies et/ou utilisées par le sujet, notamment leur hiérarchie et leurs variables d’instance, du point de vue de ce sujet.
- des *interfaces*, qui définissent la signature des opérations fournies et/ou utilisées par un sujet.
- une spécification de la *structure* du sujet, qui donne des détails notamment sur les relations entre les opérations définies dans le schéma, et les méthodes dans le code compilé.

Les schémas et interfaces d’un sujet permettent de contrôler les dépendances entre les sujets, qui sont résolues par les règles de composition. Ce sont des abstractions stables, dont dépendent les sujets, et donc nous pouvons considérer que la programmation par sujets permet d’appliquer le principe de relation dépendance-stabilité [Mar97], et rend les sujets réutilisables dans des compositions avec des sujets différents.

Les règles de composition sont définies de manière déclarative, et permettent de spécifier des correspondances entre les classes et les opérations définies dans les schémas et les interfaces des différents sujets. La richesse d’expression dans

les schémas, les interfaces et les règles de composition dépendent des implantations spécifiques de programmation par sujets. La composition est réalisée automatiquement par des outils logiciels, qui génèrent du code d'adaptation pour implanter les règles de composition.

Comme chaque sujet est compilé séparément, et que ce code compilé n'est pas modifié lors de la composition des sujets, nous pouvons considérer que chaque sujet est fermé [Mey97], et un sujet peut être composé avec de nouveaux sujets sans modifier son code. La programmation par sujets permet donc "d'ouvrir-fermer" un système à la composition avec de nouveaux sujets.

La programmation par sujets semble bien adaptée à la composition de sujets qui décrivent chacun un même système du point de vue métier : c'est comme cela que [OHBS94] présente cette technique. Mais il nous semble qu'il est difficile d'implanter des services techniques comme des sujets séparés. Par exemple, un service de gestion de concurrence d'accès aux objets nécessite de réaliser des traitements dans chaque méthode de chaque objet, pour manipuler des verrous par exemple. Si ce service technique est défini comme un sujet séparé, il faut que ce sujet définisse des méthodes pour chaque méthode du "sujet de base", et il faut ensuite définir des règles pour chaque correspondance entre les méthodes du sujet de base, et les méthodes du sujet de gestion de concurrence. Cela devient plus complexe et coûteux que la programmation par objets. Nous considérons donc que cette technique n'est pas utilisable pour résoudre notre problème de séparation et de composition de services techniques, lorsque le code des services techniques est dispersé dans tout le système.

Un autre point négatif dans la programmation par sujet est que les implantations actuelles sont trop limitées dans l'expressivité des règles de composition, et que les conflits lors de la composition sont difficiles à résoudre, comme dans la programmation par aspects.

Chapitre 3

Synthèse

Nous avons décrit dans le chapitre 1 les propriétés souhaitées des systèmes à objets, et en particulier des systèmes répartis à objets. Nous avons introduit des principes de conception, dont le respect permet de garantir des propriétés de ces systèmes, comme la flexibilité ou la stabilité. Puis nous avons présenté le standard RM-ODP, qui spécifie les concepts et les caractéristiques spécifiques aux systèmes répartis ouverts.

Nous avons ensuite présenté le concept de composant, et des modèles de composants répartis, et montré informellement comment ces modèles permettent d'appliquer systématiquement certains principes de conception, et donc de garantir des bonnes propriétés des systèmes à composants. En particulier, nous avons montré comment les modèles de composants permettent de séparer la mise en oeuvre des services techniques (“fonctions” dans RM-ODP), de l'implantation de la fonctionnalité des systèmes dans les composants.

Nous avons détaillé en particulier l'un de ces modèles de composants : les Enterprise Java Beans 2.0. Notamment, nous pouvons observer que l'objectif global des EJB, bien que non explicite dans ces spécifications, est d'appliquer certains principes de conception par objets aux systèmes (voir section 1.1), par exemple :

- l'*ouverture-fermeture* [Mey97] des conteneurs et des beans, grâce à des spécifications d'interfaces et de contrats de comportement qui forment des points de conformance pour que les beans soient portables sur des conteneurs différents, et pour que les conteneurs puissent supporter des beans différents.
- la *séparation des interfaces* des conteneurs [Mar96c], pour que chaque service technique soit accessible à travers une interface spécifique standardisée (e.g. JNDI, JTA, etc.).
- la *stabilité des abstractions* [Mar97], c'est-à-dire la stabilité des interfaces dont dépendent à la fois les clients, les beans et les conteneurs. Ces interfaces sont celles des EJB (dans le paquetage `javax.ejb`) et des services techniques. En effet, ces interfaces sont des standards de fait, donc fermées, et elles sont abstraites car indépendantes des détails d'implantation. En effet, par exemple, les EJB ne spécifient pas comment les conteneurs sont implantés, mais en spécifient une abstraction.
- la *relation dépendance-stabilité* [Mar97], pour les dépendances vers ces interfaces standards, car ces interfaces sont plus stables que les clients, conteneurs

et composants qui en dépendent. En effet, par exemple, il peut y avoir plusieurs composants qui respectent les interfaces et contrats spécifiés dans les EJB, et ces abstractions sont communes et identiques pour tous les composants : l'implantation des composants est donc plus variable que ces interfaces.

Les spécifications Enterprise Java Beans 2.0 ne référencent pas RM-ODP. Mais nous pouvons considérer qu'elles sont une spécification de traitement et d'ingénierie. Cependant, la notion de conteneur n'est pas clairement définie, et l'architecture d'un conteneur n'est spécifiée que superficiellement (i.e. en définissant la notion d'objet d'interposition).

Cependant, les spécifications EJB, comme les autres modèles de composants que nous avons présentés, n'appliquent pas tous les principes de conception souhaitables (e.g. le principe de substitution de Liskov pour les composants ou les services techniques), et la spécification de leur architecture pose des problèmes.

En effet, dans les EJB, tous les services techniques sont mis en oeuvre simultanément dans les mêmes modules : les objets d'interposition. Ces services techniques sont donc mélangés dans ces objets d'interposition, et sont chacun répandus dans tout le conteneur.

Pour résoudre ce problème, il peut être nécessaire d'utiliser des techniques spécifiques, comme celles que nous avons présentées dans le chapitre 2.

Ces *techniques de séparation et de composition de problèmes* permettent de contrôler les dépendances entre des problèmes (*concerns*) d'un système, lorsque ces problèmes ne peuvent pas être encapsulés dans un même module, mais sont mélangés et éparpillés dans tout le système.

Elles permettent notamment d'appliquer le principe d'ouverture-fermeture à ces problèmes, pour permettre la considération de nouveaux problèmes sans modifier le code existant. L'utilisation de ces techniques peut donc être intéressante pour la construction de parties des supports d'exécution de composants, tels que les conteneurs EJB.

Cependant, toutes les techniques de séparation et de composition de problèmes ne sont pas utilisables pour séparer les services techniques dans un conteneur. Par exemple, nous avons montré que la programmation par sujets (SOP, voir section 2.4), n'est pas adaptée à la séparation de services techniques tels que la gestion de concurrence.

De plus, la mise en oeuvre de ces techniques peut être coûteuse et complexe, notamment pour gérer les conflits lors de la composition des problèmes. De même que l'application des principes de conception à tout un système (notamment le principe d'ouverture-fermeture) n'est pas souhaitable pour les mêmes raisons, il n'est pas souhaitable d'utiliser de telles techniques de composition pour construire complètement un système.

Il est préférable d'utiliser plusieurs démarches, dont l'utilisation de patrons de conception réutilisables (*design patterns*), et de n'appliquer des techniques de séparation et de composition de problèmes qu'à des points stratégiques, où les autres démarches ne sont pas applicables.

Deuxième partie

Construction d'un conteneur EJB adaptable et extensible

Chapitre 4

Extension du modèle EJB

Les EJB sont répandus dans le monde industriel, notamment grâce à leur simplicité et à la portabilité des applications produites. Mais nous avons montré (voir sections 1.3.3 et 1.4.1) que cette simplicité se traduit par des limitations, en particulier dans le modèle de composant des beans.

Ce chapitre décrit quelques-uns des problèmes des spécifications EJB actuelles, et propose des extensions au modèle EJB pour résoudre ces problèmes. Nous nous basons ici sur le modèle RM-ODP, présenté dans la section 1.2, et sur les principes de conception par objets présentés dans la section 1.1.

4.1 Services techniques

Les spécifications EJB 2.0 définissent un ensemble fixe et bien défini de services techniques devant être mis en oeuvre par un conteneur EJB (e.g. gestion de transaction, etc.). Mais cet ensemble de services n'est pas extensible, et l'organisation de ces services à l'intérieur d'un conteneur n'est pas spécifiée. Ceci est principalement dû à l'absence de modèle pour construire ces services, et à l'absence de spécification de l'architecture d'un conteneur.

Nous proposons de fournir une spécification d'un modèle pour implanter les services techniques. Nous souhaitons séparer ces services les uns des autres et les encapsuler séparément. Ce modèle sera dérivé de celui présenté dans le langage de traitement de RM-ODP, en faisant communiquer les services entre eux seulement par leurs interfaces, et en spécifiant le comportement des services dans ces interfaces (e.g. avec des préconditions, postconditions et invariants). Cette spécification spécifiera aussi comment ces services seront composés entre eux.

La spécification que nous allons proposer nous permettra donc de construire un conteneur EJB modulaire et "ouvert-fermé" à l'ajout de nouveaux services. Une telle spécification est naturellement compatible avec les spécifications EJB actuelles, car notre proposition ne concerne que la construction interne du conteneur, et ne modifie pas les contrats spécifiés dans les EJB entre les beans, les clients et les conteneurs.

La section 5.1 décrit plus en détails l'impact de cette proposition sur l'architecture de notre conteneur.

4.2 Interactions et dépendances entre beans

Dans les spécifications EJB 2.0, les références vers des beans peuvent être établies explicitement par interrogation d'un service de nommage, ou implicitement en passant en paramètre un talon de bean. En effet, si un talon de bean est passé en paramètre vers une machine distante, le récepteur créera implicitement une liaison entre le talon reçu et le bean. Mais au sens du langage de traitement ODP, nous pouvons considérer que les liens (bindings) entre beans sont toujours implicites, car les beans n'ont pas d'interfaces explicites pour les interactions qu'ils initient (i.e. lorsqu'ils envoient des messages ou appellent des méthodes à distance). Cela correspond directement au modèle objet de Java.

L'obtention explicite d'une référence vers un bean se fait par une requête à un service de nommage, qui crée un talon approprié. Cette requête se fait dans le code du bean, et correspond à un nom logique défini hors du code dans le descripteur de déploiement (éléments *ejb-ref*). A ce nom sont associés les types d'interface Java du bean à lier. Ceci permet de vérifier lors du déploiement que les beans liés ont bien les types requis, et donc de vérifier statiquement que les assemblages de beans sont valides.

Mais les interfaces des beans sont souvent fortement liées aux beans eux-mêmes, parce que les beans sont souvent livrés directement avec les interfaces qu'ils implantent. Ceci implique que les autres beans qui les utilisent sont souvent dépendants directement de ces beans. De plus, même si plusieurs beans implantent la même interface, ceux-ci sont difficilement interchangeable car les interfaces de beans ne spécifient que la structure des services fournis, mais pas leur comportement. Ceci ne permet donc pas d'assurer l'application du principe de substitution de Liskov, et limite la modularité des assemblages de beans.

Pour garantir une plus grande flexibilité, nous proposons d'étendre le modèle des EJB, pour nous rapprocher du modèle décrit dans le langage de traitement de RM-ODP. Notamment, nous envisageons l'utilisation d'interfaces pour toutes les interactions auxquelles participe un bean, y compris pour les interactions qu'il initie. Ces interfaces spécifient leur comportement, avec par exemple des préconditions, postconditions et invariants. Ces mécanismes permettent de renforcer la validité des assemblages, et permettent d'appliquer le principe de substitution de Liskov aux beans. Les préconditions, postconditions et invariants des interfaces peuvent être spécifiées déclarativement dans les descripteurs de déploiement, avec des éléments XML spécifiques à notre conteneur qui seraient exploitées par les outils de déploiement.

Notre proposition impacterait :

- les développeurs de beans, dans la mesure où l'utilisation d'interfaces pour les "services requis" impose un autre modèle de programmation que Java pour initier des interactions avec un bean,
- les outils de déploiement du conteneur, dans la mesure où ces interfaces leur permettent de garantir statiquement la validité structurelle d'un assemblage.

Une telle évolution reste compatible avec les beans existants, car nous pouvons utiliser un mécanisme de lien implicite avec création automatique d'une interface lors des interactions, bien que nous ne puissions pas garantir une compatibilité de comportement avec de tels beans. Les préconditions, postconditions et invariants sont spécifiés à l'aide d'une extension au format des descripteurs de déploiement, et donc n'impactent pas les applications existantes.

4.3 Interfaces locales et transparence de répartition

Dans les dernières spécifications EJB 2.0, a été introduite une distinction entre les accès distants à un bean et les accès locaux (entre des beans dans une même machine virtuelle Java). Ces accès se font toujours à travers des interfaces, implantées par le conteneur EJB pour mettre en oeuvre les services techniques. Ainsi, même entre des beans locaux, il y a toujours une transparence pour des services comme la gestion de transaction ou la sécurité.

La notion d'interface locale a été introduite pour améliorer les performances lors des communications locales entre beans. Notamment, cette distinction permet d'améliorer les performances d'accès aux beans de grain fin, en particulier dans les schémas abstraits de persistance. Les spécifications EJB conseillent l'utilisation d'interfaces locales pour les beans de grain fin qui communiquent localement, et d'interfaces distantes pour les beans de gros grain. Ainsi, la granularité d'une architecture est liée au mode de répartition des beans, ce qui peut limiter la flexibilité des architectures.

Ces interfaces distantes et locales sont distinctes, et ont les caractéristiques suivantes :

- un bean peut avoir une interface locale ou une interface distante, ou les deux. Ce choix se fait au développement du bean.
- les interfaces locale et distante d'un bean sont structurellement différentes : les interfaces locales héritent de `EJBLocalObject` et `EJBLocalHome`, alors que les interfaces distantes héritent de `EJBObject` et `EJBHome`.
- les interfaces locale et distante d'un bean peuvent offrir des méthodes métier différentes, et donc des services différents.

De plus, les accès se font dans deux modes de programmation différents :

- les exceptions Java levées par les méthodes des beans sont différentes : `RemoteException` pour les interfaces distantes, `EJBException` pour les interfaces locales.
- les modes de passage de paramètres sont différents : par valeur pour les interfaces distantes, par référence pour les interfaces locales.

A cause de ces caractéristiques, un client ne peut pas utiliser indifféremment l'interface distante ou l'interface locale d'un bean, sans que son code soit modifié. Cela diminue donc la flexibilité des architectures.

De plus, si une interface distante n'a pas été développée lors du développement d'un bean, celui-ci ne pourra pas être accessible à distance. Ainsi, des choix faits

lors de la conception d'un bean limitent son mode de répartition. Il n'y a donc pas de séparation entre le développement d'un bean et cet aspect technique. Et à cause des différences de structure et de mode de programmation, la répartition des beans n'est pas transparente.

Nous souhaitons rester compatibles avec ces spécifications. Mais nous envisageons de permettre aux développeurs qui utilisent notre conteneur de conserver une transparence de répartition, en leur permettant d'utiliser toujours les interfaces distantes, avec de bonnes performances pour les interactions locales.

Pour optimiser les communications entre beans locaux, nous envisageons d'inclure des optimisations dans les canaux de communications, au niveau des lieux ou des objets de protocole, de manière transparente, comme cela est mis en oeuvre dans la "personnalité" Jeremie de la couche de communication Jonathan [DHTS98], qui fait partie du projet Objectweb (<http://www.objectweb.org/>), et qui implante une couche compatible avec Java RMI, et qui est utilisée dans le serveur JOnAS. Une optimisation envisageable est d'utiliser des objets de protocole triviaux qui communiquent directement entre eux. De plus, cette optimisation des canaux de communication peut se faire statiquement, car les liens entre les beans, dans une architecture donnée, sont connus dès le déploiement

4.4 Composition des beans

Une raison pour laquelle les interfaces locales ont été introduites dans les spécifications EJB 2.0 est un besoin de supporter de manière efficace des beans avec un grain fin, notamment dans les schémas abstraits de persistance. Mais nous avons vu ci-dessus que l'introduction de la notion d'interface locale brise la transparence de répartition.

D'autre part, les schémas abstraits de persistance sont définis comme des architectures d'objets persistants. C'est-à-dire que les états persistants des entités sont des ensembles d'objets liés entre eux. Or, ces objets persistants peuvent être partagés entre des entités dans le même schéma : il n'y a pas d'encapsulation. Ainsi, la modification de l'état d'une entité peut modifier l'état d'une autre entité. Le problème est qu'il n'y a pas de notion de contrat de comportement dans les spécifications EJB. Ainsi, rien ne permet de garantir que les modifications d'états se feront de manière cohérente. Cela impose donc que les beans partageant des mêmes objets persistants soient fortement couplés.

Pour supporter des beans avec un grain fin, de manière efficace et en respectant le principe d'encapsulation, nous proposons d'étendre le modèle EJB par un mécanisme de composition des beans, utilisable à la fois à l'intérieur et à l'extérieur des schémas de persistance. Le premier intérêt d'un tel mécanisme est l'introduction d'abstractions dans les architectures de beans, pour améliorer leur compréhension.

De plus, d'après le principe d'encapsulation, tous les objets, y compris les objets composites, ne peuvent interagir qu'entre leurs interfaces, et l'état d'un objet ne peut être modifié que par une action de cet objet. Or, les objets composant un composite font partie de son état. Ainsi, un objet dans un objet composite ne peut interagir qu'avec des objets à l'intérieur du même composite. Les

beans composant un composite étant définis statiquement, nous envisageons donc d'optimiser statiquement les canaux de communications pour les communications entre les objets dans un composite. Par exemple, si tous les beans formant un même composite sont déployés dans un même serveur EJB, ces beans ne participent jamais à des interactions distantes, et il n'est donc pas nécessaire de mettre en oeuvre des canaux de communication distante pour ces beans. D'autre part, la composition permettrait aussi d'isoler les états des différentes entités dans un même schéma de persistance, et donc de les découpler systématiquement. Le mécanisme de composition que nous souhaitons introduire servirait donc essentiellement à isoler des configurations de beans de manière transparente aux programmeurs.

Nous envisageons l'introduction de ce mécanisme seulement par ajout de code, et de manière compatible avec les beans et spécifications actuels. En effet, il suffirait d'ajouter des déclarations spécifiques dans les descripteurs de déploiement pour spécifier les relations de compositions, qui seraient interprétées par des outils de déploiement adéquats. Ces compositions seraient utilisées par le code des beans de manière identique aux liens entre beans tels qu'ils sont définis par les spécifications EJB 2.0 actuelles. Il n'y aurait donc pas de modification du mode de programmation actuel des beans.

4.5 Conclusion

Dans cette section, nous avons présenté des extensions au modèle des EJB, qui conservent la compatibilité avec les spécifications EJB 2.0 actuelles.

Notre première proposition est d'étendre le modèle EJB avec une spécification de l'architecture interne d'un conteneur, notamment pour pouvoir introduire de nouveaux services techniques de manière transparente. Notre proposition d'ajouter des interfaces pour les interactions initiées par les beans, ainsi que la possibilité de spécifier déclarativement des préconditions, postconditions et invariants sur les interfaces des beans, nous permettra de renforcer la sémantique des assemblages de beans. Notre proposition d'introduire des optimisations dans les canaux de communication entre beans locaux, et le concept de composition de bean, nous permettront d'optimiser statiquement les liaisons entre les beans, et de supporter de manière transparente des beans avec un grain fin.

Ces extensions ont un impact minimal sur l'architecture de conteneur que nous proposons. Cette architecture est détaillée dans le chapitre suivant.

Chapitre 5

Proposition d'architecture d'un conteneur EJB

Les spécifications des EJB évoluent dans le temps pour enrichir le modèle des beans, notamment pour s'adapter aux problèmes rencontrés par les utilisateurs lors du développement de leurs systèmes. Entre les versions 1.1 et 2.0 de ces spécifications, les évolutions suivantes ont par exemple été apportées (voir section 1.4) :

- l'ajout du support de schémas de persistance pour décrire l'état des entités,
- l'ajout du support de CORBA en tant que type de canal de communication, en plus de Java RMI,
- l'ajout du support d'interactions de type signal (message-driven beans),
- l'ajout du concept d'interface locale.

Les nouvelles versions de spécifications sont compatibles avec les versions précédentes, et sont principalement constituées d'extensions. Ces spécifications restent donc globalement fermées, et les évolutions n'impliquent pas de violation de leur fermeture.

Mais ces extensions peuvent avoir un impact important sur un conteneur EJB si celui-ci n'est pas flexible. Par exemple, l'introduction du support de CORBA dans un conteneur compatible EJB 1.1 induira des modifications d'autant plus importantes dans son architecture que ce conteneur a des dépendances fortes vers les couches de communications qu'il supporte déjà (e.g. Java RMI).

De plus, nous avons montré (voir section 4.1) que l'ensemble des services techniques mis en oeuvre par un conteneur EJB est, d'après les spécifications EJB, fixe et non extensible. Mais les capacités de paramétrage de ces services spécifiées dans les EJB, peuvent ne pas suffire à couvrir tous les besoins des utilisateurs. Par exemple, les spécifications EJB ne supportent pas des transactions imbriquées ou ouvertes. A l'inverse, si une application n'utilise pas un service technique donné (e.g. les schémas abstraits de persistance), il est inutile de le fournir dans le conteneur, au risque de consommer inutilement des ressources, qui sont critiques dans certains environnements (e.g. la mémoire est limitée sur les ordinateurs de poche). L'ensemble des services offerts doit donc être configurable.

De même, il est souhaitable qu'un conteneur puisse supporter des extensions au modèle EJB, comme celles que nous avons proposées dans le chapitre 4, pour offrir un degré d'expressivité adapté aux besoins des utilisateurs. Il doit être aussi suffisamment flexible pour supporter de futures évolutions du modèle EJB standard.

Pour répondre à ces besoins, nous proposons de construire un serveur EJB flexible, pouvant évoluer pour être adapté aux changements de spécifications, d'environnement et de besoins des utilisateurs. Comme les EJB ne spécifient pas l'architecture d'un serveur EJB, nous en proposons une dans ce chapitre, qui nous permettra de garantir une certaine flexibilité.

Dans tout ce chapitre, nous ne distinguons pas les notions de "conteneur EJB" et de "serveur EJB". Nous utilisons donc indifféremment ces deux termes pour désigner un support d'exécution de composants respectant les spécifications EJB. Nous proposons tout de même une distinction, par comparaison avec le langage d'ingénierie de RM-ODP, dans la section 5.8.

Pour justifier nos choix d'architecture, nous nous sommes basés sur l'analyse d'un conteneur EJB existant : le serveur JOnAS, du projet Objectweb (<http://www.objectweb.org/>). L'architecture du serveur JOnAS n'est pas documentée, et les fichiers sources contiennent peu de documentation : cette implantation ne respecte donc pas le principe d'auto-documentation [Mey97]. Nous avons donc dû analyser nous-même cette architecture. Ce serveur EJB est un logiciel libre, donc nous avons pu analyser ses fichiers sources.

Nous avons analysé les dépendances d'import (clauses *import* dans les fichiers source Java), entre les classes Java constituant le serveur JOnAS. Nous avons ensuite synthétisé ces dépendances au niveau des paquetages Java, pour analyser leurs dépendances. Nous avons construit automatiquement le modèle UML correspondant, sur lequel nous avons basé notre analyse. Nous avons construit ce modèle à l'aide d'un outil que nous avons développé, qui produit un modèle UML complet sous la forme d'un fichier au format XMI (XML Metadata Interchange) [RJB99], à partir de fichiers sources Java.

Nous présentons dans ce chapitre l'architecture que nous proposons, en la comparant avec celle de JOnAS, et en montrant comment cette architecture est plus flexible que celle de JOnAS. Nous nous appuyons également sur les principes de conception décrits dans la section 1.1, et sur les langages de traitement et d'ingénierie de RM-ODP.

La section 5.9 synthétise nos propositions pour une architecture de serveur EJB flexible.

5.1 Démarche globale

Globalement, le serveur JOnAS contient trois modules :

- le gestionnaire de transactions (JTM, Java Transaction Manager),
- le gestionnaire des connexions aux bases de données (DBM, Data Base Manager),
- le serveur EJB proprement dit (Jonas EJB).

Donc seulement deux services techniques sont spécifiés dans JOnAS : nous pouvons supposer que cela n'est pas suffisant pour considérer JOnAS comme modulaire. Nous proposons donc de raffiner la décomposition du conteneur en modules. En particulier, nous proposons de distinguer :

- des services techniques séparés et encapsulés séparément dans des modules,
- des types de canaux de communication, encapsulés séparément dans des modules,
- un serveur générique, indépendant de ces modules.

La gestion des dépendances étant primordiale dans une architecture (voir les principes de conception, section 1.1), nous souhaitons les maîtriser pour que des changements dans l'architecture de notre conteneur soient localisés, et aient un impact minimal.

Nous souhaitons donc concevoir un conteneur le plus générique possible, ayant des dépendances minimales vers les autres modules, en particulier vers les services techniques. L'architecture de conteneur que nous proposons sera la plus indépendante possible de ces implantations de canaux et de services techniques. Ce conteneur sera extensible pour accepter des configurations différentes de services et de canaux. La gestion des dépendances entre le conteneur et les canaux de communication est décrite dans la section 5.5.

En relation avec notre proposition d'étendre les spécifications EJB pour spécifier un modèle de représentation des services techniques (voir section 4.1), nous souhaitons avoir une flexibilité dans la composition des services techniques. Nous proposons donc de les séparer et de les encapsuler dans des modules distincts, afin qu'ils soient le moins possible dépendants les uns des autres.

De plus, nous souhaitons que ces modules de services techniques soient indépendants de leur contexte de mise en oeuvre, ici un serveur EJB. Les sections 5.2 et 5.7 décrivent les dépendances entre les services techniques, et les canaux et notre serveur EJB.

En ce qui concerne la composition des services techniques, nous souhaitons donc :

- pouvoir exprimer des abstractions de ces services, pour pouvoir changer l'implantation d'un même type de service, de manière transparente pour le reste de l'architecture,
- composer facilement un ensemble arbitraire de services techniques dans le conteneur générique,
- dans l'avenir, pouvoir spécifier le comportement de ces services techniques, et garantir qu'une composition de services techniques est structurellement et sémantiquement correcte, lors de la composition.

Pour décrire et justifier l'architecture globale que nous proposons dans ce chapitre, et atteindre les objectifs présentés ici, nous utilisons les principes de conception décrits dans la section 1.1.

Cependant, nous avons montré, dans le chapitre 3, que la mise en oeuvre d'un ensemble de services techniques dans les objets d'interposition pose des problèmes particuliers, parce que les services techniques sont mélangés dans ces objets. Nous envisageons donc d'utiliser une technique de séparation et de composition de problèmes pour construire ces objets d'interposition. Cette proposition est détaillée dans la section 5.3.

5.2 Objet d'interposition générique

Dans le conteneur JOnAS, tout le code des objets d'interposition est généré statiquement, avant l'exécution du conteneur, pour chaque bean déployé. Nous appelons ici *objet d'interposition* tout objet qui peut interagir avec un client ou bean. Ils comprennent donc les objets d'interposition dans le sens classique (voir fig. 1.7), que nous appelons ici objets d'interposition serveurs, les objets que manipule un bean pour accéder à des ressources, et les objets que manipule un client localement pour interagir à distance avec un bean, que nous appelons ici objets d'interposition clients.

Dans JOnAS, les objets d'interposition générés sont dépendants des beans, car ils implantent leurs interfaces fonctionnelles spécifiques, mais aussi des services techniques mis en oeuvre, car ils interagissent avec les interfaces de ces services, et aussi des canaux de communication. Par exemple, il est impossible d'ajouter un service technique dans ces objets d'interposition sans modifier manuellement le code des objets d'interposition générés.

De plus, le conteneur de JOnAS, et ses objets d'interposition en particulier, dépendent directement des types de canaux de communication (Java RMI ou la personnalité Jeremie de Jonathan), qui sont des modules de plus bas niveau. Les dépendances vers ces modules sont très fortes :

- le conteneur utilise directement les interfaces de ces types de canaux, qui sont spécifiques,
- les objets d'interposition du conteneur sont générés spécifiquement pour une couche de communication donnée, et il existe en fait une distribution de JOnAS spécifique à chaque couche de communication.

Cela implique les limitations suivantes :

- il n'est pas possible d'utiliser plusieurs types de canaux simultanément dans un même serveur JOnAS,
- il faut régénérer le code de tous les objets d'interposition lors du changement du type de canaux utilisé,
- la version 2.0 des spécifications EJB spécifie qu'un conteneur EJB doit supporter le standard CORBA : l'intégration de telles spécifications impose donc la création d'une nouvelle distribution de JOnAS pour supporter ce nouveau type de canaux de communications.

Pour résoudre ces problèmes, il faut appliquer le principe d'inversion des dépendances : il faut que le conteneur JOnAS ne dépende pas directement de ces modules de bas niveau, mais d'abstractions qui lui sont spécifiques.

Nous envisageons donc de construire un *conteneur générique*, indépendant des beans déployés, des types de canaux et des services techniques, et facilement extensible. En particulier, cela implique de minimiser le volume et la complexité du code généré lors du déploiement de beans.

En particulier, nous envisageons de pouvoir ne plus générer de code spécifique à chaque bean pour les objets d'interposition du conteneur. Pour rendre ces objets d'interposition génériques vis-à-vis des beans déployés, nous proposons de les placer à un *méta-niveau* par rapport aux interactions avec les beans. Ceci peut être réalisé en réifiant ces interactions dans des objets.

Dans le point de vue d'ingénierie de ODP, notre objet d'interposition devient donc un talon, lié à une ou plusieurs instances de la classe d'un bean, mais sans être spécifique à ces beans : c'est donc un talon générique. En tant que talon, il peut donc être directement intégré dans les canaux de communication, en liaison avec les lieurs. Ces canaux seront donc à un méta-niveau par rapport aux beans (objets d'ingénierie de base) qui interagissent.

Dans les EJB, un objet d'interposition met en oeuvre les services techniques pour les beans auxquels il est lié. En relation avec notre proposition d'encapsuler ces services dans des objets, un objet d'interposition interagira donc avec ces objets de service, à travers des interfaces stables et bien définies. Les objets d'interposition seront donc indépendants des implantations de ces services.

De même, nous souhaitons pouvoir utiliser plusieurs types de canaux différents (e.g. CORBA ou Java RMI) pour accéder aux beans, voire à un même bean. Nos talons serveurs génériques seront donc indépendants des canaux, et nous appliquons le principe d'inversion des dépendances pour rendre ces types de canaux dépendants d'interfaces abstraites et stables, spécifiques à notre serveur EJB. Ainsi, notre architecture sera "ouverte-fermée" à l'ajout de nouveaux types de canaux.

Notre talon serveur générique étant indépendant des canaux, il ne peut pas à lui seul assurer la transmission des interactions à distance. En effet, selon le langage d'ingénierie de RM-ODP, un talon a pour rôle de sérialiser (*marshall*) et désérialiser (*unmarshall*) les interactions, d'une manière spécifique au canal. Ceci doit donc être réalisé par le canal. Nous proposons donc que les talons serveurs soient en fait des composites, composés de deux objets :

- notre *talon générique*, qui manipule seulement des invocations réifiées dans des objets,
- un *talon (dé)sérialiseur* spécifique au canal, qui transforme ces objets en informations transmises dans le canal, et inversement.

Dans la suite de cette section, nous illustrons notre proposition dans le cas où les objets d'interposition traitent les interactions avec un session bean. Le même raisonnement peut être appliqué immédiatement aux entity beans, et avec des types d'interfaces différents aux message-driven beans.

D'après les spécifications EJB, toutes les interactions entre un client et un session bean se font à travers des objets spécifiques, qui implantent des interfaces spécifiées pour le bean, qui héritent elles-mêmes de l'interface `EJBObject`, `EJBLocalObject`, `EJBHome` ou `EJBLocalHome`. Dans notre architecture, ces objets n'auront pas d'autre fonction que d'interagir avec un talon générique client ou serveur pour lui déléguer leurs appels de méthodes.

Ainsi, notre talon serveur générique concentre toutes les invocations vers l'instance de session bean à laquelle il est lié, ainsi que la mise en oeuvre de tous les services techniques pour le bean. C'est donc ce talon serveur générique qui porte l'identité d'une session.

Les différents objets constituant un canal et un objet d'interposition serveur de notre conteneur sont illustrés dans la figure 5.1, pour un session bean ou un entity bean. Dans le cas d'un message-driven bean, les interfaces `EJBObject`, `EJBLocalObject`, `EJBHome` ou `EJBLocalHome` seraient remplacées par des interfaces spécifiques au service de file de messages JMS utilisé.

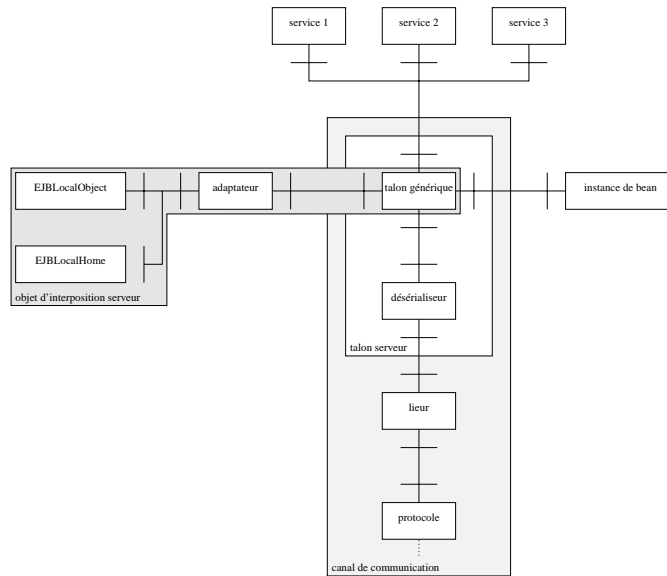


FIG. 5.1 – Architecture d’objet d’interposition serveur générique pour un session bean ou un entity bean

Nous envisageons de ne pas générer de code statiquement, avant l’exécution du conteneur, pour les objets implantant les `EJBObject`, `EJBHome`, `EJBLocalObject` et `EJBLocalHome` d’un bean lors de son déploiement, mais de le générer dynamiquement, pendant l’exécution du conteneur. En effet, toutes les fonctionnalités de ces objets étant mises en oeuvre dans nos talons génériques, le seul rôle de ces objets est de réifier les appels de méthodes, pour interagir avec les talons génériques.

Nous proposons de les réaliser aussi de manière générique, en utilisant par exemple les fonctionnalités du kit de développement 1.3 pour Java 2. En effet, il existe une classe standard `Proxy` qui réalise cette opération de manière portable et transparente. Nos `EJBObjects`, `EJBLocalObjects`, `EJBHomes` et `EJBLocalHomes` sont donc chacun constitués de deux objets : un *objet proxy* généré automatiquement par la classe `Proxy`, qui réifie les appels de méthodes de l’interface locale ou distante du bean, et qui interagit avec un objet qui adapte ces appels réifiés pour interagir avec le talon générique auquel il est lié. L’objet proxy est spécifique au bean correspondant, car il implante une de ses interfaces locales ou distantes, mais il est généré dynamiquement à l’exécution, ce qui évite une génération de code spécifique, et rend cette dépendance négligeable.

L’*objet adaptateur* est générique, car indépendant des interfaces des beans, et il est lié à un talon générique. Il peut être utilisé à la fois dans les objets d’interposition client et serveur. Pour pouvoir interagir avec les objets proxy, les objets adaptateurs doivent planter l’interface `InvocationHandler`. Un exemple de code pour générer un `EJBObject` pour un objet d’interposition client, avec la classe `Proxy` est :

```
// l’adaptateur qui interagit avec le talon serveur :
```

```

InvocationHandler adaptateur =
    new MandataireGenerique(talonServeur);

// la classe de l'interface distante du bean :
Class[] interfaces = {
    Class.forName("un.bean1.InterfaceDistante1")
};

// l'objet proxy est généré automatiquement pour
// réifier les appels de méthodes sur l'interface
// distante du bean :
Object proxy = Proxy.newProxyInstance(
    getClass().getClassLoader(),
    interfaces,
    adaptateur);

// l'EJBObject est directement l'objet proxy :
EJBObject ejbObject = (EJBObject)proxy;

```

Le diagramme de classes UML résultant pour une telle construction est donné dans la figure 5.2. Sur cette figure ne sont pas représentés les interfaces EJB-Home, ni les objets générés qui les implament.

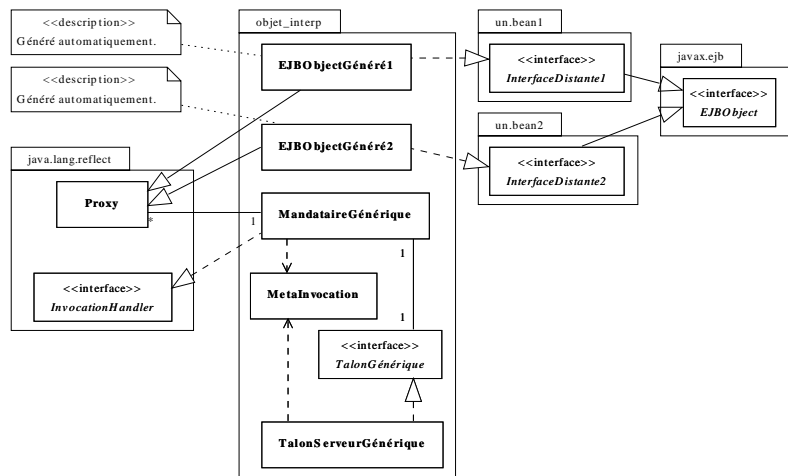


FIG. 5.2 – Diagramme de classes pour un EJBObject générique

Pour rester compatible avec des version de kits de développement Java antérieures à la version 1.3, nous pouvons aussi envisager l'utilisation d'autres outils que la classe Proxy pour générer les objets proxy. Par exemple, nous pouvons utiliser directement des bibliothèques de manipulation de bytecode Java, comme Javassist.

Pour certains protocoles de communication, comme Java RMI, il est nécessaire de générer des objets d'interposition clients, pour que les clients puissent interagir avec un session bean ou un entity bean à distance. Chaque accès à un bean par un client distant se fait à travers une des interfaces distantes du bean, qui hérite de `EJBObject` ou `EJBHome`.

L'architecture que nous avons proposée ci-dessus est utilisable à la fois côté serveur et côté client. Ainsi, l'architecture des objets d'interposition clients est similaire à celle des objets d'interposition serveurs. L'interface offerte aux clients distants peut être implantée automatiquement avec `Proxy`, de la même manière que du côté serveur. La seule différence tient au fait que le talon générique dans l'objet d'interposition n'est pas un talon serveur mais un talon client, qui fait toujours partie d'un canal pour transmettre les appels de méthodes réifiées vers le serveur. Le talon client est aussi un composite, qui comprend un talon client générique et un talon sérialiseur.

Le talon client générique est placé à un méta-niveau, et est donc indépendant des interfaces du bean. Son implantation est plus simple que celle des talons serveurs génériques, car il n'interagit directement avec aucun service technique (sauf pour la propagation implicite de contextes d'exécution, voir section 5.7). L'architecture résultante pour un objet d'interposition client est illustrée dans la figure 5.3.

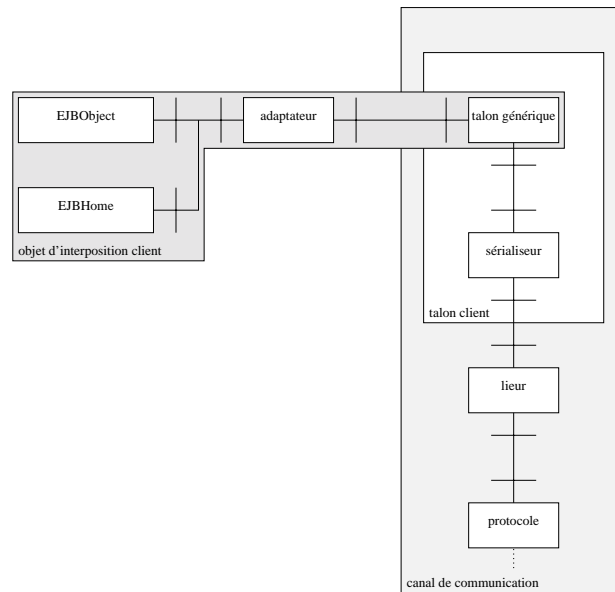


FIG. 5.3 – Architecture d'objet d'interposition client générique pour un session bean ou un entity bean

Lorsqu'un lien réparti est établi entre un objet d'interposition client et l'objet d'interposition serveur d'un bean, les objets de protocole client et serveur sont liés dans un canal, pour permettre la communication des interactions réifiées, entre les talons client et serveur.

Il est à noter que les objets d'interposition clients sont les seuls à pouvoir implanter les interfaces `EJBObject` et `EJBHome` des beans : les spécifications EJB spécifient en effet que les paramètres doivent être passés par valeur lorsqu'un client appelle une méthode de cette interface, ce qui n'est pas le cas pour les objets d'interposition serveurs qui n'interagissent pas par des canaux avec les beans.

A l'inverse, les objets d'interposition serveurs sont les seuls à pouvoir implanter les interfaces `EJBLocalObject` et `EJBLocalHome` des beans, car les paramètres sont ici passés par référence, ce qui n'est pas possible lorsque les interactions sont sérialisées dans un canal. Donc même lorsque deux beans communiquent localement, s'ils interagissent à travers une interface `EJBObject` et `EJBHome` des canaux doivent être mis en place. Si ces beans utilisent les interfaces `EJBLocalHome` et `EJBLocalObject`, ils ne passent naturellement pas par des canaux, dans notre architecture : ils interagissent à travers l'objet d'interposition serveur seulement, et leurs interactions sont bien optimisées.

La génération d'objets d'interposition clients n'est pas nécessaires pour CORBA, par exemple. Nous n'avons donc pas besoin de nous préoccuper de la partie client dans ce cas : les outils de déploiement du support d'exécution du client s'en chargent. De même, il n'est pas nécessaire de fournir de tels objets pour les message-driven beans : les clients y accèdent en utilisant les APIs de JMS (*Java Message Service*).

L'avantage d'étendre le modèle de bean avec la composition, comme nous l'avons proposé (voir section 4.4), est de permettre de ne pas construire de canaux de communication distante complexes, dans lesquels seraient placés les talons serveurs génériques d'un bean, lorsque tous les beans dans le même composite que ce bean sont déployés dans le même serveur (i.e. lorsque ce bean n'interagit pas avec des beans distants). Ainsi, les talons serveurs pour ce bean seraient accessibles seulement localement à travers les interfaces locales `EJBLocalObject` et `EJBLocalHome`, où à travers des canaux optimisés spécifiques à notre conteneur, pour les interfaces `EJBObject` et `EJBHome`. Cela permet de renforcer les règles de sécurité pour les accès à distance, et de minimiser les ressources réseau et mémoire pour la construction des canaux. De plus, nous envisageons de permettre une reconfiguration dynamique des beans composites, pour permettre dynamiquement l'accès distant ou non à un bean, et donc la construction ou non de canaux complets pour ses talons serveurs. Ceci s'intégrerait facilement avec l'architecture que nous proposons.

Nous pouvons envisager d'utiliser l'architecture présentée dans cette section pour construire les objets d'interposition qui traitent les interactions entre les instances de beans et les ressources du serveur EJB (e.g. des connexions à un serveur de données, etc.). En effet, la seule différence serait que les objets proxy implanteraient les interfaces des standards Java pour ces ressources (e.g. `java.transaction.UserTransaciton`, etc.), et que les talons génériques seraient remplacés par des objets génériques, construits de la même manière que les talons génériques, et dont la seule fonctionnalité serait de mettre en oeuvre des services techniques. Ces objets génériques ne seraient pas des talons, et donc n'interagiraient pas avec des canaux. Nous ne traitons pas ce problème spécifiquement ici.

5.3 Utilisation de l’AOP pour le talon serveur générique

Nous avons proposé (voir section 5.1), de spécifier des services techniques indépendamment de notre serveur EJB, mais aussi de concevoir un serveur indépendant des services techniques qu’il met en oeuvre : il y a donc une contradiction. Pour résoudre cette contradiction, nous proposons d’introduire une dépendance du conteneur vers les services techniques, mais en minimisant cette dépendance pour la maîtriser.

Les points stratégiques d’un serveur sont les objets d’interposition, et dans notre cas les talons serveur génériques, car ce sont ces objets qui mettent en oeuvre les services techniques. Nous proposons donc de limiter les dépendances du serveur à des dépendances de ces talons serveur génériques vers les services techniques.

Pour que les dépendances entre les talons serveur génériques et les services techniques soient minimales, nous proposons de ne pas les matérialiser dans le code source des talons génériques, voire dans leur code compilé. Idéalement, il faut donc que la configuration des services techniques dans ces talons soit réalisée pendant l’exécution du serveur. Ainsi, le code des talons serveur génériques, et plus généralement du serveur, sera ouvert-fermé à la mise en oeuvre de nouveaux services.

Nous avons montré aussi (voir chapitre 3) que les services techniques sont mélangés dans ces objets d’interposition, et que la séparation de ces services dans des modules distincts avec des techniques classiques, dans les objets d’interposition, n’est pas appropriée. Nous proposons donc d’utiliser une technique de séparation et de composition de problèmes, parmi celles que nous avons présentées dans le chapitre 2.

Nous envisageons d’utiliser l’outil JAC (Java Aspectual Components) [PSDF01], qui met en oeuvre la programmation par aspects par réflexion à l’exécution (voir section 2.2). Cet outil nous permettra de composer tous les services techniques dans les talons serveur génériques, pendant l’exécution. Cet outil est bien adapté au contexte des EJB, car cet outil propose des points de jonction qui sont :

- le début d’un appel de méthode,
- la fin d’un appel de méthode,
- la levée d’une exception.

Ces points de jonction correspondent exactement aux points où les services techniques sont mis en oeuvre dans les objets d’interposition, d’après les spécifications EJB. JAC correspond donc tout à fait à nos besoins.

Notre talon serveur générique est un point stratégique de notre serveur EJB car :

- il est stable : sa seule fonctionnalité de base est d’appeler des méthodes sur les instances de beans, et cette fonctionnalité de base n’évolue jamais. De plus, il est indépendant des beans dont il appelle les instances. Ainsi, son code n’est pas susceptible d’être modifié.
- il est abstrait : sa fonctionnalité de base est indépendante des services techniques mis en oeuvre, des beans et des modes d’accès distants, et il se situe à un méta-niveau, donc à un plus haut niveau d’abstraction.

Avec ces caractéristiques, notre proposition applique le principe de stabilité des abstractions. Mais nous devons aussi appliquer les principes d'inversion des dépendances et de relation dépendance-stabilité : notre talon serveur générique ne doit avoir aucune dépendance vers d'autres modules. En revanche, d'autres modules peuvent dépendre de lui. Notre proposition respecte ces deux principes. En effet, les services techniques seront spécifiés par des aspects spécifiques devant être intégrés avec l'aspect de base de l'objet d'interposition, et les autres aspects. Ces aspects techniques seront donc dépendants de la technique de composition utilisée, et de la structure de l'objet d'interposition. Mais l'aspect de base ne dépend pas des autres aspects.

Cependant, nous souhaitons aussi que les services techniques soient utilisables dans d'autres contextes que notre serveur EJB, de manière transparente (voir section 5.1). Il faut donc que ces services ne soient pas dépendants des interfaces spécifiques à notre serveur, et notamment de celles de l'objet d'interposition. Nous proposons donc de séparer chaque service en deux modules distincts :

- un *module fonctionnel*, qui définit des interfaces qui sont spécifiques au service, et implante tout le code fonctionnel du service. Tout module utilisant le service doit utiliser – et donc dépend – des interfaces définies dans ce module. Ainsi, comme ce module ne dépend pas d'interfaces spécifiques à notre serveur EJB, il peut être réutilisé dans d'autres contextes. Ce module correspond à notre proposition présentée dans la section 5.1.
- un *module d'aspect*, qui implante l'aspect devant être mélangé avec les autres aspects dans un talon serveur générique. Ce module n'implante pas de code du service, mais a seulement un rôle d'adaptateur. Il dépend à la fois des interfaces spécifiques à l'objet d'interposition, de l'outil de composition (JAC), et des interfaces du module fonctionnel du service technique. Son rôle est d'interagir avec ces interfaces pour mettre en oeuvre le service dans le contexte du talon serveur générique. Le code dans ce module est donc réduit au minimum.

La figure 5.4 illustre l'architecture découlant de cette décomposition.

La proposition décrite dans cette section nous permet donc de construire un objet d'interposition serveur générique “ouvert-fermé” à l'ajout de nouveaux services, mais aussi de rendre les services indépendants du contexte spécifique de notre serveur EJB.

Dans un premier temps, nous ne souhaitons pas pouvoir changer dynamiquement les types de services mis en oeuvre par un objet d'interposition donné, bien que nous envisagions de pouvoir changer dynamiquement l'implantation d'un type de service donné. En effet, il y aurait un risque d'incohérence, et de rupture de contrat entre le bean interposé et les clients qui interagissent avec lui. L'utilisation de l'AOP au chargement des classes, avec JAC, correspond donc à nos attentes. Pour modifier l'ensemble des types de services d'un objet d'interposition, nous proposons de désinstaller et réinstaller le bean concerné, afin de régénérer ses objets d'interposition pour les adapter au nouvel ensemble de types de services.

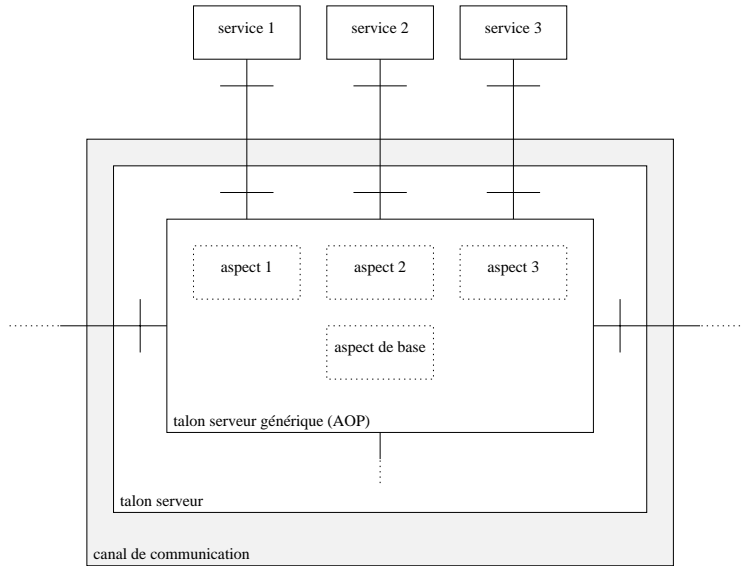


FIG. 5.4 – Architecture pour la composition des services techniques dans un talon serveur générique

5.4 Gestion de l'état des sessions et des entités

Les spécifications EJB 2.0 spécifient des diagrammes d'état-transitions pour les sessions (avec état et sans état), et pour les entités. Ces diagrammes d'état correspondent au cycle de vie des sessions et entités, vues par les clients. De plus, les EJB spécifient des diagrammes d'état-transitions pour les instances de beans, qui exécutent les opérations pour les sessions, entités et message-driven beans. Les états des instances sont gérés différemment des états des sessions et entités, car par exemple une même entité peut ne correspondre à aucune instance à un moment donné, si aucun client n'interagit avec elle, ou à plusieurs instances, pour augmenter le parallélisme des accès de plusieurs clients qui interagissent avec cette entité, car les instances de beans ne sont en général pas réentrantes.

L'état des instances et des sessions et entités comporte, en plus d'informations sur leur cycle de vie et leur identité (e.g. la clé primaire d'une entité), des informations spécifiques aux services techniques mis en oeuvre. Par exemple, l'état d'une session peut comporter un verrou, spécifique à un service de gestion de concurrence, pour éviter que plusieurs clients invoquent une même session simultanément.

Nous proposons donc de réifier ces états dans des objets, auxquels peuvent être attachés des objets d'état spécifiques aux services techniques. Nous avons proposé dans la section 5.2, que ce soient les talons serveurs génériques qui portent l'identité des sessions et des entités. Ils gèrent donc l'état des sessions et des entités. En revanche, les objets d'état ne font pas partie de l'état des talons serveurs génériques, car un même objet d'état peut être partagé par plusieurs talons. Un objet d'état est donc lié (au sens donné dans RM-ODP) à un ou

plusieurs talons serveurs génériques, et un talon pour une session ou une entité est lié à au plus un seul objet d'état.

Notre proposition de réification des informations d'état spécifiques aux services techniques met en oeuvre le patron de conception MÉMENTO [GHJV94]. En effet, un objet réifiant une telle information d'état est spécifique au service qui l'a créé, et ne peut être utilisé que par lui. Cet objet (memento) encapsule une partie de l'état d'un service technique, mais en conservant le principe d'encapsulation pour ce service.

Pour simplifier notre architecture, et le code de nos talons serveurs génériques, nous proposons qu'un talon serveur générique ne puisse être lié qu'à au plus une instance de bean à un moment donné. De plus, nous proposons de ne lier un talon générique à une instance que lorsque celle-ci est dans un état prêt pour exécuter des méthodes métier correspondantes à des interactions initiées par les clients. Cela simplifie l'architecture des talons génériques, car ils ne doivent pas gérer l'état des instances de beans. De plus, cela nous permet de ne pas réifier l'état des instances de beans dans des objets liés aux talons serveurs. Nous proposons de déléguer cette gestion de l'état des instances de beans à des services spécifiques, par exemple les services de gestion des ressources, de gestion de l'activation / passivation des instances, ou de gestion des erreurs (e.g. une instance est désactivée si elle provoque une erreur grave). Ces services sont incorporés dans le talon serveur générique, et peuvent par exemple créer ou activer une instance et la lier au talon, lorsque le talon doit gérer une interaction avec un client, si le talon n'est lié à aucune instance à ce moment. Une instance peut être détachée d'un talon serveur générique, par exemple si ce talon ne gère aucune interaction avec un client à ce moment, et si ce talon n'a pas été actif depuis un délai donné.

Plusieurs talons serveurs peuvent être liés à une même instance, par exemple pour qu'un client puisse accéder à une même session avec état (stateful session), par des canaux différents. En effet, une session avec état ne peut correspondre qu'à une seule instance à un moment donné, d'après les spécifications EJB. Dans ce cas, les talons serveurs sont liés à la même instance et au même objet d'état de session, qui comprendra par exemple un verrou spécifique à un service de gestion de concurrence, pour empêcher plusieurs interactions simultanées avec la session, conformément aux spécifications EJB.

Il est possible qu'un talon serveur générique soit lié à une instance de bean sans être lié à un objet d'état, si ce bean est un message-driven bean ou si ce talon est utilisé pour exécuter des méthodes de l'interface `EJBHome` ou `EJBLocalHome` d'un session bean ou d'un entity bean. En effet, l'exécution de telles méthodes peut se traduire par un appel de méthode métier sur une instance du bean qui n'a aucune identité pendant cet appel, donc aucun état ne doit être géré pendant cet appel de méthode métier.

La figure 5.5 illustre le diagramme de classes de l'architecture que nous proposons pour gérer les états de sessions et d'entités.

Avec notre proposition, tout objet d'interposition d'un bean peut donc traiter des appels sur les interfaces `EJBHome` et `EJBLocalHome` du bean correspondant, alors que dans JOnAS ces appels sont gérés par des objets séparés (héritant de la classe `JBeanHome` de JOnAS). Ainsi, notre proposition permet de gérer tous

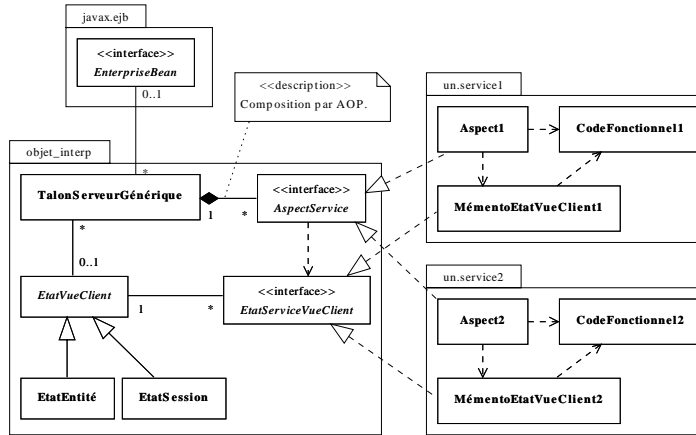


FIG. 5.5 – Architecture pour la gestion de l'état des sessions et des entités

les appels sur toutes les interfaces des beans de la même façon. L'architecture que nous proposons est donc plus générique que celle de JOnAS. De plus, nous envisageons de pouvoir exécuter les appels des méthodes des `EJBHomes` et `EJBLocalHomes` d'un même bean par plusieurs objets d'interposition simultanément, de manière configurable, pour permettre un plus grand parallélisme des ces appels.

5.5 Canaux de communication ouverts

De même que pour les services techniques, nous devons appliquer les principes d'inversion des dépendances et de relation dépendance-stabilité aux dépendances entre nos canaux de communication et nos objets d'interposition génériques.

Les standards tels que JMS, Java RMI et CORBA spécifient des APIs et des modes de programmation devant être respectés par les objets communicants. Ceci viole le principe d'inversion des dépendances : ces objets ne peuvent pas s'adapter de manière transparente à des canaux de nature différente. D'après le principe de relation dépendance-stabilité, ceci ne pose pas de problème dans le cas général, où la nature des objets communicants est plus variable que la nature des canaux. Mais dans notre cas, nos objets d'interposition sont plus stables et plus abstraits que les canaux de communication.

Pour pouvoir inverser ces dépendances dans les canaux de communications, nous proposons d'utiliser un *canevas ouvert* pour la construction de ces canaux. Nous proposons notamment d'utiliser l'outil Jonathan [DHTS98], qui est déjà utilisé dans le serveur JOnAS. L'avantage d'utiliser un canevas ouvert est la possibilité d'adapter les canaux à nos besoins, et notamment d'y inclure nos talons génériques clients et serveur, comme nous le proposons (voir section 5.2).

De plus, nos objets d'interposition génériques doivent gérer les `Handles` et `HomeHandles` des beans, c'est-à-dire les références d'interfaces d'ingénierie des beans. Ces objets servent à construire des canaux pour interagir avec les beans. D'après

le langage d'ingénierie de RM-ODP, cette fonctionnalité est implantée par les objets lieurs dans les canaux. Nous envisageons donc naturellement d'implanter cette fonctionnalité dans les lieurs. Ceci impose d'avoir accès à ces lieurs, et donc d'avoir une architecture ouverte pour les canaux, comme nous le proposons.

La possibilité d'avoir accès aux lieurs dans les canaux nous permettra aussi de mettre en oeuvre plus facilement des services techniques comme la duplication ou la migration de beans, qui nécessitent de manipuler les liens distants.

La figure 5.6 illustre un exemple d'architecture de canal et les dépendances entre ce canal et l'objet d'interposition. Nous pouvons observer sur cette figure que notre architecture respecte bien le principe de relation dépendance-stabilité : le canal dépend de notre objet d'interposition, qui est plus stable et plus abstrait, qui a son tour dépend des APIs standard des EJB, qui sont plus stables et plus abstraites.

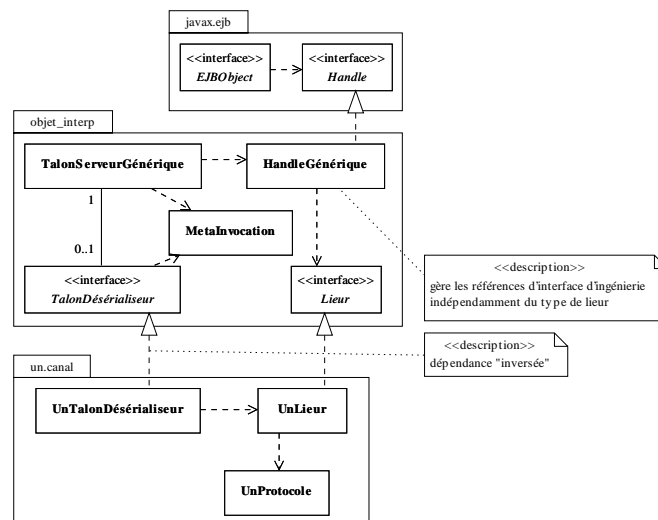


FIG. 5.6 – Inversion des dépendances entre canaux et objets d'interpositions

Cette proposition d'ouverture des canaux nous amènera probablement à modifier Jonathan, en l'étendant pour dépendre des interfaces spécifiques à notre objet d'interposition. Nous prévoyons que ces extensions seront minimales, car nous avons spécifié explicitement nos objet d'interposition comme des talons, et leur intégration en sera facilitée.

5.6 Interactions avec les talons génériques

Nous souhaitons avoir un mode d'interaction homogène entre tous les objets d'interposition et les canaux, et avoir des interactions de même type et de même niveau d'abstraction entre les talons génériques clients et serveurs et les objets auxquels ils sont liés, c'est-à-dire les talons (dé)sérialiseurs dans les canaux, et les objets adaptateurs dans les objets d'interposition.

Par exemple, JMS impose des interactions de type signal, alors que Java RMI et CORBA imposent des interactions de type interrogation (de type opération). Mais les interactions de type interrogation peuvent être toujours décomposées en deux interactions unidirectionnelles (une invocation et une terminaison), comme cela est décrit dans le langage de traitement de RM-ODP (voir section 1.2.1.3). Les signaux JMS et les invocations composant les opérations Java RMI, par exemple, ont la même sémantique : ils provoquent l'appel d'une méthode métier sur une instance de bean, et la mise en oeuvre de services techniques par le conteneur EJB. Ainsi, ces interactions sont au même niveau d'abstraction, et sont de même type : nous proposons donc de traiter ces interactions de la même façon.

Comme tout notre conteneur, et donc notre talon générique, est programmé en Java, et que le modèle d'objet de Java ne supporte que des interactions de type opération, nous proposons de faire correspondre chaque interaction de base, entre un talon générique et un talon (dé)sérialiseur ou un objet adaptateur, à une opération spécifique sur une interface Java. Ainsi, par exemple, un envoi de message JMS à un message-driven bean se traduira par l'appel d'une méthode Java sur le talon serveur générique du bean, par le talon désérialiseur d'un canal JMS. De même, un appel de méthode métier d'une session ou d'une entité par un client, à travers un canal CORBA ou Java RMI, se traduira par un appel d'une méthode du talon serveur générique par le talon désérialiseur (interaction d'invocation), suivi par un appel d'une méthode du talon désérialiseur par le talon générique (interaction de terminaison).

Ainsi, l'architecture de notre talon générique est basée seulement sur des interactions de type opération au niveau du langage Java, et est adaptable à tout type d'interaction à un niveau d'abstraction supérieur. La programmation des modules d'aspect des services techniques sera aussi facilitée, car ils seront utilisables pour tous types de beans, quelque soit le type de leurs interactions.

Nous appliquerons cette architecture du côté client, pour les interactions entre les talons clients génériques des sessions et entités, et leurs canaux. La seule différence est que les causalités des interfaces des talons génériques et des canaux seront inversées pour leurs interactions, par rapport au côté serveur.

Il est à noter que les talons génériques client et serveur possèdent chacun deux interfaces, pour interagir selon notre proposition :

- une interface “locale”, pour interagir avec l'objet adaptateur auquel il est lié dans un objet d'interposition,
- une interface “de canal”, pour interagir avec le talon (dé)sérialiseur auquel il est lié dans le canal.

Ces deux interfaces participent à des interactions de même nature, correspondant à notre proposition ci-dessus. Mais elles doivent être séparées, pour que l'interaction de terminaison terminant une interaction de type opération sur un bean, se passe bien avec le même objet (objet adaptateur ou (dé)sérialiseur) que lors de l'interaction d'invocation de cette opération. Les interfaces “locales” ont toujours une causalité “de répondeur” aux interactions, alors que les interfaces “de canal” ont une causalité “de répondeur” côté serveur et “d'initiateur” côté client.

La figure 5.7 illustre ces interfaces, et les interactions au plus bas niveau, pour des objets d'interposition client et serveur qui gèrent des interactions de type interrogation, pour des session beans ou des entity beans. Nous pouvons observer que ces opérations sont décomposées chacune en une invocation et une terminaison.

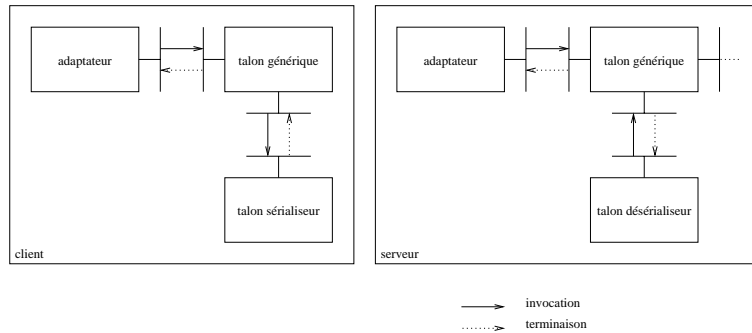


FIG. 5.7 – Interactions dans les objets d'interposition pour l'exécution d'une interrogation

5.7 Propagation de contextes d'exécution

Un des problèmes que nous avons observés dans le serveur JOnAS est qu'il propage implicitement un contexte de transaction lors des interactions à l'intérieur du conteneur. Cette propagation implicite de contexte consiste en l'attachement d'informations (ici, des identifiants de transactions) aux flux d'exécution (*threads*).

Les spécifications EJB imposent que les contextes de transaction et de sécurité soient propagés implicitement lors des interactions distantes entre les beans. Et cette propagation de contexte se fait aussi de manière implicite lorsqu'une instance de bean est invoquée : ceci permet aux beans d'accéder aux ressources (e.g. bases de données) dans un contexte cohérent et de manière transparente. L'avantage est que cela simplifie le modèle de programmation des beans, et rend transparent l'ajout de services pour les beans existants.

Mais les spécifications EJB ne spécifient pas l'architecture d'un conteneur, et donc n'imposent pas aux conteneurs EJB de propager ces contextes implicitement lors des interactions qui n'impliquent pas les beans, à l'intérieur du conteneur, comme cela est fait dans JOnAS. En effet, dans JOnAS des informations (e.g. l'identifiant de la transaction en cours) sont associées au contexte des flux d'exécution. Cela impose que tous les traitements concernant une interaction avec un bean se fassent dans un même flux d'exécution, dans le conteneur JOnAS. Le gestionnaire de transactions de JOnAS est donc lié au modèle d'activité de Java, et empêche d'utiliser d'autres modèles d'activité, comme la programmation réactive, pour implanter le conteneur : ceci limite donc la flexibilité de l'architecture de ce conteneur.

De plus, cette propagation implicite de contexte induit un couplage fort entre les modules du conteneur (ici, surtout le gestionnaire de transactions), et ce couplage n'est pas visible explicitement dans l'architecture. De plus, ceci ne respecte pas les spécifications du langage de traitement de RM-ODP, qui spécifie que toutes les interactions, donc toutes les communications, doivent se faire entre les interfaces des objets, alors que la propagation implicite est une forme de communication qui ne se fait pas entre des interfaces. La figure 5.8 illustre de telles interactions :

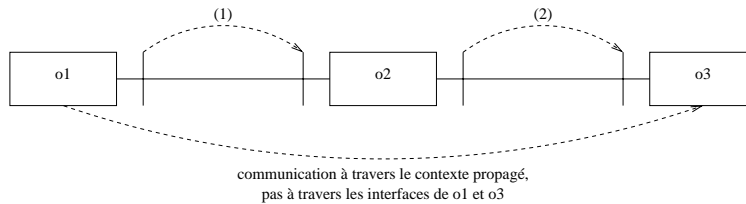


FIG. 5.8 – Communication par propagation implicite de contexte d'exécution

- o1 attache un contexte d'exécution à une activité (*thread*).
- (1) o1 appelle une méthode sur l'interface de o2 dans cette activité, donc le contexte d'exécution est propagé par cet appel. o2 ne peut pas manipuler le contexte d'exécution, et n'est pas conscient de ce contexte.
- (2) o2 appelle une méthode sur l'interface de o3 dans la même activité, donc le contexte est à nouveau propagé de manière transparente.
- o3 utilise le contexte propagé depuis o1, dans la méthode appelée.

Donc cette séquence d'interactions propage des informations entre les objets o1 et o3, alors qu'ils ne sont pas liés par leurs interfaces.

Pour résoudre ce problème, nous proposons de ne faire une propagation implicite de contexte que lorsque les spécifications EJB l'imposent. Pour les interactions internes au conteneur, nous souhaitons profiter du fait que notre objet d'interposition (et donc notre conteneur) est placé à un méta-niveau, pour réifier les contextes d'exécution et les lier aux objets réifiant les invocations de beans. Comme les aspects des services techniques sont aussi au méta-niveau, ils peuvent manipuler naturellement les informations de contexte qui les concernent lorsque les invocations sont traitées par le talon serveur. Cette proposition nous permet donc de transformer une propagation implicite de contexte en propagation explicite, en conservant les mêmes fonctionnalités. De plus, ce mécanisme est générique et peut s'adapter à tout service nécessitant une propagation de contexte.

Notre proposition de réification des contextes d'exécution est une mise en oeuvre du patron de conception MÉMENTO [GHJV94]. En effet, un objet réifiant une information de contexte d'exécution est spécifique au service qui l'a créé, et ne peut être utilisé que par lui. Cet objet (le memento) encapsule une partie de l'état d'un service technique, mais en conservant le principe d'encapsulation pour ce service. L'utilisation de ce patron de conception nous permettrait aussi d'ajouter des services, comme par exemple la persistance des contextes pour retrouver le contexte d'appel après une panne du serveur.

Nous pouvons considérer que la propagation implicite de contextes d'exécution, lors des invocations des beans, est un aspect technique à part entière. En effet, les mécanismes mis en place pour propager implicitement un contexte sont identiques pour tous les services techniques qui propagent des contextes. Nous pouvons donc considérer que c'est un aspect dont le code est éparpillé dans plusieurs autres services techniques. Pour séparer ce service de propagation implicite des autres services, nous proposons d'attacher aux flux d'exécution (*threads*) les objets réifiant les invocations, auxquels sont attachés les objets memento réifiant les contextes, avant tout appel sur une instance de bean, et de récupérer ces objets dès que l'appel est fini ou que l'instance de bean interagit avec le conteneur. Ainsi, nos services techniques n'ont plus à propager eux-même leurs contextes d'exécution, et se limitent à la manipulation des objets mémentos qui les concernent. Cette proposition nous permet donc de séparer simplement, dans notre conteneur, l'aspect de propagation implicite des contextes d'exécution.

Un autre problème est qu'il y a un couplage fort entre l'implantation des services et le code pour sérialiser et désérialiser (*marshall / unmarshall*) leurs informations de contexte dans les canaux de communication. En effet, les informations dans les objets mémentos sont une partie de l'état des modules fonctionnels des services, donc y sont fortement couplés, et ces informations peuvent être transmises de manière spécifique pour un type de canal particulier. Par exemple, la propagation implicite de contextes de transaction est standardisée dans CORBA, avec le service CORBA Object Transaction Service. Un service de gestion de transaction doit donc s'adapter à ces spécifications lorsqu'il est utilisé avec des canaux CORBA. En revanche, dans Java RMI il est impossible de propager implicitement un contexte d'exécution, et nous sommes obligés de transformer les propagations implicites en propagations explicites, comme cela est réalisé dans JOnAS. Donc les différents types de canaux imposent des modèles et des modes de programmation différents pour certains services techniques. Il y a donc une dépendance des services techniques vers les canaux de communication.

Pour résoudre ce problème de couplage, nous proposons d'appliquer le principe d'inversion des dépendances [Mar96a]. En effet, nous avons choisi (voir section 5.1) d'encapsuler séparément les services techniques dans des modules fonctionnels indépendants de notre conteneur, et donc des canaux. Ils ne doivent donc pas dépendre de ces canaux, mais l'inverse, et nous devons conserver pour chaque service un seul module fonctionnel qui contient le code fonctionnel de ce service (voir section 5.2). Nous proposons donc d'introduire des modules supplémentaires spécifiques à chaque service, pour (dé)sérialiser les contextes d'exécution, à l'intérieur des canaux de communication. Ainsi, chaque module de (dé)sérialisation de contexte est spécifique à la fois au canal dans lequel il est placé, pour respecter son modèle de propagation de contexte, et au service technique avec lequel il interagit, car il manipule les objets mémentos qui encapsulent ces contextes et qui sont spécifiques seulement au module fonctionnel du service. Ce module de (dé)sérialisation a donc des dépendances fortes vis-à-vis de son environnement, mais cela n'est pas important car tout le code fonctionnel des services se trouve dans leurs modules fonctionnels. A priori, la (dé)sérialisation des contextes se fait en même temps que la (dé)sérialisation de l'invocation correspondante. Nous proposons donc de rattacher ces modules à l'objet du talon qui (dé)sérialise les interactions dans le canal.

Pour pouvoir introduire des modules de (dé)sérialisation quelconques dans les canaux de communication, il est nécessaire d’avoir une architecture ouverte pour ces canaux, comme nous le proposons dans la section 5.5. L’avantage de notre proposition est que nos canaux de communication, et notre conteneur globalement, sont “ouverts-fermés” à l’ajout de nouveaux services techniques qui utilisent la propagation implicite de contextes d’exécution.

La figure 5.9 illustre l’architecture correspondant à notre proposition, pour la partie serveur d’un canal (dé)sérialisation de contextes). Nous pouvons observer, sur cette figure, que les modules fonctionnels sont bien indépendants du conteneur et des canaux de communication, et que leur adaptation se fait par des modules supplémentaires (e.g. Désérialiseur1, MementoContexte1). Sur cette figure ne sont représentés que les modules de la partie serveur.

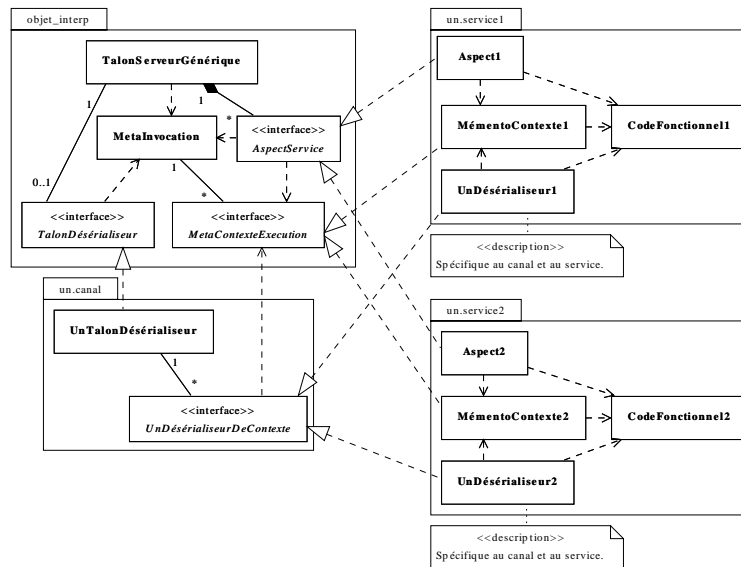


FIG. 5.9 – Proposition d’architecture pour la désérialisation de contextes d’exécution

Lorsque des objets d’interposition clients doivent être fournis (e.g. pour Java RMI), la même architecture doit être appliquée pour la sérialisation des contextes d’exécution. La seule différence est que dans le cas du serveur, seuls des désérialiseurs de contextes sont utilisés, alors que côté client ce sont des sérialiseurs. Un service technique qui propage des contextes doit donc fournir toujours un désérialiseur pour chaque type de canal utilisé, et un sérialiseur seulement si des objets d’interposition clients doivent être fournis pour ce type de canal.

Le problème de la transmission des informations de contexte d’exécution dans les canaux est donc résolu. Mais cela ne résoud pas le problème de la réification des contextes d’exécution, afin de les transmettre à distance, lorsqu’un programme client appelle une méthode d’un objet d’interposition d’un bean, sur une de ses interfaces implantant EJBHome, EJBLocalHome, EJBObject ou EJBLocalObject. Nous distinguons deux cas généraux :

1. l'objet d'interposition est un objet d'interposition serveur, c'est-à-dire que son talon générique est un talon serveur générique. Dans ce cas, l'interaction est une interaction locale entre deux beans dans le même serveur. Cela signifie que l'appel se déroule dans un contexte d'exécution déjà géré par notre serveur. Donc, en nous appuyant sur notre proposition ci-dessus, il suffit de récupérer l'objet réifiant l'invocation qui est à l'origine de cet appel, propagé implicitement, et de récupérer les objets memento de contexte d'exécution qui lui sont liés. Ce mécanisme est mis en oeuvre par l'objet adaptateur dans l'objet d'interposition, avant d'interagir avec le talon générique. Aucun code spécifique aux services n'est nécessaire ici : notre mécanisme est générique.
2. l'objet d'interposition est un objet d'interposition client, c'est-à-dire que son talon générique est un talon client générique. Cela signifie que le client peut être dans un autre serveur EJB que le bean avec lequel il interagit à distance. Nous pouvons distinguer quatre cas ici :
 - (a) le programme client s'exécute dans une autre instance de notre serveur, ou dans la même instance de serveur. Dans ce cas, nous utilisons le même mécanisme que pour un objet d'interposition serveur : l'objet réifiant l'invocation de bean est manipulé pour récupérer les objets memento de contexte. De la même manière que côté serveur, ce mécanisme est mis en oeuvre par l'objet adaptateur dans l'objet d'interposition, de manière générique.
 - (b) le programme client ne s'exécute pas dans une instance de notre serveur, mais le type de canal utilisé offre une sérialisation automatique des contextes d'exécution, dans le talon sérialiseur. Dans ce cas, l'objet d'interposition client et les services techniques n'ont pas à gérer les contextes d'exécution. Ce cas est le plus simple à implanter.
 - (c) le programme client ne s'exécute pas dans une instance de notre serveur, et le type de canal ne transmet pas automatiquement le contexte d'exécution (e.g. dans Java RMI). Mais la spécification du type du canal, dans lequel est placé le talon générique client, spécifie un mécanisme (e.g. des interfaces standardisées) pour récupérer des informations de contexte d'exécution qui peuvent être transmises directement à distance. Ainsi, l'objet adaptateur pourrait utiliser ce mécanisme pour récupérer les informations de contexte, les adapter, les encapsuler dans des objets memento, et les transmettre au talon client générique avec l'invocation réifiée. Nous n'avons aucun exemple de telles spécifications.
 - (d) le programme client ne s'exécute pas dans une instance de notre serveur, le type de canal ne transmet pas automatiquement le contexte d'exécution, et il n'y a aucun moyen de récupérer des informations de contexte d'exécution qui puissent être directement transmises dans le canal. Il n'y a pas de moyen simple de résoudre ce problème.

La réification des contextes d'exécution dans nos objets d'interposition se fait toujours dans les objets adaptateurs. Cette réification est générique et triviale, dans le cas où cet objet d'interposition se trouve dans une instance de notre serveur EJB.

Mais dans le cas 2.(c) ci-dessus, il serait nécessaire que les services techniques fournissent un module permettant de réifier les informations de contexte spécifiques à ces services, à partir du mécanisme fourni par le support d'exécution du client, en plus du module de sérialisation de ces informations attaché au sérialiseur dans le canal. Ces modules de réification seraient donc spécifiques à la fois au support d'exécution du client, au service technique et au type du canal. Ce cas peut être difficile à gérer et à maintenir : nous ne le traitons pas ici.

Par contre, il n'y a pas de problème pour les types de canaux qui ne nécessitent pas que des outils de notre serveur génèrent des objets d'interposition clients (e.g. CORBA). Dans ce cas, c'est le support d'exécution du programme client qui est en charge de générer des objets d'interposition clients appropriés pour manipuler les contextes d'exécution et les propager à distance vers notre serveur.

Pour simplifier notre proposition, nous proposons donc de ne pas supporter les cas 2.(c) et 2.(d) évoqués ci-dessus, et ne supporter que trois types de support d'exécution des programmes clients :

- des clients dont le support d'exécution est le serveur que nous décrivons dans ce document (cas 2.(a) ci-dessus),
- des clients dont le support d'exécution et les types de canaux utilisés n'imposent pas que des objets d'interposition clients soient générés par les outils de notre serveur,
- des clients qui utilisent des types de canaux qui transmettent automatiquement le contexte d'exécution, mais qui nécessitent quand même que des objets d'interposition clients soient générés (cas 2.(b) ci-dessus).

Dans aucun de ces cas les services techniques n'ont besoin d'interagir avec les talons clients génériques. Nous n'avons donc pas besoin de composer des aspects de services techniques dans ces talons clients génériques, comme pour les talons serveurs génériques.

Il reste le problème de l'interopérabilité entre des services de même type, par exemple entre des gestionnaires de transactions, lors de la propagation de leurs contextes à distance. Certaines spécifications de canaux, comme CORBA, définissent comment certains services peuvent interopérer. Dans ce cas, les modules de (dé)sérialisation des services se conforment à ces spécifications et sont naturellement interopérables. Mais pour les types de canaux qui ne spécifient pas comment les services interopèrent, par exemple dans Java RMI, il faut que nous spécifions nous-même comment les services doivent interopérer. Pour chaque type de canal, nous proposons donc de fournir des *spécifications d'ingénierie* qui décrivent comment chaque type de service technique doit (dé)sérialiser ses contextes d'exécution pour leur propagation à distance.

5.8 Déploiement des beans

Un conteneur est l'unité de déploiement, dans les spécifications EJB. Or, ces spécifications EJB ne précisent pas les concepts de "conteneur EJB" et de "serveur EJB". Pour définir ces termes dans l'architecture que nous proposons, nous nous basons sur les concepts définis dans le langage d'ingénierie de RM-ODP.

Les instances des beans sont gérées pour chaque bean indépendamment, et les instances d'un même bean sont toutes gérées de la même manière. Donc nous considérons que chaque bean (et donc toutes ses instances) est déployé dans une *grappe (cluster)* distincte (au sens donné dans RM-ODP), que nous qualifions de *conteneur EJB*. Cette différenciation des conteneurs nous permet de mettre en oeuvre des services techniques différents pour chaque bean déployé, même si plusieurs conteneurs peuvent partager des modules de services techniques communs dans un même serveur. La seule différence entre un conteneur EJB et une grappe est que des beans dans un même serveur, mais dans des conteneurs différents, peuvent interagir sans passer par des liens distants, pour respecter la sémantique des interfaces locales (`EJBLocalObject`, `EJBLocalHome`) introduites dans les spécifications EJB 2.0.

De plus, contrairement à une grappe, chaque objet abstrait (session ou entité) dans notre conteneur est traité séparément pour ce qui est des services techniques tels que l'activation / passivation. Chaque session ou entité est donc gérée dans un environnement qui peut être informellement qualifié de "sous-grappe", et qui est matérialisé par tous les objets d'interposition serveur qui gèrent son état.

Nous appelons *serveur EJB* un ensemble de conteneurs EJB pouvant partager des modules de services techniques et des objets de canaux de communication. Deux serveurs EJB sont isolés, et ne peuvent interagir que par des interactions distantes. Nous proposons donc de n'exécuter qu'un seul serveur EJB dans une machine virtuelle. Or, il ne peut s'exécuter qu'une seule machine virtuelle par processus, ce qui correspond à la notion de *capsule* de RM-ODP. Notre serveur EJB correspond donc à une capsule.

Nos outils de déploiement permettront de spécifier les services disponibles à l'échelle d'un serveur EJB, et de configurer les services utilisés dans chaque conteneur, et composés dans chaque talon serveur générique pour chaque bean déployé. Nous envisageons aussi d'utiliser des outils de déploiement spécifiques pour mettre en oeuvre les extensions au modèle EJB que nous avons présentées dans le chapitre 4 : la composition des beans, la configuration des canaux pour optimiser les interactions entre beans dans un même conteneur, la vérification statique des assemblages de beans, etc. Nous prévoyons que ces outils de déploiement, et nos propositions d'extensions du modèle de beans, auront un impact minimal sur l'architecture que nous avons proposée dans ce chapitre.

Nous envisageons aussi de pouvoir générer statiquement, avant l'exécution d'un serveur, les objets d'interpositions serveur et client, comme cela est réalisé par les outils de déploiement du serveur JOnAS. Ceci est possible, car tous les éléments constituant les objets d'interposition (i.e. objets proxy, objets adaptateurs, talon générique serveur ou client, aspects de services techniques, talon (dé)sérialiseur) sont génériques. Il suffirait donc de les assembler et de les spécialiser statiquement pour un bean, un ensemble de services techniques et un type de canal donnés. Ainsi, nous pourrions générer le code d'un objet d'interposition pour chaque interface locale ou distante d'un bean, qui implanterait pour chacune de ses méthodes les fonctionnalités suivantes :

- réification des interactions, comme dans les objets proxy,
- récupération des contextes d'exécution propagés implicitement, comme dans les objets adaptateurs,

- propagation implicite des contextes d’exécution, comme dans le talon serveur générique,
- mise en oeuvre des aspects des services techniques, comme dans le talon serveur générique,
- (dé)sérialisation des interaction dans un canal, comme dans le talon sérialiseur ou désérialiseur,
- propagation implicite de contextes d’exécution à distance, comme dans le talon (dé)sérialiseur et les modules de (dé)sérialisation des services techniques,
- interaction avec les instances de bean, pour appeler leurs méthodes métier, comme dans l’aspect de base du talon serveur générique.

Cette proposition est réalisable en utilisation une programmation par aspects par modification de code source, avec les mêmes points de jonction que JAC (voir section 5.3). Ainsi, il est envisageable d’utiliser directement ici les modules d’aspects des services techniques, spécifiques à JAC, sans les modifier.

Avec une telle proposition, nous pourrions obtenir des objets d’interposition similaires à ceux qui sont produits par les outils de déploiement du serveur JONAS. Mais, contrairement à JONAS, nous avons le choix de générer ces objets d’interposition dynamiquement, comme nous l’avons présenté dans les sections précédentes, ou statiquement, comme dans JONAS. Notre architecture serait donc suffisamment générique pour que les utilisateurs puissent choisir le niveau de genericité qu’ils souhaitent obtenir avec ces objets d’interposition, et les adapter à leurs contraintes de flexibilité et de performances.

5.9 Synthèse

La figure 5.10 représente le modèle complet de l’architecture que nous proposons pour notre serveur EJB, sous la forme d’un diagramme de classes UML. Ce diagramme synthétise donc toute la démarche que nous avons menée.

Conformément à notre proposition de la section 5.1, nous pouvons vérifier sur ce diagramme que nous avons globalement séparé l’architecture de notre serveur EJB en plusieurs grandes parties :

- des *beans*. Dans ce diagramme, nous avons modélisé un seul bean, dans le paquetage `un.bean1`, qui correspond à un entity bean ou un session bean, et qui offre seulement des interfaces distantes (`InterfaceDistante1` et `InterfaceHome1`).
- des *services techniques*. Dans ce diagramme, nous avons modélisé un seul service technique, dans le paquetage `un.service1`.
- des *canaux* de communication pour interagir à distance avec notre serveur et le bean qui y est déployé. Dans ce diagramme, nous avons modélisé un seul canal, dans le paquetage `un.canal`.
- notre *conteneur générique*, défini dans le paquetage `objet_interp`.

Nous avons bien inversé les dépendances entre les services techniques, les canaux et notre serveur. En effet, les seules dépendances sortantes de ce serveur concernent les APIs standards de Java et des EJB, qui sont très stables, et les beans déployés, à partir d’objets proxy générés automatiquement et dynamiquement. Ces dépendances sont donc tout à fait acceptables et maîtrisées. Notre conteneur est donc bien “ouvert-fermé” vis-à-vis de son environnement.

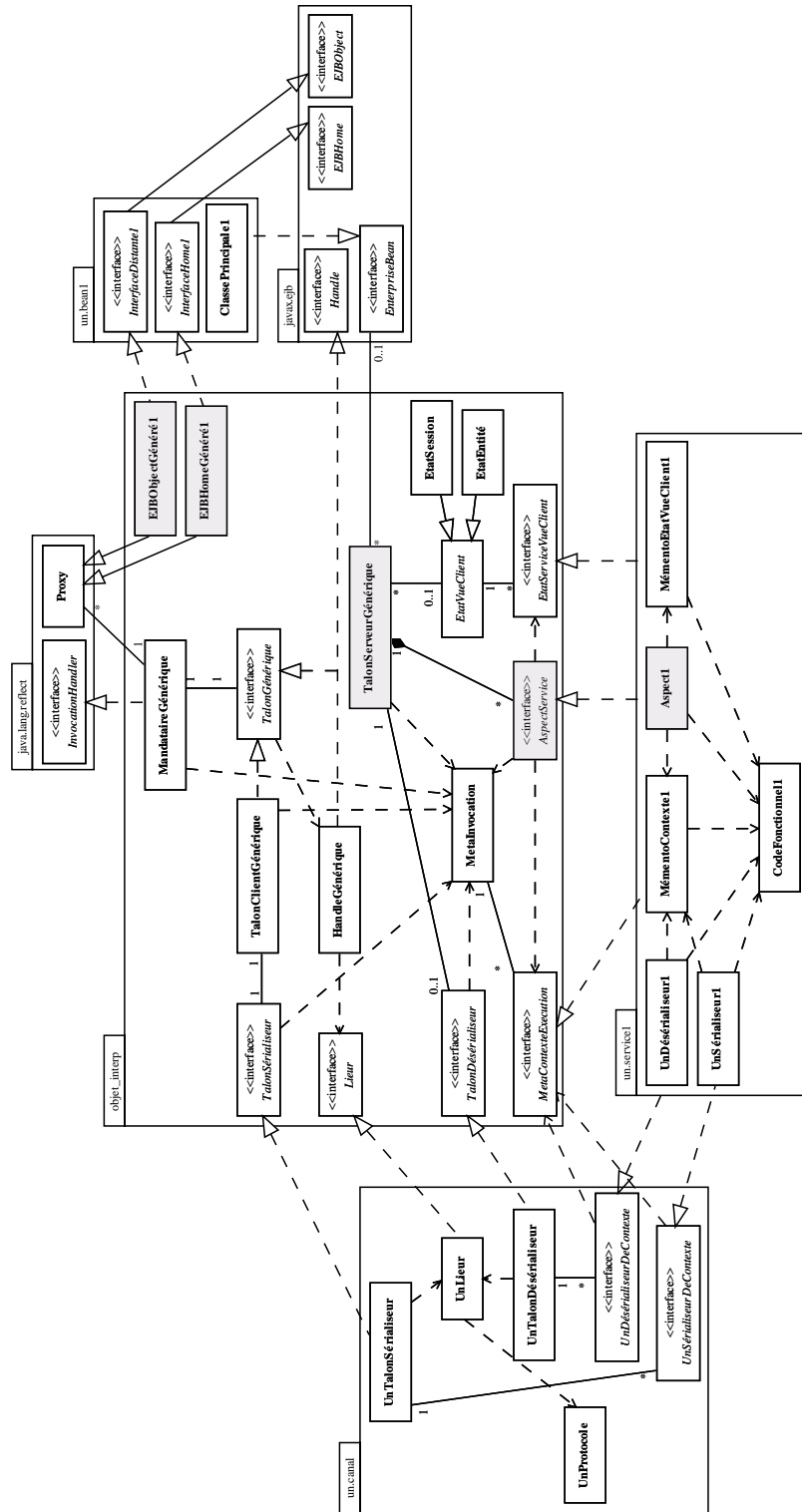


FIG. 5.10 – Architecture globale de notre conteneur

Nous résumons dans la suite de cette section les différents éléments composant les services techniques et les canaux intégrés dans notre serveur EJB, et les services de base fournis par ce serveur.

Services techniques

Pour chaque service technique, par exemple dans le paquetage `un.service1` dans le diagramme, doivent être fournis :

- un *module fonctionnel*, qui implante la fonctionnalité du service technique. Ici, ce module est la classe `CodeFonctionnel1`. Nous pouvons observer sur le diagramme que ce module ne dépend d’aucun autre module : le module fonctionnel d’un service peut donc être réutilisé dans des contextes différents, y compris en dehors d’un serveur EJB.
- un *module d’aspect*, spécifique à la fois au service technique et à notre conteneur, qui sert à mettre en oeuvre ce service technique dans un talon serveur générique. Ici, ce module est la classe `Aspect1`. Ce module a pour seule fonction d’interagir avec le module fonctionnel. Ce module d’aspect est composé avec les modules d’aspects des autres services techniques, dans les talons serveurs génériques, par programmation par aspects. Les classes `TalonServeurGénérique`, `AspectService` et `Aspect1`, grisées dans le diagramme, sont les seules classes concernées par la programmation par aspects.
- un *module d’état*, pour conserver l’état des objets abstraits manipulés par les clients (sessions et entités), spécifique au service technique. Ici, ce module est la classe `MementoEtatVueClient1`. Si un service technique ne conserve pas d’état, il ne doit pas fournir de module d’état.
- un *module de contexte*, pour encapsuler l’état qui est associé aux activités qui réalisent des interactions avec les beans, et qui est spécifique au service technique. Ici, un module de contexte est la classe `MementoContexte1`. Si un service technique ne propage pas de contexte d’exécution, il ne doit pas fournir de module de contexte.
- un *module de désérialisation*, et éventuellement un *module de sérialisation*, pour propager les contextes d’exécution du service technique lors des interactions distantes, dans un type de canal donné. Ici, de tels modules sont les classes `UnDésérialiseur1` et `UnSérialiseur1`. Si un service technique ne propage pas de contextes d’exécution, il ne doit pas fournir de modules de (dé)sérialisation de contextes. Ces modules implantent des interfaces spécifiques à chaque type de canal, et donc un service technique qui propage des contextes d’exécution doit fournir une implantation de tels modules pour chaque type de canal supporté. Un module de sérialisation ne doit être fourni que pour les types de canaux qui requièrent que les outils du serveur EJB génèrent des objets d’interposition client.

Canaux

Pour chaque type de canal, par exemple dans le paquetage `un.canal` dans le diagramme, doivent être fournis :

- un *talon désérialiseur*, qui désérialise et réifie les interactions qui ont été transmises dans un canal, dans des objets manipulables par notre conteneur

(classe `MetaInvocation`). Ici, un tel module est la classe `UnTalonDésérialiseur`. Une instance de ce module est liée à un talon serveur générique, dans un canal donné. Un talon désérialiseur doit spécifier comment les contextes d'exécution des services techniques, attachés aux activités, peuvent être désérialisés. Par exemple, des interfaces peuvent être spécifiées, comme ici l'interface `UnDésérialiseurDeContexte`.

- un *talon sérialiseur*, qui est le symétrique d'un talon désérialiseur. Ici, un tel module est la classe `UnTalonSérialiseur`. Un tel module ne doit être fourni que si le type de canal nécessite que notre serveur EJB génère des objets d'interposition clients.
- un *lieur*, pour maintenir les liens dans des canaux, et implanter la fonctionnalité des classes `Handle` et `HomeHandle` spécifiées dans les EJB. Un exemple de lieu ici est la classe `UnLieur`.
- d'autres modules pour pouvoir construire les canaux, tels qu'un module de protocole, par exemple ici la classe `UnProtocole`. L'architecture interne des canaux est spécifique à chaque type de canal.

Services de base

Notre serveur EJB fournit des services de base, pour mettre en oeuvre les modules des services techniques et des canaux que nous avons spécifiés ci-dessus :

- génération automatique et dynamique des objets proxy spécifiques aux interfaces de chaque bean. Par exemple, les classes `EJBObjectGénéral` et `EJBHomeGénéral`, en grisé dans le diagramme, sont des objets proxy générés automatiquement pour les interfaces du bean `un.bean1`. Les objets proxy sont les seuls objets de notre serveur EJB qui dépendent des beans déployés.
- génération automatique et dynamique des objets d'interposition, et notamment des talons génériques par programmation par aspects.
- propagation implicite des contextes d'exécution lors des interactions locales avec les instances des beans.
- gestion de l'état global des objets abstraits (sessions et entités) manipulés par les clients.
- outils de déploiement, pour la configuration des services techniques et des canaux disponibles dans un serveur et mis en oeuvre dans chaque conteneur, pour la vérification statique des configurations de beans, l'optimisation des ressources (e.g. présence de canaux ou non pour un bean), etc.

Les autres services de base spécifiques aux EJB, tels que l'activation / passivation des instances de beans, peuvent être spécifiés comme les autres services techniques indépendants des EJB.

Chapitre 6

Conclusion

Nous avons exposé dans ce document la problématique de la construction de systèmes répartis ouverts. Nous avons en particulier étudié les problèmes posés par la conception d'un support d'exécution de composants répartis, et notamment par la conception d'un conteneur EJB. La principale propriété que nous recherchons pour un tel conteneur est une capacité d'extension en ce qui concerne les services techniques mis en oeuvre, tels que la gestion de transactions, et les canaux pour interagir à distance avec les composants. Nous avons proposé une architecture qui nous permet d'atteindre cet objectif.

Nous avons mené une démarche structurée, pour identifier les points stratégiques d'un conteneur EJB, afin de rendre ce conteneur adaptable et extensible. Cette démarche s'est appuyée sur des principes généraux de conception par objets, tels que le principe d'ouverture-fermeture, et sur un canevas standard de spécification de systèmes répartis ouverts : RM-ODP.

Nous avons montré que l'application de ces principes de conception dans un conteneur EJB ne peut pas être réalisée de manière simple avec seulement les paradigmes de la programmation par objets classique. En effet, un des problèmes que nous avons rencontré est qu'un conteneur met en oeuvre un ensemble de services techniques qui sont mélangés et dispersés dans tout le conteneur.

Nous avons donc présenté des techniques de séparation et de composition de problèmes (*separation of concerns*), qui nous permettent de résoudre simplement ce problème. Nous avons ainsi proposé d'utiliser la programmation par aspects pour implanter les objets d'interposition, qui sont une partie stratégique d'un conteneur EJB. Globalement, notre travail a donc été d'étudier les techniques de séparation et de composition de services techniques, dans le contexte spécifique de la construction d'un conteneur EJB. Mais pour des raisons de complexité et d'efficacité, le modèle d'architecture que nous proposons utilise en plus d'autres démarches et techniques, et notamment les patrons de conception réutilisables (*design patterns*) et la réflexivité.

Nous avons comparé notre architecture avec celle d'un conteneur EJB existant : JOnAS. Nous avons montré que notre architecture est plus ouverte et extensible que celle de JOnAS. Nous avons notamment montré que, contrairement à JO-

nAS, notre architecture est “ouverte-fermée” à la mise en oeuvre de nouveaux services techniques et de nouveaux types de canaux de communication.

Nous n’avons actuellement fourni qu’une spécification de notre conteneur, et nous travaillons sur une implantation. Cette implantation nous permettra de valider cette architecture, notamment en termes de performances pour ce qui est de l’utilisation de la programmation par aspects. Cela nous permettra également de tester les outils existants pour la séparation et la composition de problèmes, tels que JAC.

Les perspectives de ce travail, que nous allons explorer pendant la thèse, s’appuieront sur un raffinement des techniques de séparation et de composition des services techniques, et sur l’introduction d’un modèle de composants techniques pour les exprimer. Nous allons aussi spécifier tous les services techniques mis en oeuvre dans les EJB, et étendre ces spécifications à de nouveaux services techniques, comme la duplication et la migration de composants. Ces travaux nous permettront d’accroître encore la flexibilité de notre conteneur EJB, et des supports d’exécution répartis en général.

Bibliographie

- [ABV92] Mehmet Askit, Lodewijk M.J. Bergmans, and Sinan Vural. An object-oriented language-database integration model : The composition-filters approach. In *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92)*, volume 615, pages 372–395. Springer-Verlag, June 1992.
- [And96] D. H. Andrews. Ibm's san francisco project : Java frameworks for application developers, December 1996.
- [BSL] Noury M. N. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. *Techniques et sciences informatiques*, 20(4).
- [CE00] Thierry Coupaye and Jacky Estublier. Foundations of enterprise software deployment. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'2000)*. IEEE Computer Society Press, March 2000.
- [Cha99] Jean-Marie Chauvet. *Composants et transactions : Corba/OTS, EJB/JTS, COM/MTS*. Eyrolles, Paris, France, 1999.
- [DHTS98] Bruno Dumant, François Horn, Frédéric Dang Tran, and Jean-Bernard Stefani. Jonathan : an open distributed processing environment in java. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 175–190. Springer-Verlag, 1998.
- [DmYK01] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. Enterprise Java Beans Specification Version 2.0 Proposed Final Draft 2. Sun Microsystems Inc., April 24, 2001.
- [ea99] BEA Systems et al. CORBA Component Model joint revised submission. Object Management Group, OMG document orbos/99-07-01 ed., July 1999.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
- [GJS96] James Gosling, Bill Joy, and Guy Steele. *The JavaTM Language Specification*. Addison-Wesley, August 1996.
- [GM01] Marc Guiot and Régis Medina. Principes avancés de conception objet – <http://www.design-up.com/design/principesoo/index.html>, March 2001.

- [ISO95a] ISO. ITU/ISO Reference Model of Open Distributed Processing – Part 2 : Foundations, International Standard ISO/IEC 10746–2, ITU–T Recommendation X.902, 1995, 1995.
- [ISO95b] ISO. ITU/ISO Reference Model of Open Distributed Processing – Part 3 : Architecture, International Standard ISO/IEC 10746–3, ITU–T Recommendation X.903, 1995, 1995.
- [ISO95c] ISO. Reference Model of Open Distributed Processing. International Standard ISO/IEC 10746, ITU–T Recommendations X.901–904, 1995, 1995.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Jyväskylä, Finland, volume 1241, pages 220–242. Springer-Verlag, 1997.
- [LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6) :1811–1841, November 1994.
- [Mar96a] Robert C. Martin. The Dependency Inversion Principle. *C++ Report*, May 1996.
- [Mar96b] Robert C. Martin. Granularity. *C++ Report*, November 1996.
- [Mar96c] Robert C. Martin. The Interface Segregation Principle. *C++ Report*, August 1996.
- [Mar96d] Robert C. Martin. The Liskov Substitution Principle. *C++ Report*, March 1996.
- [Mar97] Robert C. Martin. Stability. *C++ Report*, February 1997.
- [MC95] Microsoft Corporation. The Component Object Model specification, March 1995.
- [Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.
- [MK90] John D. McGregor and Timothy D. Korson. Understanding object-oriented : A unifying paradigm. *Communications of the ACM*, 33(9) :40–60, 1990.
- [MT97] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer-Verlag, 1997.
- [OHBS94] Harold Ossher, William Harrison, Franck Budinsky, and Ian Simmonds. Subject-oriented programming : Supporting decentralized development of objects. In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, 1994.
- [PSDF01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC : A flexible and efficient framework for aop in java. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, Kyoto, Japan, September 2001.

- [RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
- [SDK⁺95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4) :314–335, April 1995.
- [Ste95] Jean-Bernard Stefani. Open distributed processing : an architectural basis for information networks. *Computer Communications*, 18(11) :849–862, November 1995.

Annexe A

Concepts de OSI RM-ODP

Cette annexe résume les concepts généraux définis dans la partie II du standard OSI Reference Model of Open Distributed Processing [ISO95a]. Cette partie du standard est utile pour interpréter les constructions des langages de spécification de RM-ODP. Ces concepts sont répartis dans différentes catégories, dont les principales sont présentées dans cette annexe.

Concepts d'interprétation de base

Cette catégorie définit des méta-concepts, qui peuvent être utilisés dans toute démarche de modélisation :

entité (*entity*) une chose concrète ou abstraite qui a de l'intérêt.

abstraction la démarche qui consiste à ignorer les détails inutiles pour établir un modèle simplifié ; ou le résultat de cette démarche.

atomicité (*atomicity*) une entité est atomique à un certain niveau d'abstraction si elle ne peut pas être subdivisée dans ce niveau d'abstraction.

système (*system*) une chose qui a de l'intérêt comme un tout, ou comme composée de ces parties. Un système peut être considéré comme une entité. Un système peut comprendre d'autres systèmes : dans ce cas, ce sont des sous-systèmes (subsystems).

architecture un ensemble de règles qui définissent la structure d'un système et les relations entre ses parties.

Concepts de modélisation de base

Cette catégorie définit des concepts de modélisation, indépendamment du langage d'implantation :

objet (*object*) un modèle d'une entité. Un modèle est caractérisé par son comportement et son état. Les objets sont tous distincts.

environnement (*environment*) (d'un objet) la partie du modèle qui n'est pas l'objet.

action une chose qui arrive. Il y a deux types d'actions : les actions internes et les actions externes (interactions). Une *action interne* n'implique qu'un seul objet, sans la participation de son environnement. Une *interaction* implique la participation de l'environnement de l'objet.

interface abstraction du comportement d'un objet qui consiste en un ensemble d'interactions de cet objet, avec des contraintes pour l'occurrence de ces interactions. Toute interaction d'un objet fait partie d'une interface unique : les interfaces d'un objet forment donc une partition des interactions de ce objet.

activité (*activity*) séquence d'actions.

comportement (*behaviour*) (d'un objet) une collection d'actions avec un ensemble de contraintes sur leur occurrence. Ces contraintes peuvent être par exemple des contraintes de séquentialité, de non-déterminisme, de concurrence ou de temps-réel.

état (*state*) (d'un objet) à un instant donné, les conditions qui déterminent l'ensemble de toutes les séquences d'actions auxquelles l'objet peut participer. Les objets sont encapsulés : l'état d'un objet ne peut pas être modifié directement par l'environnement, mais seulement par les actions auxquelles il participe.

communication transmission d'information entre deux objets ou plus, comme résultat d'une ou plusieurs interactions.

localisation (*location*) un intervalle arbitraire dans l'espace (ou dans le temps), dans lequel une action peut se produire.

point d'interaction une localisation où il existe un ensemble d'interfaces.

Concepts de spécification

composition (d'objets) la combinaison de deux objets ou plus, formant un nouvel objet, à un niveau d'abstraction différent. Les caractéristiques du nouvel objet sont déterminées par les objets combinés, et la façon dont ils sont combinés. Le comportement du nouvel objet est la composition des comportements des objets combinés.

composition (de comportements) la combinaison de deux comportements ou plus, formant un nouveau comportement. Les caractéristiques du nouveau comportement sont déterminées par les comportements combinés, et la façon dont ils sont combinés.

objet composite (*composite object*) un objet exprimé comme une composition.

décomposition la spécification d'un objet ou d'un comportement donné par une composition. C'est l'activité duale de la composition.

compatibilité comportementale (*behavioural compatibility*) un objet est comportementalement compatible (*behaviourally compatible*) avec un second objet, respectivement à un ensemble de critères, si le premier objet peut

remplacer le second objet sans que l'environnement puisse détecter de différence dans le comportement des objets selon l'ensemble de critères.

raffinement (*refinement*) démarche de transformation d'une spécification en une spécification plus détaillée.

trace un enregistrement des interactions auxquelles participe un objet, depuis son état initial jusqu'à un autre état. Le comportement d'un objet détermine de façon unique l'ensemble de toutes ses traces possibles, mais l'inverse n'est pas vrai.

type (d'un $\langle X \rangle$) un prédicat caractérisant une collection de $\langle X \rangle$ s. Un $\langle X \rangle$ "est de ce type", ou "satisfait ce type", si le prédicat est vérifié pour ce $\langle X \rangle$. Dans RM-ODP, des types sont nécessaires pour les objets, les interfaces et les actions.

classe (*class*) (de $\langle X \rangle$ s) l'ensemble des $\langle X \rangle$ s satisfaisant un type. Les éléments de cet ensemble sont membres de cette classe.

sous-type / super-type (*subtype/supertype*) un type A est sous-type d'un type B, et B est un super-type de A, si tout $\langle X \rangle$ qui satisfait A satisfait aussi B.

sous-classe / super-classe (*subclass/superclass*) une classe A est une sous-classe d'une classe B, et B est une super-classe de A, si le type associé à A est un sous-type du type associé à B.

patron de $\langle X \rangle$ (*$\langle X \rangle$ template*) la spécification des fonctionnalités communes d'une collection de $\langle X \rangle$ s, suffisamment détaillée pour pouvoir instancier un $\langle X \rangle$. C'est une abstraction de la collection de $\langle X \rangle$ s. $\langle X \rangle$ peut être n'importe quoi aillant un type. Peut spécifier des paramètres à renseigner lors d'une instantiation.

signature d'interface (*interface signature*) l'ensemble des patrons d'actions associés aux interactions d'une interface.

instanciation (*instantiation*) (d'un patron de $\langle X \rangle$) un $\langle X \rangle$ produit à partir d'un patron de $\langle X \rangle$, et d'autres informations nécessaires à cette instantiation. Le $\langle X \rangle$ produit exhibe les mêmes fonctionnalités que celles spécifiées dans le patron de $\langle X \rangle$.

création (*creation*) (d'un $\langle X \rangle$) instantiation d'un $\langle X \rangle$ par une action impliquant des objets du modèle.

introduction (d'un $\langle X \rangle$) instantiation d'un $\langle X \rangle$ non réalisée par une action impliquant des objets du modèle.

suppression (*deletion*) (d'un $\langle X \rangle$) l'action de destruction d'un $\langle X \rangle$ instancié.

rôle (*role*) identifiant d'un comportement, qui peut apparaître comme paramètre dans un patron d'un objet composite, et qui est associé à l'un des objets composants l'objet composite.

instance (d'un type) un $\langle X \rangle$ qui satisfait le type.

type de patron (*template type*) (d'un $\langle X \rangle$) un prédicat défini dans un patron, qui est vérifié pour toutes les instantiations du patron.

type de classe (*class template*) (d'un $\langle X \rangle$) l'ensemble de tous les $\langle X \rangle$ s satisfaisant un type de patron de $\langle X \rangle$, c'est-à-dire l'ensemble des $\langle X \rangle$ s qui sont les instances du patron de $\langle X \rangle$.

invariant un prédicat, requis par une spécification, qui doit être vérifié pendant toute la durée de vie de l'ensemble d'objets.

précondition (*precondition*) un prédicat, requis par une spécification, qui doit être vérifié pour qu'une action puisse se produire.

postcondition un prédicat, requis par une spécification, qui doit être vérifié immédiatement après l'occurrence d'une action.

Concepts d'organisation

configuration (d'objets) une collection d'objets capables de communiquer via leurs interfaces. Une configuration détermine l'ensemble des objets impliqués dans chaque interaction. Sa spécification peut être statique ou dynamique (par attachement et détachement).

domaine $\langle X \rangle$ ($\langle X \rangle$ domain) un ensemble d'objets, dont chaque objet est associé à un objet contrôleur (*controlling object*) par une relation caractéristique $\langle X \rangle$ ($\langle X \rangle$ *characterizing relationship*). Tout domaine a un objet contrôleur associé. Exemple : domaine d'administration (*administration domain*).

époque (*epoch*) une période de temps pendant laquelle l'objet expose un comportement donné. Chaque objet est dans une seule époque à un moment donné. Plusieurs objets qui interagissent peuvent être dans des époques différentes.

point de référence (*reference point*) un point d'interaction défini dans une architecture, pouvant être choisi comme point de conformité dans une spécification à laquelle se conforme cette architecture.

point de conformité (*conformance point*) un point de référence où un comportement peut être observé pour tester la conformité d'une architecture.

Propriétés des systèmes et objets

contrat un accord réglant une partie du comportement commun à un ensemble d'objets. Un contrat spécifie des obligations, permissions et interdictions pour les objets impliqués.

contrat d'environnement (*environment contract*) un contrat entre un objet et son environnement. Les contraintes d'environnement décrivent à la fois les contraintes sur l'environnement pour un comportement correct de l'objet, et les contraintes sur le comportement de l'objet dans un environnement correct.

persistance (*persistence*) la propriété qu'un objet continue d'exister à travers des changements de contexte contractuel.

Concepts pour les comportements

comportement d'établissement (*establishing behaviour*) le comportement par lequel un contrat donné est mis en place entre des objets donnés.

Il peut être explicite (résultat des interactions entre les objets prenant part au contrat) ou implicite (mis en place par un objet externe, ou dans une époque précédente).

comportement permis (*enabled behaviour*) le comportement caractérisant un ensemble d'objets, rendu possible par un comportement d'établissement.

contexte contractuel (*contractual context*) la connaissance qu'un contrat donné est en place. Un objet peut être dans un nombre quelconque de contextes contractuels simultanément ; son comportement est contraint par l'intersection des comportements prescrits par chaque contexte contractuel.

liaison la relation entre des objets, qui résulte d'un comportement d'établissement.

comportement de terminaison (*terminating behaviour*) le comportement qui brise une liaison, et rejete le contexte contractuel et le contrat correspondants.

objet initiateur (*initiating object*) un objet qui produit une communication.

objet répondeur (*responding object*) un objet participant à une communication, qui n'en est pas l'objet initiateur.

objet producteur (*producer object*) un objet qui est la source de l'information transmise, dans une communication.

objet consommateur (*consumer object*) un objet qui est une destination de l'information transmise, dans une communication.

objet client (*client object*) un objet qui requiert l'exécution d'une fonction par un autre objet.

objet serveur (*server object*) un objet qui exécute une fonction à la demande d'un objet client.

comportement de ligature (*binding behaviour*) un comportement d'établissement entre deux interfaces ou plus.

lien (*binding*) un contexte contractuel, résultant d'un comportement de ligature. Un binding est un cas particulier de liaison.

comportement de détachement (*unbinding behaviour*) un comportement qui termine un lien.

défaillance (*failure*) violation d'un contrat. Un contrat exprimant le comportement correct des objets, une défaillance est un comportement incorrect.

erreur (*error*) partie de l'état d'un objet qui peut conduire à des défaillances. La manifestation d'une faute dans un objet.

faute (*fault*) une situation qui peut causer des erreurs dans un objet.

stabilité (*stability*) la propriété qu'a un objet, respectivement à un mode de défaillance donné, s'il ne peut pas exhiber ce mode de défaillance.