



HAL
open science

Une Approche Générique pour la Reconfiguration Dynamique des Applications à base de Composants Logiciels

Abdelmadjid Ketfi

► **To cite this version:**

Abdelmadjid Ketfi. Une Approche Générique pour la Reconfiguration Dynamique des Applications à base de Composants Logiciels. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 2004. Français. NNT: . tel-00008771

HAL Id: tel-00008771

<https://theses.hal.science/tel-00008771>

Submitted on 14 Mar 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE JOSEPH FOURIER

THESE

pour obtenir le grade de
DOCTEUR de l'Université Joseph Fourier de Grenoble

Discipline : Informatique

présentée et soutenue publiquement par

Abdelmadjid KETFI

le 10 décembre 2004

UNE APPROCHE GNERIQUE POUR LA RECONFIGURATION DYNAMIQUE DES APPLICATIONS A BASE DE COMPOSANTS LOGICIELS

JURY

Y. Chiaramella	Professeur UJF, Grenoble	(Président)
J. M. Andreoli	Directeur de recherche, CNRS Xerox Research Centre Europe	(Rapporteur)
I. Borne	Professeur Université de Bretagne-Sud	(Rapporteur)
F. Plasil	Full Professor Charles University, Prague	(Examineur)
N. Belkhatir	Professeur UPMF, Grenoble	(Directeur de thèse)
P. Y. Cunin	Professeur UJF, Grenoble	(Directeur de thèse)

*à toi ma chère maman,
à toi mon cher papa,
à ma femme et à mon petit ange Inès,
à mon oncle qui a toujours été pour moi un 2^{ème} papa.*

« Ceci est mon travail de thèse... mais c'est avant tout un travail d'équipe dans lequel chacun a apporté plus ou moins sa petite perle... sans ces perles, ce travail ne serait bien évidemment pas ce qu'il est... »

Je remercie d'abord M. Yves Chiaramella Professeur à l'UJF, M. Jean-Marc Andreoli Directeur de recherche au CNRS, Mme Isabelle Borne Professeur à l'UBS et M. Frantisek Plasil Professeur à l'Université Charles d'avoir accepté d'évaluer mon travail de thèse et de faire partie du Jury de ma soutenance.

Je tiens à remercier chaleureusement mes directeurs de thèse, M. Nouredine Belkhatir Professeur à l'UPMF et M. Pierre-Yves Cunin Professeur à l'UJF, pour leur soutien permanent au cours de ces trois dernières années, pour leur conseils et leur grande patience, surtout dans les moments de doute, où je suis difficile à convaincre. Merci à toi Nourry pour ta disponibilité permanente. Merci à toi Pierre-Yves pour la lecture attentive de mon manuscrit et pour tes remarques pertinentes tant sur le fond que sur la forme.

Je remercie Noëlle pour la lecture minutieuse de mon manuscrit.

Je remercie tous les membres de l'équipe de Génie Logiciel Adèle pour l'ambiance agréable, l'aide, le soutien et la bonne humeur. Merci pour les apports à la fois humains et scientifiques qui ont fortement participé à la mise en œuvre de ce travail.

Un grand Merci à ma femme, et à tous les membres de ma famille, qui m'ont toujours apporté leur interminable soutien durant toutes ces années, et qui n'ont jamais cessé d'être à mes côtés, malgré les barrières géographiques...

Une grande reconnaissance à tous mes enseignants, du primaire à l'université, de m'avoir inculqué l'amour de la connaissance.

Je tiens à saluer et à remercier sincèrement tous mes ami(e)s qui ont toujours été là quand il fallait, et qui n'ont jamais eu marre d'apprécier les hauts et de supporter les bas de ma vie. Si je dois nommer quelqu'un, ça sera bien sûr toi Tahar^{1,2}...

*« Les sciences sont des lunettes pour grossir les problèmes »
L. Scutenaire*

*« Informatique : Alliance d'une science inexacte et d'une activité humaine faillible »
L. Fayard*

Résumé

Le déploiement est une phase qui prend de plus en plus d'importance dans le cycle de vie du logiciel. Il prend le relais après la phase de développement et couvre les étapes restantes du cycle de vie : de l'installation jusqu'à la désinstallation du logiciel en assurant sa maintenance corrective et évolutive . Cette maintenance est nécessaire pour prendre en compte de nouvelles conditions, non envisagées et difficiles à prédire dans la phase de développement.

Depuis plusieurs années, le développement du logiciel s'oriente vers un nouveau paradigme permettant de simplifier la maintenance. Ce paradigme a pour objectif la construction d'applications en intégrant des briques logicielles bien définies et "autonomes", appelées composants. La notion classique de développement d'applications en écrivant du code a été remplacée par l'assemblage de composants préfabriqués. En général, pour introduire les modifications nécessaires, l'application doit être arrêtée, modifiée, recompilée puis démarrée à nouveau. Cependant, ce processus classique de maintenance ne peut pas être appliqué à certaines catégories d'applications, pour lesquelles l'interruption complète de service ne peut pas être tolérée. Ces applications, qualifiées de "non-stop", doivent être adaptées d'une manière dynamique avec le minimum de perturbation.

Dans cette thèse, nous proposons une approche réflexive permettant de supporter la reconfiguration dynamique des applications à base de composants. Cette approche est mise en œuvre sous forme du système DYVA. Le rôle de ce système est de prendre en charge la responsabilité de reconfiguration, et de permettre aux développeurs de se concentrer sur la logique applicative. Notre objectif majeur est de proposer une solution de reconfiguration générale, basée sur des abstractions et séparée des applications à reconfigurer. Ceci permet de développer des applications propres, qui focalisent sur la logique métier. La séparation favorise aussi l'évolution et facilite la maintenance, aussi bien des applications que du système de reconfiguration. Pour assurer l'ouverture et la réutilisabilité de notre système, nous l'avons basé sur un modèle de composants abstrait. Ce modèle incarne l'image de l'application à reconfigurer et permet de traiter d'une manière homogène, des applications issues de modèles de composants différents. Notre approche favorise aussi l'auto-reconfiguration qui reflète la capacité du système de reconfiguration à prendre des décisions de reconfiguration cohérentes, et à matérialiser ces décisions sans l'intervention d'un acteur humain.

Mots clés

Reconfiguration, adaptation, dynamique, déploiement, composant, DYVA, autonome, non-stop, réflexion.

Abstract

Recent software engineering technologies, and in particular those related to component-based software engineering (CBSE), have highlighted an important part of the software life cycle known as *deployment*. This part, usually mixed up with software installation, covers the remaining software life cycle phases, starting after the development and ensuring the software maintenance until its removal. This maintenance is required to take into account new user requirements and new conditions, not considered and usually unpredictable at build-time.

CBSE focuses on building large software systems by integrating existing software components. A software system is no longer obtained through writing code but rather through assembling existing components. CBSE aims to enhance the flexibility and to facilitate the maintainability of developed systems. Usually, the system to be maintained has to be stopped, updated, rebuilt and finally restarted. This maintenance process is not suitable for "non-stop" critical systems that have to be highly available like bank, aeronautic, e-commerce and telecommunication services. With that kind of systems the maintenance is more complex and must take place at run-time.

Dynamic maintenance and reconfiguration at run-time of component-based software systems remains one of the key challenges facing software developers today. This thesis fits in the dynamic reconfiguration research area. It describes a reflexive approach to support the dynamic reconfiguration of component-based software systems. This approach has been implemented in the DYVA system. The role of this system is to take in charge the reconfiguration responsibility, and to enable developers to focus on the business logic of the software systems they build. Our main aim is to define a dynamic reconfiguration solution, based on abstractions, and independent as much as possible of the targeted software systems and their component models. DYVA is based on an abstract component model that enhances its openness and its reusability. This abstract model reflects the image of the concrete application to be reconfigured and allows homogeneous treatment of systems resulting from different component models. The consistency is an important aspect we considered in our approach through state transfer and communication management during the dynamic reconfiguration. Dynamically reconfiguring an application requires a complex analysis and decision process. Our approach uses a combination of 1) sensor objects that supervise the system environment, 2) reconfiguration rules that represent the reconfiguration logic, and 3) a reasoning engine that takes and applies the reconfiguration decisions.

Keywords

Reconfiguration, adaptation, dynamic, deployment, component, DYVA, autonomous, non-stop, reflection.

Table des matières

CHAPITRE 1 : INTRODUCTION	9
1 Contexte général.....	9
2 Introduction à la problématique de recherche.....	10
3 Principes de conception et démarche suivie.....	12
4 Organisation du document.....	13
CHAPITRE 2 : CONTEXTE SCIENTIFIQUE	15
1 Introduction.....	15
2 Le modèle de programmation par composants.....	15
2.1 Notion de composant.....	16
2.2 Le modèle JavaBeans.....	19
2.3 Le modèle Entreprise JavaBeans.....	22
2.4 Le modèle OSGi.....	24
2.5 Le modèle COM.....	28
2.6 Synthèse.....	30
3 Reconfiguration dynamique.....	31
3.1 Types de reconfiguration.....	31
3.2 Taxonomie.....	32
3.3 Propriétés des systèmes de reconfiguration.....	34
4 Systèmes réflexifs.....	36
4.1 Réification.....	37
4.2 Introspection.....	37
4.3 Intercession.....	37
4.4 Types de réflexion.....	38
5 Conclusion.....	40
CHAPITRE 3 : ETAT DE L'ART	41
1 Introduction.....	41
2 Solutions de reconfiguration fermées.....	42
2.1 Exemple d'adaptabilité dans OSGi.....	43
3 Solutions de reconfiguration partiellement ouvertes.....	44
3.1 OpenVL.....	44
4 Solutions de reconfiguration ouvertes.....	46
4.1 Iguana/J.....	47
5 Solutions de reconfiguration procédurales.....	54
5.1 DYMOS.....	54
5.2 PODUS.....	56
6 Solutions de reconfiguration modulaires.....	59
6.1 Polylith.....	60
7 Solutions de reconfiguration orientées objet.....	65
7.1 Les classes dynamiques de Java.....	65
8 Solutions de reconfiguration orientées composant.....	72
8.1 K-Components.....	72

8.2	DCUP	78
9	Solutions de reconfiguration issues du monde industriel.....	84
9.1	SNMP	84
9.2	JMX.....	87
10	Solutions de reconfiguration matérielles	93
10.1	ACARS.....	93
10.2	SCP.....	93
10.3	Discussion	94
11	Synthèse et conclusion.....	94
11.1	Synthèse des approches étudiées.....	94
11.2	Conclusion sur les approches étudiées.....	96
11.3	Aspects liés aux systèmes de reconfiguration.....	97
CHAPITRE 4 : EXPERIMENTATIONS PRELIMINAIRES		99
1	Introduction.....	99
2	Reconfiguration dynamique dans le modèle JavaBeans	100
2.1	Support de développement : la BeanBox	100
2.2	Extension de la BeanBox : la DBeanBox.....	102
3	Reconfiguration dynamique dans le modèle OSGi	110
3.1	OSGiAdaptor : service OSGi dédié à la reconfiguration.....	111
3.2	Gestion des communications.....	112
4	Adaptabilité dynamique des interfaces.....	113
4.1	Problème à l'origine d'OSL	113
4.2	OSL (OSGiServiceLookup).....	114
5	Synthèses et conclusion	119
5.1	Synthèse de l'expérimentation sur le modèle JavaBeans	119
5.2	Synthèse de l'expérimentation sur le modèle OSGi	120
5.3	Synthèse globale.....	120
5.4	Conclusion sur les expérimentations préliminaires	122
CHAPITRE 5 : DYVA : UN NOYAU GENERIQUE POUR LA RECONFIGURATION DYNAMIQUE.....		125
1	Introduction.....	125
2	Motivations	126
3	DYVA : une conception réflexive	127
4	Architecture interne de DYVA	128
4.1	Modèle abstrait d'application utilisé par DYVA	129
4.2	Gestionnaire de reconfiguration	132
4.3	Capacité d'auto-reconfiguration.....	134
5	Vue externe de DYVA	137
5.1	Interface de notification	138
5.2	Interface de reconfiguration	139
5.3	Plugins de projection	139
6	Problème de transfert d'état	139
7	Processus de personnalisation et d'utilisation	141
8	Conclusion	142

CHAPITRE 6 : DYVA : IMPLEMENTATION.....	145
1 Introduction.....	145
2 Implémentation de l'architecture.....	146
2.1 Modèle abstrait	146
2.2 Organisation du méta-niveau	147
3 Outil d'assistance au transfert d'état	151
4 Interface graphique d'administration	153
5 Base de composants.....	154
5.1 Description explicite	155
5.2 Description partiellement calculée	157
5.3 Outil d'édition et de manipulation de la base de composants.....	159
6 Conclusion	160
CHAPITRE 7 : DYVA : PERSONNALISATION	161
1 Introduction.....	161
2 Interface de notification d'OSGi	162
3 Interface de reconfiguration d'OSGi.....	163
4 Plugins de projection	166
5 Outil d'instrumentation OSGi	170
6 Application de Portail Web.....	173
6.1 Description de l'application	173
6.2 Lancement de l'application.....	174
6.3 Utilisation et reconfiguration dynamique de l'application	175
7 Conclusion	182
CHAPITRE 8 : CONCLUSION ET PERSPECTIVES	183
8 Conclusion	183
8.1 Problématique de reconfiguration dynamique	183
8.2 Principales contributions et résultats obtenus.....	184
9 Perspectives de la thèse.....	186
9.1 Formalisation du problème de transfert d'état.....	187
9.2 Cohérence de la reconfiguration.....	187
9.3 Auto-reconfiguration	188
9.4 Approche par raffinement	188
9.5 Perspectives à court terme	189

Liste des figures

Figure 1. Vision logique d'un JavaBean	20
Figure 2. Communication événementielle de Java	21
Figure 3. Architecture multi-niveaux	22
Figure 4. Exécution des composants EJB	24
Figure 5. Structure d'un bundle OSGi	25
Figure 6. Cycle de vie d'un bundle OSGi	27
Figure 7. Vue d'un composant COM	29
Figure 8. Assemblage de composants COM	30
Figure 9. Utilisation des méta-classes	38
Figure 10. Utilisation des méta-objets	39
Figure 11. Système architecturalement réflexif	39
Figure 12. Solutions de reconfiguration partiellement ouvertes	44
Figure 13. Architecture à plugins de OpenVL	45
Figure 14. Solutions de reconfiguration ouvertes	46
Figure 15. Architecture du système Iguana/J	47
Figure 16. Connexion entre le niveau de base et le méta-niveau	53
Figure 17. Architecture de DYMOS	55
Figure 18. Utilisation des inter-procédures	57
Figure 19. Architecture de PODUS	58
Figure 20. Communication directe entre modules	62
Figure 21. Communication indirecte entre modules	63
Figure 22. Communication entre modules distribués	63
Figure 23. Cycle de vie d'une classe Java	66
Figure 24. Adaptation des instances d'une classe	70
Figure 25. Exemple de graphe	74
Figure 26. Exemple de transformation de graphe	75
Figure 27. Structure d'un composant DCUP	78
Figure 28. Partie permanente et partie remplaçable	79
Figure 29. Références descendantes et montantes	81
Figure 30. Architecture d'administration SNMP	84
Figure 31. Protocole de communication	86
Figure 32. Niveaux de l'architecture JMX	88
Figure 33. Aperçu de la BeanBox	100
Figure 34. Architecture de la BeanBox	101
Figure 35. Architecture de l'application	102
Figure 36. Interface de reconfiguration de la DBeanBox	104
Figure 37. Mapping de l'état et des opérations	105
Figure 38. Passivation d'une instance	106
Figure 39. Reconnexion dynamique	107
Figure 40. Remplacement dynamique des instances	108

Figure 41. Architecture de la DBeanBox	110
Figure 42. Enregistrement et recherche de services	111
Figure 43. Utilisation du service OSL	112
Figure 44. Blocage et libération des appels interceptés	113
Figure 45. Sélection des services de substitution	115
Figure 46. Mapping des opérations	116
Figure 47. Mapping des paramètres et des types de retour	117
Figure 48. Utilisation des informations de mapping dans un appel	117
Figure 49. Mapping multiple d'opérations	118
Figure 50. Code de mapping	118
Figure 51. Architecture après le mapping	119
Figure 52. Conception réflexive de DYVA	127
Figure 53. Architecture de référence de DYVA	129
Figure 54. Modèle abstrait d'application	130
Figure 55. Exemple d'application	130
Figure 56. Cycle d'automatisation	134
Figure 57. Organisation de la base de composants	136
Figure 58. Vue externe de DYVA	137
Figure 59. Modèle abstrait de l'état	140
Figure 60. Personnalisation de DYVA	141
Figure 61. Organisation du méta-niveau	147
Figure 62. Outil de spécification et d'instrumentation d'état	152
Figure 63. Interface graphique d'administration	154
Figure 64. Analyse du code binaire d'une application	158
Figure 65. Outil d'édition et de manipulation	159
Figure 66. Mise à jour de la base de composants	160
Figure 67. Services OSGi	162
Figure 68. Interface textuelle de contrôle OSGi	166
Figure 69. Instrumentation des bundles OSGi	171
Figure 70. Base de composants de l'application portail Web	174
Figure 71. Lancement de l'application	174
Figure 72. Interface graphique de contrôle d'OSCAR	175
Figure 73. Page principale du portail	176
Figure 74. Architecture de l'application portail	177
Figure 75. Affichage de la description d'un composant	178
Figure 76. Simulation des variations de la bande passante	179
Figure 77. Remplacement du composant de diffusion vidéo	181
Figure 78. Reconfiguration à l'aide de l'interface graphique de DYVA	181
Figure 79. Architecture de l'infrastructure du projet PISE	190

CHAPITRE 1

INTRODUCTION

1 CONTEXTE GENERAL

Le déploiement est une phase qui prend de plus en plus d'importance dans le cycle de vie du logiciel [Les03]. Il prend le relais après la phase de développement et couvre plusieurs étapes, allant de l'installation et assurant la maintenance corrective et évolutive du logiciel jusqu'à sa désinstallation. Cette maintenance est nécessaire pour prendre en compte de nouvelles conditions, non considérées et difficiles à prédire dans la phase de développement.

Depuis plusieurs années, le développement du logiciel s'oriente vers un nouveau paradigme permettant de simplifier la maintenance. Ce paradigme a pour objectif la construction d'applications en intégrant des briques logicielles bien définies et autonomes, appelées composants [HC01]. La notion classique de développement d'applications en écrivant du code a été remplacée par l'assemblage de composants préfabriqués. Les applications développées à base de ce paradigme ne sont plus des blocs monolithiques mais le regroupement de "pièces" bien définies. La nature composable des applications permet, en cas de besoin de maintenance, de repérer et de remplacer plus facilement les pièces concernées.

La réalisation des modifications requises pour la maintenance de l'application diffère selon le contexte et la nature de cette application. En général, pour introduire les modifications nécessaires, l'application doit être arrêtée, modifiée, recompilée puis démarrée à nouveau. Cependant, ce processus classique de maintenance ne peut pas être appliqué à certaines catégories d'applications, où l'interruption complète de service ne peut pas être tolérée. Ces applications, qualifiées de "non-stop", doivent être adaptées d'une manière dynamique avec le minimum de perturbation. Les points suivants illustrent quelques exemples d'applications, nécessitant une exécution continue et bénéficiant de l'adaptation dynamique :

- *Les applications à haute disponibilité*, comme les services de commerce électronique et les applications bancaires. L'arrêt de ce type d'application peut affecter la confiance des clients et peut avoir par conséquent un effet négatif sur les activités des exploitants de l'application.

- *Les applications utilisées dans des domaines critiques et temps-réel*, comme les applications de contrôle industriel et les services de télécommunication. La rupture du service téléphonique, par exemple, pendant plusieurs heures ne peut pas être tolérée, surtout si cette rupture se répète dans le temps [Fab76]. Les systèmes de télécommunication sont en général très complexes, et leur adaptation demande beaucoup de temps [SF93]. Il est nécessaire d'adapter dynamiquement ce type de systèmes pour minimiser la perturbation et garantir une qualité de service acceptable.
- *Les applications sensibles au contexte* [SAW94, Kor02]. Ces applications doivent être adaptées pour répondre aux changements affectant ce contexte [Som03]. La dynamicité se justifie si l'adaptation doit être réalisée fréquemment. Un système de diffusion de vidéo à la demande est un exemple type de cette catégorie d'applications. La vidéo est le média le plus gourmand en terme de bande passante du réseau. Le système de diffusion doit être capable de s'adapter en fonction des variations éventuelles de cette bande passante.
- *Les applications embarquées dans des équipements distants ou mobiles* [AM00]. On ne peut pas imaginer le déplacement d'un agent pour assurer la maintenance de ce type d'applications. Il est nécessaire de disposer de moyens permettant d'intervenir à distance sur l'application embarquée, pendant son exécution.
- *Les applications complexes*. En général, les applications de grande taille et surtout celles qui sont distribuées nécessitent beaucoup d'efforts pour qu'elles soient arrêtées, modifiées, reconstruites et correctement remises en service. Par conséquent, il est plus facile et souhaitable de réaliser les modifications nécessaires sans un arrêt complet de telles applications.

La maintenance d'une application à base de composants peut être réalisée en changeant sa configuration [Lit01]. Dans cette thèse, nous appelons *configuration* l'ensemble des composants qui constituent une application et les liens entre ces composants. Nous appelons *reconfiguration* toute opération dont le rôle est le changement de la configuration courante. Nous appelons, enfin, *reconfiguration dynamique*, une reconfiguration réalisée à l'exécution et assurant la cohérence et l'intégrité de l'application reconfigurée. La reconfiguration dynamique est un sujet qui a mobilisé une grande communauté de chercheurs depuis de longues années. Le travail que nous présentons dans cette thèse s'inscrit dans la lignée des travaux qui ont adressé ce sujet.

2 INTRODUCTION A LA PROBLEMATIQUE DE RECHERCHE

Pour favoriser la productivité et simplifier le développement, le programmeur ne doit se préoccuper que de la logique métier de son application. Les autres services, souvent qualifiés de non-fonctionnels, et nécessaires pour l'exécution de l'application, doivent être réalisés séparément par des spécialistes qualifiés dans des domaines spécifiques comme les

transactions, la sécurité et la persistance. La reconfiguration dynamique, qui fait l'objet de cette thèse, doit être traitée comme une propriété non-fonctionnelle. Elle doit être considérée d'une manière transparente afin de décharger le développeur le plus possible de la mise en œuvre des mécanismes liés à la reconfiguration. Cette vision coïncide avec d'autres approches [KM90, RC00, DC01] fondées sur la séparation entre les applications et le système responsable de leur reconfiguration dynamique.

Dans cette thèse, nous proposons une approche réflexive permettant de supporter la reconfiguration dynamique des applications à base de composants. Nous avons opté pour une approche réflexive du fait que la réflexion a prouvé depuis de longues années son intérêt dans la construction de systèmes flexibles et facilement adaptables. La thèse comporte une partie théorique analysant la problématique de reconfiguration dynamique d'une façon générale. Elle comporte aussi une partie expérimentale et pratique qui permet de tester et de mettre en œuvre les notions de base de la problématique étudiée.

De nombreux travaux existants, que nous avons étudiés dans cette thèse, ont traité la problématique de reconfiguration dynamique autour d'un modèle particulier de composants. Les solutions proposées sont par conséquent spécifiques au modèle traité. La reconfiguration dynamique d'une application dépend de son modèle sous-jacent, et de la plate-forme d'exécution liée à ce modèle. Ceci explique pourquoi les solutions développées sont spécifiques et figées à des modèles et des plate-formes d'exécution particulières. En passant d'un modèle à un autre, et même d'une plate-forme d'exécution à une autre, il est nécessaire de développer une nouvelle solution spécifique, même si la logique de reconfiguration est similaire. Ceci pose un sérieux problème de réutilisation.

Notre travail de thèse vise à développer une solution de reconfiguration plus générale, et indépendante le plus possible des applications à reconfigurer et de leurs modèles de composants. En arrivant à abstraire les éléments techniques et les concepts liés à chaque modèle, il devrait être possible de définir une approche qui favorise la réutilisation et qui peut être exploitée dans différents contextes. La séparation entre les applications à reconfigurer et le système responsable de leur reconfiguration permet de développer des applications propres, qui se focalisent sur la logique métier. La séparation favorise aussi l'évolution et facilite la maintenance, aussi bien des applications que du système de reconfiguration.

Notre approche est mise en œuvre sous forme d'un système baptisé DYVA. Le rôle de ce système est de prendre en charge la responsabilité de reconfiguration, et de permettre aux développeurs de se concentrer sur la logique applicative. Pour assurer l'ouverture et la réutilisabilité de notre système, nous l'avons basé sur un modèle abstrait de composants. Ce modèle incarne l'image de l'application à reconfigurer et permet de traiter d'une manière homogène, des applications issues de modèles de composants différents. Notre approche favorise aussi l'auto-reconfiguration qui signifie la capacité du système de reconfiguration à prendre des décisions de reconfiguration cohérentes, et à matérialiser ces décisions sans l'intervention d'un acteur humain. L'automatisation des décisions et de leur application

facilite considérablement l'utilisation du système de reconfiguration, et permet de décharger dans la mesure du possible les administrateurs des tâches d'observation, d'analyse et de décision.

3 PRINCIPES DE CONCEPTION ET DEMARCHE SUIVIE

Dans cette thèse, nous avons traité dans un premier temps la problématique de reconfiguration dynamique par rapport à des modèles de composants spécifiques. Nous avons plus particulièrement mené des expérimentations dans l'optique de mettre en œuvre des solutions de reconfiguration pour des modèles particuliers. Au lieu de définir un nouveau modèle de composants, et de le doter de capacités d'adaptation, nous avons préféré baser notre travail sur des modèles existants. Ce choix est justifié par le fait que notre intérêt, dans cette thèse, est porté sur la problématique de reconfiguration dynamique, et pas sur les modèles de composants en tant que tels. D'un côté, le choix d'un modèle existant nous permet de nous attaquer directement au fond du problème auquel nous nous intéressons. D'un autre côté, il nous force à raisonner sur les limitations du modèle choisi en terme d'adaptabilité, et sur les mécanismes nécessaires pour combler les limitations potentielles.

Notre première contribution pratique dans cette thèse était l'intégration d'un support de reconfiguration dynamique au modèle JavaBeans [BEAN97]. Ce modèle a été essentiellement choisi pour sa simplicité, et pour sa facilité d'utilisation. Nous avons ensuite réalisé un travail similaire pour le modèle OSGi [OSGI99]. Cette deuxième expérimentation sur un autre modèle avait pour objectif, d'un côté, la validation des idées de la première expérimentation, et d'un autre côté, l'exploration des différences entre les deux modèles.

Du point de vue de la mise en œuvre, la solution de reconfiguration pour le modèle OSGi a été entièrement développée (à partir de rien). Le principe de la solution, les algorithmes et les opérations de reconfiguration sont cependant pratiquement identiques dans les deux expérimentations. Nous n'avons exploité de l'expérimentation sur le modèle JavaBeans que les idées. Ceci n'est pas suffisant, et nous avons constaté qu'il était nécessaire de développer une solution plus générale, permettant plus de réutilisation afin d'éviter de réimplanter à chaque fois la même chose.

Comme premier pas, nous avons travaillé sur l'abstraction des éléments techniques et des concepts liés à différents modèles, afin de définir une approche qui favorise la réutilisation et qui puisse être exploitée dans différents contextes. Pour atteindre un degré acceptable de généricité, nous avons ensuite développé une solution séparée :

- ❑ des applications à développer : le programmeur ne doit s'occuper que de la logique métier de son application. Il doit être dégagé de la responsabilité de reconfiguration.
- ❑ des modèles visés : pour pouvoir appliquer la solution à plusieurs modèles cibles, il faut qu'elle soit indépendante des particularités de ces modèles.

Notre approche favorise aussi l'automatisation. Elle supporte, d'un coté, des formalismes pour décrire les règles qui permettent d'automatiser la reconfiguration, et d'un autre coté, les mécanismes nécessaires pour raisonner sur ces règles et prendre des décisions cohérentes.

4 ORGANISATION DU DOCUMENT

Ce document est organisé en deux grandes parties. La première comporte deux chapitres :

- Le chapitre 2 présente les principaux concepts, en relation avec le contexte scientifique dans lequel se déroule cette thèse. Il illustre d'abord les fondements du modèle de programmation par composants. Il discute ensuite la problématique de reconfiguration dynamique, et enfin, il présente les propriétés de base des systèmes réflexifs.
- Le chapitre 3 présente les différentes approches de reconfiguration dynamique que nous avons étudiées dans cette thèse. Chaque approche est illustrée par un exemple de système.

La deuxième partie présente notre contribution au domaine de la reconfiguration dynamique. Elle est constituée de quatre chapitres :

- Le chapitre 4 décrit deux expérimentations que nous avons menées après l'étude de l'état de l'art. L'objectif de la première était la mise en œuvre d'une solution de reconfiguration pour le modèle JavaBeans. La deuxième, quant-à-elle, adresse le modèle OSGi. Ce chapitre met aussi en évidence les différences et les aspects communs entre les deux solutions mises en œuvre. Enfin, il présente une synthèse qui illustre, d'un coté, les limitations des deux solutions, et d'un autre coté, les besoins à satisfaire pour développer une solution plus générale.
- Le chapitre 5 illustre la solution de reconfiguration dynamique que nous avons développée dans le cadre de cette thèse. Cette solution, implantée dans le système DYVA, constitue un aboutissement et une généralisation des différentes expérimentations que nous avons menées. Notre solution générale se présente sous forme d'une machine virtuelle qui prend en charge la responsabilité de reconfiguration, et qui doit être personnalisée pour être exploitée pour un modèle particulier. Les motivations derrière notre approche, l'architecture logique de notre machine et les concepts sur lesquels elle est basée, sont expliqués dans ce chapitre.
- Le chapitre 6 décrit l'implémentation de notre système de reconfiguration. Il illustre la mise en œuvre des différents éléments de l'architecture, et présente les outils que nous avons développés comme support à notre approche.

- Le chapitre 7 traite de la personnalisation de notre système de reconfiguration pour le modèle OSGi. Nous illustrons, dans un premier temps, la création des interfaces de notification et de reconfiguration, spécifiques au modèle OSGi, en fonction des interfaces exposées par DYVA. Nous donnons ensuite des exemples de mise en œuvre des plugins de projection. Ces plugins sont responsables de l'exécution effective des opérations de reconfiguration sur les applications OSGi. Ce chapitre illustre aussi un portail Web, une des applications que nous avons développées en OSGi, dans le but de valider la personnalisation.

CHAPITRE 2

CONTEXTE SCIENTIFIQUE

PRÉAMBULE.

Avant de présenter les travaux autour de la reconfiguration dynamique des applications, nous présentons dans ce chapitre les principaux concepts, en relation avec le contexte scientifique dans lequel se déroule cette thèse. Nous présentons d'abord les fondements du modèle de programmation par composants. Nous discutons ensuite la problématique de reconfiguration dynamique, et avant de conclure ce premier chapitre, nous présentons les propriétés de base des systèmes réflexifs.

1 INTRODUCTION

Ce chapitre présente les principaux concepts liés à la problématique de reconfiguration dynamique que nous traitons dans cette thèse. Notre travail vise en particulier les applications à base de composants. Nous discutons par conséquent en premier lieu le modèle de programmation par composants et nous présentons quelques modèles actuels issus de l'industrie.

La deuxième section de ce chapitre présente la problématique de reconfiguration dynamique. Nous présentons d'abord une taxonomie des différents types de modifications, pouvant être appliquées à une application à base de composants. Les approches de reconfiguration dynamique diffèrent selon plusieurs critères comme la performance, la simplicité d'utilisation et la cohérence. Ces critères, permettant d'évaluer et de comparer les différentes approches, sont discutés à la fin de la deuxième section.

Enfin et avant de conclure ce chapitre, nous présentons le concept de réflexion, un moyen efficace pour le développement de systèmes flexibles et facilement adaptables.

2 LE MODELE DE PROGRAMMATION PAR COMPOSANTS

Le développement orienté composant [HC01] a pour objectif la construction d'applications en intégrant des briques logicielles bien définies et autonomes, appelées composants. La notion classique de développer des applications en écrivant du code a été

remplacée par l'assemblage de composants préfabriqués. Plusieurs avantages d'un tel modèle de programmation peuvent être cités :

- ❑ Les applications développées ne sont plus des blocs monolithiques mais le regroupement de "pièces" bien définies.
- ❑ Le fait que les frontières entre les différentes pièces soient bien spécifiées simplifie l'extension et la maintenance des applications. Grâce à la nature composable de l'application, et en cas de besoin de maintenance, il est plus facile de repérer et de remplacer les pièces concernées. Des mécanismes de surveillance sont souvent nécessaires pour permettre le repérage des composants qui doivent être remplacés.
- ❑ Le développement peut se faire d'une manière incrémentale en fonction des capacités de développement et selon les besoins des utilisateurs. Une application partiellement définie peut être exploitée dans l'état sans qu'elle soit obligatoirement complète. Ceci est particulièrement important pour les applications de grande taille.
- ❑ La collaboration est plus intuitive et la répartition des compétences est meilleure : des programmeurs bien spécialisés développent les composants de base et un assembleur se charge d'intégrer ces composants pour former l'application. Ce dernier manipule ainsi les composants comme des boîtes noires respectant une spécification précise sans se soucier de leurs détails internes.
- ❑ La réutilisation de composants préfabriqués, achetés sur étagère comme les COTS [Gen97, Voa98]. Ces composants sont développés indépendamment et destinés à être utilisés dans la construction de différents systèmes.

2.1 Notion de composant

Plusieurs définitions du concept de composant peuvent être trouvées dans la littérature. Ces définitions diffèrent naturellement en fonction du domaine d'application et en fonction de la compréhension qui peut être attachée aux composants. Dans le domaine de la réutilisation du logiciel par exemple, un composant est vu comme n'importe quelle entité qui peut être réutilisée. Les architectes du logiciel voient les composants comme des éléments de conception. Les déployeurs les considèrent comme des unités de déploiement. Les utilisateurs des applications patrimoniales les confondent avec les éléments patrimoniaux qu'ils manipulent. Dans d'autres considérations, un composant peut être également vu comme un élément d'un projet, comme une configuration ou même comme une documentation.

Du fait de l'absence d'une définition commune des composants, nous avons choisi de citer, dans cette section, deux définitions importantes en soulignant les points les plus pertinents derrière chacune d'elles.

Définition 1 :

"Un composant logiciel est une unité de composition avec des interfaces contractuellement spécifiées et des dépendances explicites au contexte. Un composant logiciel peut être déployé indépendamment et est sujet à une tierce composition"

Clemens Szyperski [Szy02]

Dans cette première définition trois points clés peuvent être identifiés :

- Un composant est une unité possédant des interfaces bien définies et ayant des dépendances exprimées explicitement. Les interfaces spécifient les fonctionnalités fournies par le composant, et pour assurer ces fonctionnalités, certaines dépendances vers d'autres composants ou même vers l'environnement doivent être résolues.
- Un composant peut être déployé d'une manière indépendante. Il n'est pas nécessaire de l'intégrer dans une application pour pouvoir le déployer.
- Un composant peut être assemblé dans une application par un tiers. Ceci signifie que le composant peut exister indépendamment d'une architecture particulière, et que l'aspect architectural qui permet d'intégrer des composants pour former une application n'est pas de la responsabilité du composant.

Définition 2 :

"Un composant est une unité non triviale, presque indépendante, et une partie remplaçable d'un système qui accomplit une fonction claire dans le contexte d'une architecture bien définie. Un composant se conforme à un ensemble d'interfaces et fournit leur réalisation physique"

Philippe Kruchten [Kru98]

Cette deuxième définition met en évidence d'autres aspects importants qui caractérisent les composants. Ces aspects complémentaires de ceux évoqués par la première définition peuvent être résumés dans les points suivants :

- Un composant n'est pas aussi trivial qu'une ligne de code ou une simple classe. Il est à noter que cette vision ne s'applique pas à certains modèles où les composants peuvent être de granularité très variée allant d'un simple bouton à des composants de gros grain.
- Un composant n'est pas entièrement indépendant, mais travaille en collaboration avec d'autres composants pour fournir les fonctionnalités requises.
- Un composant remplit un contrat bien précis en réalisant l'ensemble d'interfaces formant le contrat.
- Un composant a une cohésion logique et physique; il remplit un rôle dans l'application et ne peut être qu'un simple regroupement arbitraire.

- ❑ Un point très important concerne la remplaçabilité des composants. Le fait de raisonner avec une séparation claire et explicite entre les composants formant une application permet de substituer plus facilement un composant par un autre remplissant le même contrat. Ceci constitue un aspect central pour supporter l'évolution et l'adaptation des applications.

On pourrait remarquer que l'aspect architectural n'est pas considéré de la même manière dans les deux définitions. Dans la première, nous avons expliqué qu'un composant doit être indépendant de l'architecture dans laquelle il doit être utilisé, tandis que dans la deuxième définition, il est énoncé qu'un composant accomplit une fonction claire dans le contexte d'une architecture bien définie. Cette différence de point de vue peut s'expliquer par le fait que dans le premier cas, on parle plutôt de l'indépendance du composant dans la phase de développement, tandis que dans le deuxième cas, on parle de la dépendance d'une architecture dans la phase d'utilisation.

Ce que nous pouvons retenir des points précédents c'est que le développement orienté composant favorise la réutilisation, l'évolution et l'adaptation des applications. Ce sont justement ces derniers défis que le développement orienté objet n'a pas pu relever. Par exemple, supposons que nous ayons une classe qui assure, entre autres, une certaine fonctionnalité que nous souhaitons réutiliser dans une nouvelle application. La façon la plus "directe" de procéder est de copier la partie du code concernant cette fonctionnalité, ensuite de compiler la nouvelle application. Cette compilation a de fortes chances d'échouer si la partie du code copiée a des dépendances vers d'autres classes. Nous devons aussi copier ces classes, qui à leur tour, peuvent avoir des dépendances vers d'autres classes, et ainsi de suite jusqu'à recopier toute l'application originale, même si le code que nous souhaitons réutiliser est minimal. Dans un développement orienté composant, les frontières et les dépendances entre les composants sont bien définies. Elles permettent de réutiliser les composants comme des boîtes noires, ce qui évite la duplication de code et simplifie leur remplacement.

Dans la suite, nous regardons de plus près quelques modèles de composants actuels. Pour chaque modèle nous nous intéressons plus particulièrement aux trois aspects suivants :

- ❑ La création des composants : nous présentons d'abord la vision externe d'un composant, ce qui reflète ses frontières avec le monde extérieur. Nous discutons ensuite sa vision interne c'est à dire comment il est développé.
- ❑ L'assemblage des composants : nous montrons comment les composants sont assemblés pour former des applications. Ce deuxième aspect reflète la vision architecturale des applications, ce qui est indispensable pour pouvoir les reconfigurer. Il est à noter que, dans cette phase, nous nous intéressons à l'assemblage statique des composants avant la phase de déploiement et d'exécution.
- ❑ Le déploiement et l'exécution des composants : nous discutons l'instanciation des composants à l'exécution, leurs besoins pour pouvoir s'exécuter et leur capacité

naturelle à supporter la reconfiguration dynamique. Naturellement, les instances des composants sont interconnectées après leur instanciation selon les informations de la phase d'assemblage (si elle existe), ou simplement selon leur implémentation.

Ces trois aspects, que nous discutons dans la suite, sont étroitement liés à la reconfiguration dynamique. D'abord, les frontières entre les composants permettent d'identifier et d'isoler les entités à reconfigurer. Leur implémentation, même si elle est normalement cachée, peut aussi avoir un rôle dans le tracé des frontières. Puis, la vision architecturale définit les liens concrets entre les composants à un moment donné. Et enfin, la vision d'exécution est importante dans la mesure où la reconfiguration a lieu durant cette phase.

2.2 Le modèle JavaBeans

JavaBeans [BEAN97] est un modèle de composants introduit par Sun Microsystems en 1996. Il est basé sur le langage Java [JAVA] et permet de créer de manière visuelle, en utilisant des outils d'assemblage graphiques, des applications centralisées. Le modèle JavaBeans est basé sur un ensemble de conventions d'écriture du code Java. Ces conventions d'écriture permettent aux outils de composition, en utilisant l'introspection de Java [Bou00], de déterminer les caractéristiques des composants pour pouvoir les personnaliser et les utiliser pour former des applications. Comme alternative aux règles d'écriture, le développeur peut fournir un descripteur, appelé le *"BeanInfo"*, qui exprime explicitement les caractéristiques des composants. L'avantage d'un descripteur explicite, séparé du code des composants, est de pouvoir reformuler comme JavaBeans, du code Java déjà existant sous une forme binaire, et ne respectant pas les conventions lexicales énoncées par le modèle. L'utilisation du *BeanInfo* permet aussi d'écrire des applications en JavaBeans, même en ignorant les règles lexicales énoncées par le modèle.

2.2.1 Construction des composants

Les Beans peuvent être de différentes granularités, allant d'un simple bouton à des composants complexes. Malgré que le modèle vise en particulier les applications graphiques, il est possible de développer des Beans invisibles, qui servent par exemple comme des ressources partagées, utilisées dans des applications graphiques ou non.

La Figure 1 montre la vision logique d'un Bean. L'interface du Bean est constituée des éléments suivants :

- Les événements qu'il peut émettre ou qu'il peut recevoir.
- Les méthodes publiques qu'il implémente, et qui peuvent être appelées par les autres Beans.

- Les propriétés qui permettent de personnaliser le Bean pour l'utiliser dans un contexte particulier.

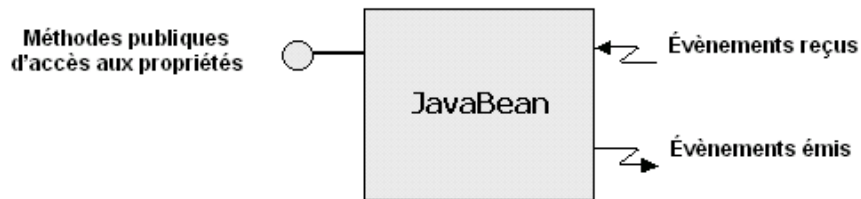


Figure 1. Vision logique d'un JavaBean

L'implémentation du Bean n'a rien de différent par rapport à la programmation Java standard car, comme nous l'avons énoncé précédemment, le modèle n'est rien d'autre qu'un ensemble de conventions lexicales. D'une part, le développement d'un Bean ne nécessite aucun concept ou élément syntaxique nouveau, et d'autre part, son utilisation ne requiert aucune bibliothèque ni extension particulière. Toute classe Java peut alors être considérée comme un Bean, la seule contrainte à respecter est d'avoir un constructeur publique sans arguments. En effet, les outils d'assemblage graphiques utilisent la méthode `newInstance()`, définie dans la classe `java.lang.Class` de l'API Java. Cette méthode ne peut être appelée sur la classe du Bean, pour créer une instance, que si cette classe définit un constructeur publique sans arguments. En relation avec l'architecture JavaBeans, deux packages ont été ajoutés à Java, fournissant des services pour simplifier le développement et l'utilisation des Beans (`java.beans` et `java.beans.beancontext`). L'utilisation de ces deux packages n'est pas obligatoire.

2.2.2 Assemblage des composants

La communication entre les Beans est assurée par le mécanisme des événements de Java. Un Bean *A* peut être assemblé avec un autre Bean *B* en déclarant *B* comme auditeur d'un ou de plusieurs événements dont *A* est la source. La Figure 2 montre le principe de communication événementielle de Java.

Dans la pratique, le Bean intéressé par un certain type d'événements, doit implémenter les méthodes nécessaires par lesquelles les événements de ce type peuvent être notifiés. Le Bean censé notifier l'événement, doit implémenter deux méthodes pour ajouter et supprimer les auditeurs potentiels. La notification de l'événement consiste d'abord à créer un objet représentant l'événement, et ensuite à appeler une méthode correspondante de l'auditeur avec l'objet événement comme argument.



Figure 2. Communication événementielle de Java

La composition d'applications à base de JavaBeans peut se faire, d'un côté, par programmation, en développant la glue qui assemble les Beans entre eux et qui permet de lancer l'application. D'un autre côté, il est possible d'utiliser des outils d'assemblage graphiques, qui permettent de manipuler graphiquement les Beans, de les personnaliser et de les interconnecter pour obtenir des applications. Parmi ces outils nous pouvons citer la BeanBox [BBOX], le BeanBuilder [BBUI] et l'environnement NetBeans [NETB].

2.2.3 Déploiement, exécution et aspects dynamiques

Les Beans sont délivrés sous forme de fichiers Jar aux applications et aux outils qui sont censés les manipuler. Un Jar contient principalement les classes d'un ou de plusieurs Beans et les ressources dont ils ont besoin. Le manifeste du Jar contient des méta-données décrivant les propriétés des Beans comme leurs noms et leurs dépendances.

Comme le développement des Beans, leur exécution ne requiert aucun framework spécifique. Une application en JavaBeans se lance et se présente en exécution comme toute autre application Java. Deux différences importantes, dans notre contexte d'évolution dynamique, méritent d'être soulignées :

- ❑ Les Beans sont développés en théorie indépendamment les uns des autres, et sont composés par un élément tiers pour former des applications. Ceci est possible car les Beans définissent explicitement des méthodes d'assemblage qui peuvent être appelées par un tiers. Il est par conséquent possible qu'un tiers puisse agir, même en cours d'exécution, sur un assemblage de Beans et changer ainsi son architecture.
- ❑ Un Bean définit explicitement un ensemble de propriétés permettant de le personnaliser. Ces propriétés, représentant en quelque sorte l'état du Bean, ont obligatoirement un couple de méthodes publiques permettant de les consulter et de les modifier. Nous pouvons ainsi dire qu'un Bean possède un état explicite, consultable et modifiable de l'extérieur en cours d'exécution.

2.3 Le modèle Entreprise JavaBeans

Les EJB (Entreprise JavaBeans) [EJB, AJ02] sont un élément clé de la plate-forme J2EE (Java 2 Enterprise Édition) [J2EE]. Ils sont basés sur une architecture de type client-serveur.

Plus particulièrement, un EJB est un composant logiciel écrit en Java, et s'exécutant du côté serveur. Comme le montre la Figure 3, la norme EJB est essentiellement orientée vers le développement d'applications offrant des services complémentaires au dessus d'un système d'information d'entreprise. La couche *client* comporte les applications (applets [Rod00], applications Java, pages HTML...) qui permettent aux utilisateurs d'interagir avec les données de l'entreprise. La couche *données* concerne les bases de données de l'entreprise. Enfin, la couche *serveur* joue le rôle d'intermédiaire entre les deux autres couches. Elle englobe un serveur Web incluant les JSP (JAVA server page) [Pat00], les servlets JAVA [Pat00], et un serveur de composants métiers lié à des conteneurs EJB.

La spécification EJB inclut un ensemble de conventions, un ensemble de classes et d'interfaces constituant l'interface de programmation des composants.

2.3.1 Construction des composants

Un EJB se distingue par deux interfaces spécifiques et une classe Java qui représente son implémentation effective. La première interface, de type "`javax.ejb.EJBObject`", définit l'ensemble des fonctionnalités que les clients de l'EJB peuvent appeler. L'autre interface est de type "`javax.ejb.EJBHome`", elle définit un ensemble de méthodes permettant de gérer le cycle de vie des instances de l'EJB (création, destruction, recherche...). Un client d'un EJB peut être n'importe quel programme pouvant interagir avec le protocole de communication supporté par le serveur EJB, et peut notamment être un autre EJB.

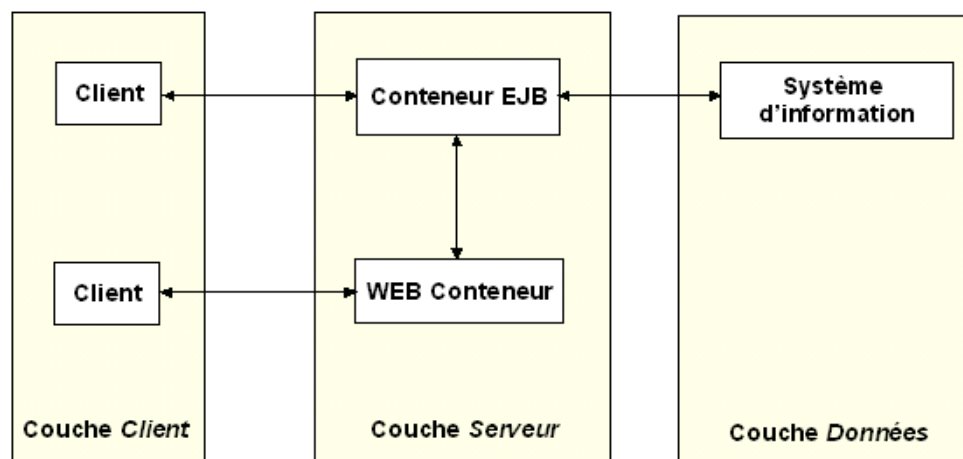


Figure 3. Architecture multi-niveaux

Trois types de composants peuvent être définis :

- Les EJB session : représentent la logique applicative c'est à dire l'ensemble des fonctionnalités fournies aux clients. Ils peuvent être avec ou sans état.
 - Sans état : utilisés pour traiter les requêtes de plusieurs clients.
 - Avec état : conservent un état entre les appels. Un EJB de ce type est associé à un seul client et ne peut être partagé.
- Les EJB entité : représentent les objets persistants de l'entreprise, stockés dans une base de données. Chaque enregistrement de la base de données, est chargé dans un EJB entité pour qu'il puisse être manipulé.
- Les EJB orientés message : introduits dans la spécification 2.0, ils permettent de traiter les requêtes d'une façon asynchrone.

Les instances de composants EJB sont destinées à être exécutées dans des *conteneurs* qui les entourent et qui prennent en charge leur gestion (cf. 2.3.3). L'état d'une instance d'un composant peut être géré par l'instance elle-même ou sa gestion peut être déléguée à son conteneur. Dans ce dernier cas, le composant doit déclarer explicitement les différents attributs qui constituent son état.

2.3.2 Assemblage des composants

Les composants EJB sont fournis par le développeur sous forme de fichiers Jar contenant typiquement l'ensemble des interfaces et des classes d'un ou de plusieurs composants et un descripteur de déploiement. La phase d'assemblage dans le modèle EJB consiste à regrouper plusieurs composants EJB de base. Le résultat de l'assemblage est un module EJB qui représente une unité de déploiement de plus gros grain. Un composant fourni par un développeur n'est pas censé contenir certaines informations systèmes comme les transactions, la sécurité et la distribution. Un module fourni par l'assembleur contient un ou plusieurs composants avec les instructions qui décrivent comment ces composants sont combinés dans une même unité de déploiement, comment ils doivent être déployés et les informations système non fournies par le développeur.

2.3.3 Déploiement, exécution et aspects dynamiques

Les modules EJB sont livrés sous forme de fichiers contenant, en plus des composants, un ensemble de descripteurs. Ces descripteurs contiennent certaines informations relatives aux composants et décrivent la façon de les gérer à l'exécution. Le déploiement d'un composant EJB consiste à injecter le composant effectif, contenu dans l'unité de déploiement, dans un environnement d'exécution spécifique.

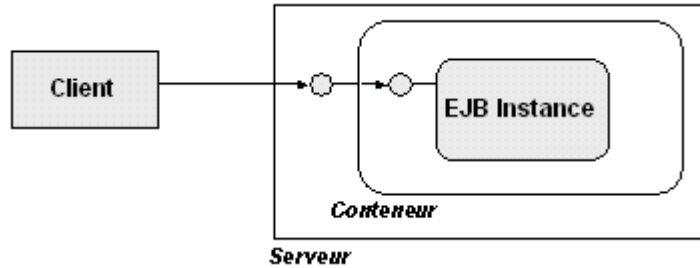


Figure 4. Exécution des composants EJB

Les instances de composants EJB s'exécutent dans des conteneurs, qui eux mêmes, sont logés dans un serveur. Ceci est illustré par la Figure 4. Un conteneur est tout simplement un médiateur qui isole les instances de composants. Il a aussi la responsabilité de gérer leur cycle de vie (création, destruction, activation...), et de leur fournir les services systèmes dont ces instances ont besoin.

2.4 Le modèle OSGi

OSGi (Open Services Gateway initiative) [OSGI99] est un modèle de composants proposé en 1999 par un groupe d'entreprises partageant des problématiques communes. Le modèle vise en particulier les environnements avec des contraintes de ressources comme les passerelles résidentielles et véhiculaires. Il est basé sur Java. Parmi les implémentations du modèle OSGi, nous pouvons citer :

- OSCAR [OSCAR] : développée au sein de l'équipe Adèle du laboratoire LSR (implémentation source-libre).
- JES [JES] : développée par Sun Microsystems (implémentation commerciale).
- SMF [SMF] : développée par IBM (implémentation commerciale).

La spécification OSGi introduit le concept de *bundle*, un Jar contenant un ensemble de classes Java et de ressources. Il est important de noter qu'un bundle représente l'unité de déploiement associée au modèle OSGi, et sert à conditionner les composants, qui, à leur tour, sont appelés services dans la spécification. La Figure 5 présente la structure d'un bundle OSGi.

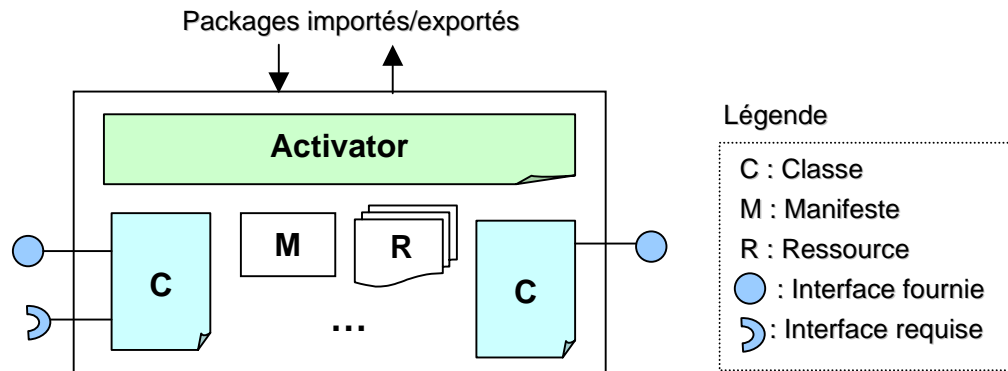


Figure 5. Structure d'un bundle OSGi

Comme illustré dans la figure précédente, un bundle est concrètement constitué d'un ensemble de classes et d'interfaces Java, d'un manifeste, d'un ensemble de ressources (pages HTML, images...) et éventuellement d'une classe spécifique appelée *activateur*.

2.4.1 Construction des composants

Nous précisons que ce que nous appelons composant OSGi dans cette thèse, correspond à ce qui est nommé service dans la spécification OSGi. Plus particulièrement, un composant est une classe Java qui implémente une ou plusieurs interfaces sous lesquelles le composant est connu. La définition des interfaces du composant ou de sa classe d'implémentation est identique au développement Java ordinaire.

Chaque bundle (unité de déploiement) possède en plus des composants qu'il exporte, un manifeste contenant un ensemble de propriétés le décrivant. Un bundle peut également contenir une classe spécifique appelée *activateur*. Cette classe définit deux méthodes "*start*" et "*stop*" appelées respectivement au démarrage et à l'arrêt des bundles. La méthode *start* peut être utilisée pour enregistrer les différents composants inclus dans le bundle et aussi localiser et obtenir les références vers les composants requis.

Il est possible de décrire dans le manifeste du bundle l'ensemble des services fournis et requis par les composants qu'il contient. Cependant ces informations, nécessaires pour la reconfiguration dynamique, sont facultatives et ne sont mentionnées qu'à titre informatif. Nous pouvons constater donc que d'une manière générale, la description des fonctionnalités fournies et les besoins en termes de fonctionnalités requises par un composant donné, ne sont pas déclarés explicitement, et sont cachés dans l'implémentation. En effet, les fonctionnalités fournies se traduisent en général par les interfaces implémentées par le composant. Les fonctionnalités requises, d'un autre côté, correspondent aux services que le composant

référence dans son code. Il n'y a pas une description externe qui permet de connaître les fonctionnalités fournies et requises, sans être obligé de les chercher dans le code.

2.4.2 Assemblage des composants

La phase d'assemblage n'apparaît pas comme une étape explicite dans le développement des applications OSGi. La notion même d'application n'existe pas en tant que telle. Elle se construit d'une manière incrémentale au fur et à mesure que les composants apparaissent ou disparaissent. En d'autres termes, le concept d'application n'apparaît que dans la phase d'exécution et évolue dans le temps.

2.4.3 Déploiement, exécution et aspects dynamiques

Le déploiement et l'exécution des composants OSGi se font au dessus du *framework* OSGi, qui lui, s'exécute au dessus d'une machine virtuelle Java. Le rôle du framework est d'offrir un certain nombre de fonctionnalités de base et de gérer le cycle de vie des bundles (installation, démarrage, arrêt, mise à jour et désinstallation).

L'une des innovations d'OSGi en matière de dynamicité est de supporter la construction incrémentale des applications. Ceci est possible car les composants peuvent arriver et disparaître d'une façon arbitraire. Pour pouvoir exploiter cette caractéristique, les composants doivent être conscients de cette nature volatile et doivent se préparer pour gérer de telles situations. Le développeur doit donc penser à toutes les situations qui peuvent se présenter en exécution. La Figure 6 présente les différentes étapes du cycle de vie d'un bundle.

Cycle de vie des bundles

Le cycle de vie d'un bundle est illustré par la Figure 6. Les différents états formant ce cycle sont expliqués ci-dessous :

- *Installé* : le bundle est ajouté au framework OSGi avec succès.
- *Résolu* : toutes les classes Java et tout le code natif requis par le bundle sont disponibles. Ceci signifie que le bundle peut être démarré.
- *En activation* : le bundle est en train de démarrer. Ceci signifie que la méthode `"start ()"` de l'activateur a été appelée mais n'a pas encore terminé son exécution.
- *En arrêt* : le bundle est en train de s'arrêter. Ceci signifie que la méthode `"stop ()"` de l'activateur a été appelée mais n'a pas encore terminé son exécution.
- *Actif* : le bundle est démarré avec succès et tous les services qu'il requiert sont disponibles (toutes ses dépendances sont résolues).

- *Désinstallé* : le bundle est désinstallé et ne peut pas passer à un autre état.

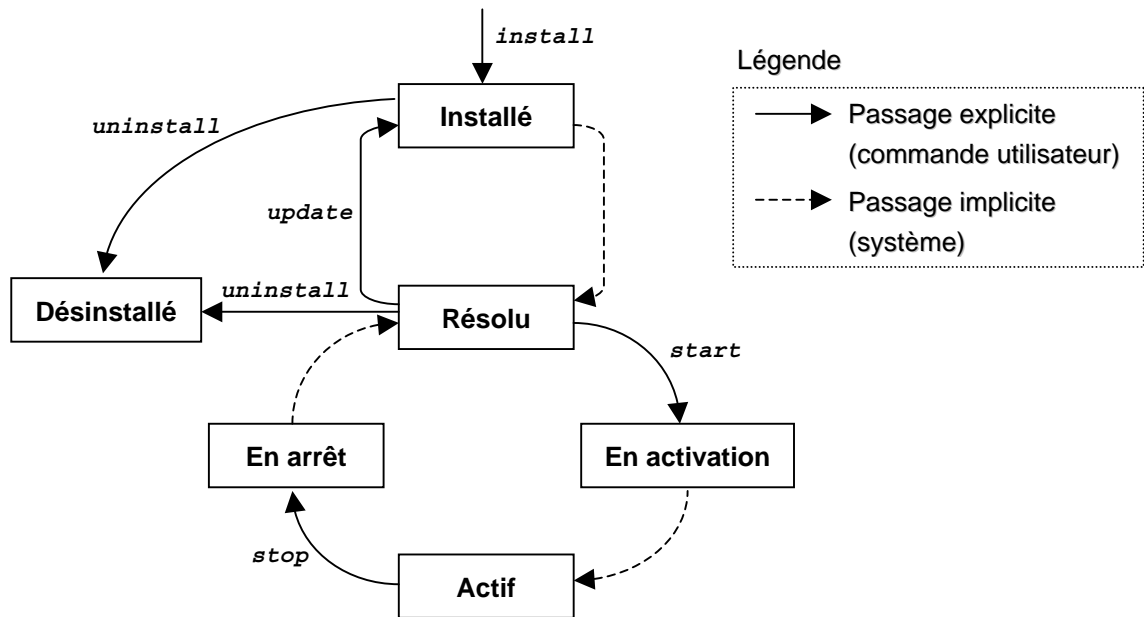


Figure 6. Cycle de vie d'un bundle OSGi

Dans le cycle de vie illustré par la Figure 6, deux types de transition d'un état à un autre peuvent être distingués :

- ❑ Les transitions explicites : elles résultent de l'application des commandes de l'utilisateur. Par exemple, en appliquant la commande "*stop monBundle*", *monBundle* passe de l'état "*actif*" à l'état "*en arrêt*".
- ❑ Les transitions implicites : elles sont réalisées automatiquement par le framework OSGi. Par exemple, lorsqu'un bundle est dans l'état "*en arrêt*", le framework se charge de le passer explicitement dans l'état "*résolu*".

Mécanisme des événements

Le framework OSGi est capable de notifier des événements aux instances de composants qui se déclarent explicitement intéressées. Ces événements permettent de suivre l'évolution de l'application et de réagir aux changements qui se produisent. Trois classes d'événement sont supportés par le framework OSGi :

- ❑ Les événements du framework : concernent le démarrage et les erreurs du framework. Le composant intéressé doit implémenter l'interface "*FrameworkListener*".

- ❑ Les événements des bundles : concernent le changement du cycle de vie des bundles. Le composant intéressé doit implémenter l'interface *"BundleListener"*.
- ❑ Les événements des services : concernent l'enregistrement et le dés-enregistrement des services. Le composant intéressé doit implémenter l'interface *"ServiceListener"*.

Le mécanisme des événements est fondamental dans OSGi. Il permet de développer des applications capables de répondre et de suivre leur contexte d'utilisation.

Instanciation des composants

Par défaut, un composant n'est instancié qu'une seule fois. Cette instance est partagée par toutes les autres instances de composants qui en ont besoin. Pour changer ce fonctionnement par défaut, et pour pouvoir contrôler le nombre d'instances à créer, le composant doit définir une fabrique [GH+95] en implémentant l'interface *"ServiceFactory"*.

Plusieurs améliorations ont été introduites dans les nouvelles versions de la spécification OSGi (en l'occurrence la versions 3). En particulier un nouveau concept appelé le *ServiceTracker* a été introduit. C'est un ensemble de classes utilitaires qui facilitent le suivi de l'apparition et de la disparition des services. Un travail visant la gestion automatique des services OSGi a été également proposé dans [Cer04].

2.5 Le modèle COM

COM (Component Object Model) [COM95] est un modèle de composants développé par Microsoft en 1995. Son objectif est de permettre le développement d'applications en assemblant des composants pouvant être écrits en différents langages, et communiquant en COM. Ces composants, pour qu'ils puissent interagir, doivent adhérer à une structure binaire commune, spécifiée par Microsoft. Ceci signifie que les compilateurs des différents langages de programmation, génèrent les composants suivant le même format binaire. Un composant COM, écrit dans un langage donné, peut ainsi interagir avec un autre composant, même s'il est écrit dans un langage différent.

Bien que COM (et sa version distribuée DCOM [Tha99]) représente un modèle d'interaction de bas-niveau (binaire), d'autres technologies de plus haut niveau ont été bâties par Microsoft au dessus de COM. Parmi ces technologies nous pouvons citer OLE [Cha96] qui permet aux applications de partager et d'échanger des données, ActiveX [Cha96] qui permet l'intégration des composants dans des sites Web, et COM+ [Kir97] qui étend les capacités de COM avec des services système comme les transactions et la sécurité pour mieux supporter les systèmes d'information des entreprises. Dans cette section, nous nous intéressons uniquement aux fondements de base du modèle COM indépendamment de ses extensions.

2.5.1 Construction des composants

COM est indépendant d'un langage de programmation particulier, le développement des composants peut se faire en n'importe quel langage capable de générer du code binaire en format standard de COM comme C [Ker00], C++ [Str97], Pascal [DDW03], Ada [Ala95], Smalltalk [SB96], Visual Basic [MC99] et autres. Concrètement, un composant COM est une classe implémentant une ou plusieurs interfaces (classes abstraites). Même si une instance d'un composant COM se présente en mémoire comme un objet C++, les clients ne peuvent utiliser cette instance qu'à travers ses interfaces (les détails d'implémentation ne sont pas visibles). La Figure 7 illustre une vue typique d'un composant COM.

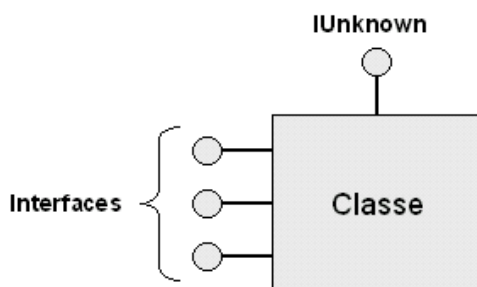


Figure 7. Vue d'un composant COM

Une interface d'un composant possède un identificateur unique (GUID) avec lequel elle est connue dans le système. Toute interface doit avoir comme base une interface spéciale appelée "*IUnknown*" qui doit aussi être implémentée par la classe du composant. Cette interface définit deux fonctions pour gérer le cycle de vie des instances de composants, "*AddRef*" qui permet d'incrémenter un compteur représentant le nombre de références vers l'instance du composant, et "*Release*" qui permet de décrémenter ce compteur. La troisième fonction définie dans l'interface *IUnknown* s'appelle "*QueryInterface*", elle permet aux clients d'un composant de naviguer entre les différentes interfaces que ce composant supporte. Elle permet aussi de déterminer dynamiquement le comportement supporté par le composant. Contrairement à la programmation en C++, les clients n'instancient jamais directement les composants dont ils ont besoin (avec un `new()`) car ils ne sont pas censés connaître les détails d'implémentation. Nous verrons ci-dessous comment se fait l'instanciation.

2.5.2 Assemblage des composants

Un composant COM peut être une entité indépendante déployée indépendamment de ses clients et des autres composants dont elle a besoin. D'un autre côté, les composants COM peuvent être intégrés dans des applications et déployés par les programmes d'installation de ces applications. Dans ce deuxième cas, l'application peut être vue comme le résultat

d'assemblage d'un ensemble de composants (et d'autres objets qui ne sont pas forcément des composants). Comme le montre la Figure 8, tout composant de l'application doit être utilisé à travers ses interfaces que ce soit par les objets de l'application ou par les autres composants.

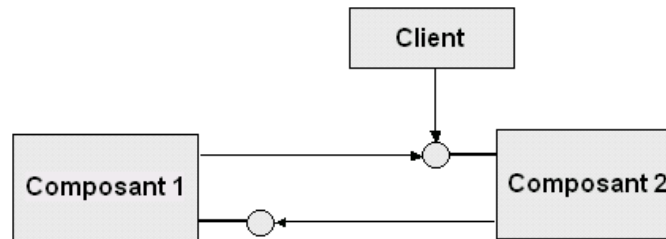


Figure 8. Assemblage de composants COM

2.5.3 Déploiement, exécution et aspects dynamiques

Les composants COM se présentent concrètement sous forme de fichiers exécutables (.EXE) ou sous forme de bibliothèques dynamiques (.DLL). Leur déploiement se traduit par leur installation dans le registre Windows, et leur exécution nécessite alors forcément ce système. L'instanciation d'un certain composant est gérée par ce composant lui-même (le composant inclut une *factory* qui gère ses instances). Quand un client (instance d'un composant ou autre) a besoin d'une instance d'un composant, il s'adresse au système COM qui se charge alors de chercher le composant en question, de lui demander éventuellement la création d'une instance, ensuite de la retourner au client qui peut alors communiquer directement avec cette instance.

Du point de vue évolution et adaptation dynamiques, le modèle COM profite des avantages de la liaison dynamique supportée par les DLLs. Cet aspect augmente les capacités du modèle en permettant d'ajouter, de retirer et de remplacer dynamiquement les composants.

2.6 Synthèse

Dans cette section, nous avons abordé le modèle de programmation par composants. Nous avons commencé par présenter les concepts fondamentaux sur lesquels ce modèle est basé, ensuite nous avons brièvement décrit quelques modèles de composants industriels en insistant sur trois aspects principaux : la construction des composants, leur assemblage en applications et enfin leur déploiement, leur exécution et leurs aspects dynamiques. Nous pouvons conclure après l'étude de ces modèles que les concepts de base, dégagés par les définitions que nous avons analysées, ne sont pas toujours présents. En particulier, les contrats des composants ne sont pas toujours définis d'une façon explicite. Comme dans le cas du modèle OSGi, les services requis, et les services fournis, sont cachés dans l'implémentation

des composants. Aucune description externe n'est fournie. Ceci rend difficile la possibilité d'assembler les composants par un tiers et complique et diminue fortement leur capacité de substitution.

Même si le déploiement est adressé par quelques modèles de façon explicite voire implicite, les problèmes d'évolution et d'adaptation dynamique ne sont jamais véritablement traités. En général, la plupart des modèles de composants ne définissent pas les besoins et les règles à respecter pour que les applications basées sur ces modèles puissent évoluer dynamiquement et puissent être adaptées au cours de leurs exécutions. Ces modèles ne définissent pas non plus les mécanismes et les moyens nécessaires pour pouvoir agir sur les applications en exécution dans le but de les adapter. Une vision conceptuelle intéressante du développement orienté composant a été proposée dans [Vil03]. Cette vision est accompagnée de l'analyse de plusieurs modèles de composants industriels.

3 RECONFIGURATION DYNAMIQUE

Nous avons présenté dans le premier chapitre de cette thèse nos motivations pour la reconfiguration dynamique. Nous avons principalement expliqué l'importance et la nécessité de la dynamique pour les applications qui doivent être hautement disponibles : les applications critiques, de grande taille et distribuées, pour les applications ayant un environnement d'exécution fréquemment variable, et aussi les applications embarquées dans des équipements mobiles. Dans cette section, nous discutons d'abord les raisons naturelles qui poussent à changer la configuration initiale d'une application après son développement et son installation. Nous présentons ensuite une taxonomie des différents types de modifications, potentiellement applicables à une application à base de composants en cours d'exécution. Nous terminons cette section par la description de quelques propriétés importantes que doit respecter une solution de reconfiguration.

3.1 Types de reconfiguration

Indépendamment du fait qu'elle soit statique ou dynamique, la reconfiguration peut être réalisée pour différentes raisons. Ces raisons peuvent être classées en plusieurs catégories :

3.1.1 Reconfiguration correctionnelle

La reconfiguration correctionnelle est nécessaire si une application en cours d'exécution ne se comporte pas correctement ou comme prévu. Elle consiste d'abord à identifier les composants de l'application qui sont responsables du mauvais comportement, ensuite à les remplacer par de nouveaux composants supposés avoir un comportement plus adéquat. Les nouveaux composants doivent fournir les mêmes fonctionnalités que les anciens et doivent se contenter simplement de corriger leurs défauts.

3.1.2 Reconfiguration évolutive

Une application est initialement développée pour fournir un certain nombre de services à ses clients. Les besoins de ces clients peuvent changer, et de nouveaux services deviennent nécessaires. Pour répondre à cette évolution de besoins, l'application doit être étendue avec de nouvelles fonctionnalités. Dans une application à base de composants, l'évolution peut être réalisée :

- Soit en ajoutant de nouveaux composants pour assurer les nouvelles fonctionnalités requises.
- Soit en remplaçant les composants déjà existants avec d'autres composants offrant plus de fonctionnalités. Il faut garantir, dans ce cas, que le remplacement ne met pas en cause l'intégrité de l'application.

3.1.3 Reconfiguration adaptative

Une application est en général destinée à être déployée et exécutée dans un contexte particulier. Ce contexte, qui peut se traduire par l'environnement où l'application doit s'exécuter, peut changer au fil du temps. Certains changements dans le contexte peuvent rendre le comportement de l'application incohérent. Pour cette raison, l'application doit être reconfigurée pour prendre en compte les nouveaux paramètres de son contexte d'exécution.

3.1.4 Reconfiguration perfective

L'objectif de ce type de reconfiguration est d'améliorer les performances d'une application même si son comportement est cohérent. Par exemple, on se rend compte que l'implémentation d'un certain composant, s'exécutant dans un environnement contraint, n'est pas optimisée. On décide alors de remplacer l'implémentation du composant en question.

Les deux formes de reconfiguration adaptative et perfective visent parfois les mêmes objectifs, simplement leurs raisons d'être ne sont pas les mêmes. La première est nécessaire pour que l'application puisse continuer à s'exécuter, tandis que la deuxième est décidée uniquement dans un but d'amélioration.

3.2 Taxonomie

Pour reconfigurer dynamiquement une application à base de composants, plusieurs types de modifications peuvent être appliqués. Ces modifications, susceptibles de répondre aux besoins cités dans le paragraphe précédent, peuvent être classées en plusieurs catégories :

3.2.1 Modification de l'architecture

La modification de l'architecture de l'application peut se faire en ajoutant de nouvelles instances de composants, en supprimant des instances déjà existantes, ou en changeant les interconnexions entre les instances. Plusieurs opérations sont possibles :

- ❑ Ajouter une nouvelle instance de composant. L'instance peut être créée à partir d'un composant déjà déployé dans l'application, ou d'un nouveau composant à déployer avant de créer l'instance.
- ❑ Supprimer une instance existante : la suppression de l'instance ne doit pas affecter l'exécution du reste de l'application.
- ❑ Modifier les interconnexions entre les instances : quand une nouvelle instance est ajoutée à l'application, il faut la connecter aux autres instances. Quand une instance doit être supprimée, il faut d'abord l'isoler en supprimant toutes ses connexions. Également, on peut souhaiter parfois réorienter certaines connexions de l'application.

3.2.2 Modification de l'implémentation

Ce type de modification est généralement introduit pour des raisons de correction, d'adaptation aux changements de l'environnement ou d'amélioration de performances. Par exemple, on peut décider de modifier une implémentation d'un composant pour corriger des erreurs ou pour optimiser l'application.

3.2.3 Modification des interfaces

La modification de l'interface d'un composant correspond à la modification de la liste des services qu'il fournit. Ceci se traduit par l'ajout de nouvelles interfaces, par la suppression des interfaces initialement supportées par le composant, ou par la modification des opérations déclarées dans les interfaces.

3.2.4 Modification de la localisation (migration)

La modification de la localisation correspond à la migration des instances d'un site d'exécution vers un autre site pour la répartition des charges par exemple. Ce type de modification n'affecte pas l'architecture logique de l'application, cependant, la communication entre les instances migrées et les autres instances doit être adaptée selon la nouvelle localisation des instances migrées.

3.3 Propriétés des systèmes de reconfiguration

Plusieurs propriétés doivent être vérifiées par un système de reconfiguration. Ces propriétés, qui vont permettre d'évaluer et de comparer les systèmes, sont discutées dans cette sous-section.

3.3.1 La cohérence de l'application reconfigurée

Le système de reconfiguration chargé de lancer l'opération de reconfiguration, a naturellement des privilèges par rapport à l'application qu'il est en charge de reconfigurer. Cependant, ceci ne lui donne pas le droit de tout faire de cette application. Le système de reconfiguration doit préserver la cohérence de l'application :

- ❑ la reconfiguration doit être lancée à des moments adéquats,
- ❑ le système de reconfiguration ne doit pas forcer l'application à se reconfigurer brusquement,
- ❑ il ne doit agir sur cette application que si elle est dans un état stable, c'est-à-dire une situation à partir de laquelle elle peut reprendre normalement son exécution.

Dans une application à base de composants, la cohérence peut être locale ou globale [Dep01].

- ❑ Cohérence locale : elle concerne une seule instance de composant indépendamment des autres instances de l'application. Par exemple, lorsqu'une instance est altérée au moment où elle modifie l'une de ses ressources (base de données, fichier...), elle peut ne pas pouvoir revenir dans un état stable.
- ❑ Cohérence globale : elle concerne l'application dans sa globalité. Par exemple, les messages ou les données transitant dans l'application ne doivent pas être perdus.

Dans [Hic01], la cohérence d'une application a été caractérisée par les propriétés suivantes:

- ❑ Sécurité : le système de reconfiguration ne doit pas, par exemple, créer des trous de sécurité ou causer le blocage de l'application.
- ❑ Complétude : après un temps fini, le système de reconfiguration doit terminer les opérations qu'il entreprend sur l'application.
- ❑ "Well-timedness" : la reconfiguration ne doit être lancée qu'à des points temporels bien précis où l'application est considérée dans un état stable. Les points temporels de stabilité sont souvent appelés les points de reconfiguration [Tan00, Dep01].
- ❑ Possibilité de retour en arrière : il est nécessaire de prévoir des mécanismes permettant d'annuler les modifications introduites par le système de

reconfiguration en cas de problèmes, et de rétablir l'application dans un état cohérent.

3.3.2 Performance de reconfiguration

Les opérations réalisées par le système de reconfiguration doivent être efficaces : leur durée doit être la plus courte possible. Dans le même sens, le nombre de composants et d'instances de composants affectés par les opérations de reconfiguration doit être le plus faible possible.

3.3.3 Degré d'automatisation

Le degré d'automatisation représente la capacité du système de reconfiguration à décider et à lancer des opérations de reconfiguration. Ceci peut être possible si en exécution, le système de reconfiguration a toutes les informations et la logique lui permettant de décider qu'une reconfiguration est nécessaire, et également, tous les mécanismes nécessaires à réaliser une telle reconfiguration sans l'intervention d'un acteur externe.

3.3.4 Facilité d'utilisation

Raisonnement sur une certaine situation particulière, découvrir même cette situation, décider les opérations adéquates à lancer, lancer effectivement ces opérations et raisonner à nouveau sur la nouvelle situation est un long cycle de tâches complexes et non triviales. Le système de reconfiguration doit être intuitif, facile à utiliser, automatisé au maximum et doit permettre aux administrateurs de comprendre et d'agir facilement sur les applications administrées.

Kramer et Magee [KM90] proposent plusieurs propriétés à respecter pour la reconfiguration dynamique des applications modulaires. Ces propriétés sont résumées dans les points suivants :

- ❑ Les changements doivent être spécifiés en termes de la structure de l'application : les changements au niveau de la programmation d'un composant sont considérés de très bas niveau. Ils sont complexes et trop détaillés à cause du fort couplage entre les éléments de programmation du même composant.
- ❑ Les changements doivent être spécifiés d'une manière déclarative : le programmeur peut potentiellement spécifier les changements nécessaires à l'application et sous quelles conditions. Il appartient au système de reconfiguration de décider des actions qu'il faut appliquer et de l'ordre dans lequel elles sont appliquées. Ceci permet de séparer la spécification des besoins des solutions qu'il faut mettre en oeuvre pour les satisfaire.

- ❑ Le système de reconfiguration doit être séparé des applications à reconfigurer. Ceci permet d'obtenir un système de reconfiguration générique (indépendant d'une application particulière).
- ❑ Les changements doivent garantir la cohérence de l'application reconfigurée : un état cohérent est un point à partir duquel l'application peut continuer à s'exécuter normalement pour fournir des résultats corrects.
- ❑ L'application reconfigurée doit être affectée le moins possible : le système de reconfiguration doit être capable de localiser l'ensemble des composants concernés par la reconfiguration. Le reste de l'application doit continuer à s'exécuter normalement.

La réflexion, qui se traduit par la capacité d'un système à raisonner et à agir sur lui-même, a prouvé depuis de longues années son importance pour le développement de systèmes flexibles et facilement adaptables [CCL00]. La section suivante est consacrée à l'étude des systèmes réflexifs.

4 SYSTEMES REFLEXIFS

Les systèmes réflexifs ont la propriété d'être capables d'utiliser leur propre représentation pour étendre et adapter leur comportement [OGB98]. Ils incorporent des structures représentant leurs propres aspects, et grâce à ces structures, ils accèdent et manipulent leurs comportements et structures internes. Dans [Mae87], la réflexion a été définie comme une activité réalisée par un agent quand il réalise des calculs sur lui-même.

La réflexion est donc l'une des solutions permettant de créer des applications qui sont capables de maintenir, d'utiliser et de changer la représentation de leur propre conception structurelle et comportementale. Son intégration dans les langages de programmation date des travaux déterminants de B. Smith [Smi82,Smi84] au début des années 80. B. Smith a introduit la notion des tours réflexives, en considérant qu'un système réflexif peut être structuré en un nombre infini de tours, où chacune d'elles représente un niveau logique du système.

Dans une vue simplifiée des tours réflexives, il est possible de considérer qu'un système réflexif réunit, en général, deux niveaux principaux :

- ❑ Le niveau de base représente le système concret, responsable de fournir les fonctionnalités de base attendues du système.
- ❑ Le méta-niveau constitue une représentation abstraite du système.

A partir des définitions que nous avons présentées, trois notions principales peuvent être identifiées : la réification, l'introspection et l'intercession.

4.1 Réification

La réification est l'opération par laquelle quelque chose d'implicite, de non exprimé, est formulé explicitement, et, est rendu disponible à la manipulation. En d'autres termes, la réification est le processus qui permet d'extraire et de fournir des informations sur les entités composant un système. Le résultat de la réification peut être vu comme une représentation abstraite du système. Par exemple, dans le langage Java, le concept de méthode est réifié comme un objet de type `"java.lang.reflect.Method"`. Cet objet peut être manipulé d'une manière explicite comme n'importe quel autre objet de l'application.

Une question importante concerne les entités du système qui doivent faire l'objet de réification. Dans certaines considérations [Mae87], toutes les entités d'un système, qualifié de réflexif, sont considérées comme des objets, et tous ces objets doivent être réifiés. Dans d'autres considérations, et selon les besoins, seulement un sous-ensemble des entités du système réflexif est sujet à la réification. Par exemple, dans le langage Java, le concept d'instruction n'est pas réifié.

4.2 Introspection

L'introspection est l'opération de consultation des informations décrivant le système, résultant de la phase de réification. Par exemple dans une application en Java, il est possible, en exécution, de lire les propriétés d'une méthode (types des arguments, type de retour, classe...) et d'appeler cette méthode sur un objet particulier.

4.3 Intercession

L'intercession signifie la modification des structures contenant la représentation abstraite d'un système réflexif. Cette modification est naturellement reflétée sur le système lui-même. Il est important de noter que la représentation abstraite du système n'est pas uniquement une image indépendante, mais qu'elle est causalement liée à ce dernier [Mar01]. Ceci signifie que la modification du méta-niveau doit causer la modification du niveau de base et vice-versa. La relation entre le niveau de base et le méta-niveau est communément appelée *la connexion causale*. Elle garantit que le système réflexif a toujours une représentation cohérente de lui-même. En reprenant l'exemple des méthodes Java, il est possible de constater qu'il n'y a pas de moyens permettant d'agir et de modifier leur représentation réifiée. Par exemple, il n'est pas possible d'ajouter un nouvel argument ou de changer le type de retour de la méthode. D'une manière générale, la réflexion dans Java est presque limitée à l'introspection [Chi00]. La capacité de Java à altérer la structure et le comportement des applications est très limitée, elle permet uniquement d'instancier les classes, de lire et d'écrire les valeurs des attributs, et d'appeler les méthodes.

4.4 Types de réflexion

Initialement, deux types de réflexion ont été introduits : la réflexion structurelle et la réflexion comportementale. La notion de réflexion a été ensuite étendue aux aspects architecturaux des systèmes, ce qui a donné naissance à la réflexion architecturale.

4.4.1 Réflexion structurelle

Elle concerne les structures de données comme des entités de première classe [MDC92]. Elle est en général plus facile à implémenter que la réflexion comportementale [MJD96]. La réflexion structurelle permet en général l'extension de la structure des objets mais pas de leur comportement.

L'approche la plus courante pour mettre en œuvre la réflexion structurelle consiste à utiliser les méta-classes. Une méta-classe est en quelque sorte le type d'une classe. Toute classe est alors une instance d'une méta-classe. En effet, dans les langages orientés objet, en général, seuls les objets ont une existence concrète à l'exécution. Les méta-classes permettent de concrétiser les classes, et de les rendre manipulables en exécution, au même titre que les objets. Une classe englobe la structure de ses objets. En rendant les classes manipulables, on permet alors de manipuler la structure des objets. La Figure 9 montre une simple illustration des trois niveaux d'un système réflexif basé sur les méta-classes.

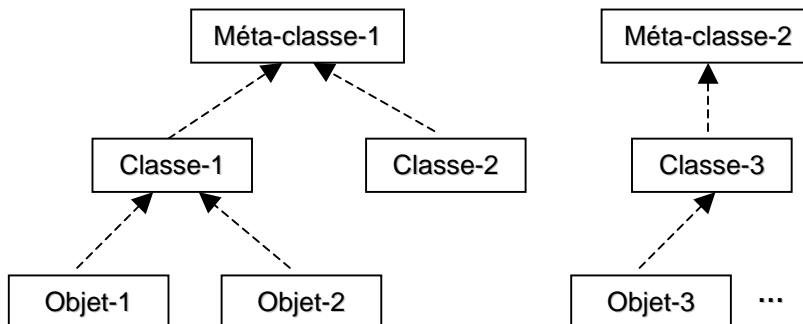


Figure 9. Utilisation des méta-classes

La classe "*java.lang.Class*" de Java est un exemple de méta-classe. Elle permet, par exemple, de retourner des informations décrivant les classes (attributs, méthodes, super-classes, etc.), d'instancier les classes, etc. Toute autre classe est alors une instance de cette méta-classe. Cependant, il n'est pas possible dans Java de définir de nouvelles méta-classes et de les associer à des classes particulières.

Malgré qu'une classe englobe aussi le comportement de ses objets, les méta-classes sont traditionnellement utilisées uniquement pour la mise en œuvre de la réflexion de structure [Fer89]. Nous verrons dans le paragraphe suivant comment la réflexion du comportement peut être mise en œuvre.

4.4.2 Réflexion comportementale

Elle concerne l'exécution des objets [MDC92]. Elle permet d'un coté de fournir une description du comportement des objets, et d'un autre coté, de donner la possibilité aux utilisateurs d'intervenir sur le déroulement de l'exécution du système. En général, la réflexion comportementale peut être réalisée par l'association d'un méta-objet à chaque objet de base. Ce méta-objet permet de contrôler et de modifier le comportement de l'objet auquel il est associé. La Figure 10 montre le lien entre un objet et son méta-objet.

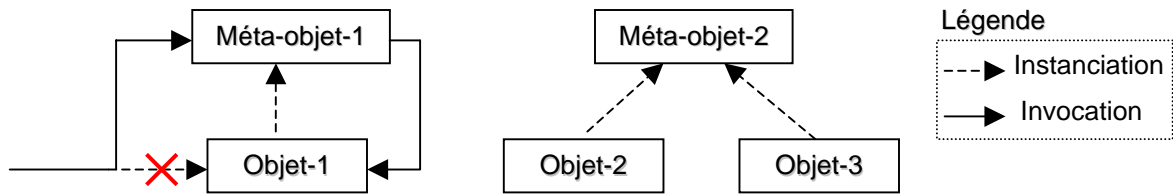


Figure 10. Utilisation des méta-objets

Un méta-objet permet de contrôler la sémantique d'un objet particulier. Pour cela, lorsqu'un appel de méthode a lieu sur un objet qui possède un méta-objet, l'appel est intercepté, réifié, et remplacé par un appel à une méthode du méta-objet [BR00]. Le méta-objet peut alors faire le traitement souhaité : il peut traiter lui-même l'appel, ou le déléguer tel quel à l'objet de base. Il peut aussi transformer le nom de la méthode ou la valeur des arguments, réaliser des pré et post traitements, etc.

4.4.3 Réflexion architecturale

Au lieu qu'elle soit appliquée à la structure des objets ou à leur comportement, la réflexion architecturale s'intéresse plutôt aux interconnexions entre les objets. Dans [CS+99] par exemple, les auteurs ont introduit la notion des tours architecturales réflexives qui peuvent être vues comme une extension des tours réflexives introduites par Maes [Mae87], en considérant qu'un système architecturalement réflexif est constitué d'un ensemble infini de couches architecturales causalement connectées. La Figure 11 montre une simple illustration d'un système architecturalement réflexif.

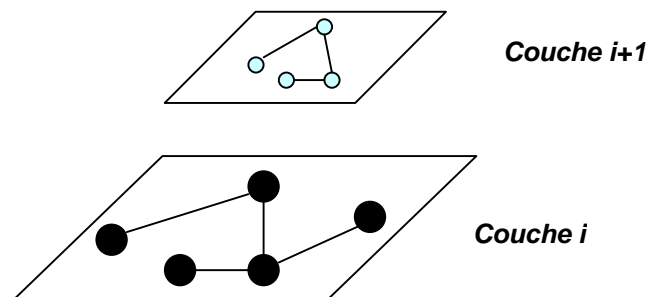


Figure 11. Système architecturalement réflexif

Les entités de chaque couche sont considérées comme la réification des entités de la couche juste au dessous. Tout changement appliqué à une entité d'une certaine couche, est reflété sur l'entité qu'elle réifie, et vice-versa.

4.4.4 Synthèse

La réflexion, qui a fait l'objet de cette section, est un concept présent depuis longtemps dans quelques langages de programmation. Elle a prouvé son efficacité dans la construction de systèmes flexibles, dotés de grandes capacités rendant plus facile leur adaptation. La réflexion consiste d'abord à réifier, selon les besoins, tout ou partie des éléments d'un système. Cette réification définit un méta-niveau qui n'est rien d'autre qu'une image du système. Ce méta-niveau peut être consulté et peut être modifié, ce qui provoque la modification du système grâce à la connexion causale qui assure qu'à tout moment, chaque niveau est une vision cohérente de l'autre niveau.

5 CONCLUSION

Ce chapitre a été dédié à la présentation de quelques concepts clés constituant le contexte scientifique dans lequel s'est déroulée cette thèse. Nous avons d'abord introduit le modèle de programmation par composants, en montrant à travers l'analyse de quelques modèles que les notions d'évolution et d'adaptation dynamiques ne sont pas adressées explicitement, et ne constituent pas des éléments déterminants de ces modèles. Nous avons ensuite présenté à travers plusieurs exemples la problématique de reconfiguration dynamique. L'accent a été plus particulièrement mis sur les raisons qui poussent à faire une reconfiguration à un moment donné, puis sur les modifications potentiellement applicables à une application à base de composants pour la reconfigurer dynamiquement, et enfin, sur les propriétés que doit avoir un système de reconfiguration. Enfin, nous avons illustré la notion de réflexion, souvent utilisée comme un support pour développer des systèmes ouverts, flexibles et facilement adaptables.

Dans le chapitre suivant, nous présentons quelques travaux qui ont traité la problématique de reconfiguration dynamique. Nous mettons l'accent plus particulièrement sur les solutions de reconfiguration autour des modèles de composants.

CHAPITRE 3

ETAT DE L'ART

PRÉAMBULE.

Ce chapitre présente les différentes approches de reconfiguration dynamique que nous avons étudiées dans cette thèse. Chaque approche est illustrée par un exemple de système. Nous présentons d'abord des solutions par rapport à leur propriété d'ouverture. Ces solutions de reconfiguration peuvent être fermées, partiellement ouvertes ou ouvertes. Puis nous présentons des solutions de reconfiguration ayant adressé, comme granularité de changement, la procédure, le module, l'objet et enfin le composant. Deux solutions d'administration issues de l'industrie sont ensuite analysées. Et avant de conclure, nous présentons deux solutions de reconfiguration matérielles.

1 INTRODUCTION

La reconfiguration dynamique est un sujet qui a mobilisé une grande communauté de chercheurs depuis de longues années. L'un des travaux qui a explicitement adressé ce sujet date déjà de 1976. Il a été réalisé par R. Fabry [Fab76] qui a tenté de répondre à la question : comment concevoir des systèmes dans lesquels les modules peuvent être changés à la volée (à l'exécution). Dans ce chapitre nous présentons les différentes solutions de reconfiguration dynamique que nous avons étudiées. Nous mettons d'abord l'accent sur les approches générales de reconfiguration dynamique, puis nous présentons des exemples de systèmes ou de solutions soutenues par ces approches.

A cause de la diversité des domaines d'application, des modèles de programmation utilisés au fil du temps, des contraintes à respecter et même de la nature des solutions, il est difficile de trouver un seul critère déterminant qui permette d'analyser et de regrouper les différentes solutions de reconfiguration. Nous présentons donc, dans ce chapitre, ces solutions, en les classifiant selon un ensemble de critères qui nous semblent importants.

L'une des propriétés les plus importantes est l'ouverture de la solution de reconfiguration. Cette propriété reflète la capacité de la solution à supporter des extensions et des modifications ultérieures, initialement imprévues et applicables même en exécution. Ce chapitre situe d'abord les solutions étudiées par rapport à la propriété d'ouverture. Trois catégories de solutions peuvent être dégagées : les solutions fermées, les solutions partiellement ouvertes et les solutions ouvertes. Nous présentons ensuite d'autres

classifications en fonction d'autres critères comme le modèle de programmation utilisé, le support de reconfiguration et la granularité de changement.

Ainsi, nous discutons dans ce chapitre les solutions de reconfiguration :

- ❑ fermées,
- ❑ partiellement ouvertes,
- ❑ ouvertes,
- ❑ procédurales,
- ❑ modulaires,
- ❑ orientées objet,
- ❑ orientées composant,
- ❑ industrielles,
- ❑ et enfin, les solutions de reconfiguration basées sur un support matériel.

Chacun de ces points est illustré par un exemple de système issu soit du monde académique, soit du monde industriel.

2 SOLUTIONS DE RECONFIGURATION FERMEES

Nous qualifions une solution de reconfiguration, de fermée, si toutes les modifications susceptibles d'être faites sur une application à reconfigurer, sont prévues à l'avance et sont codées au moment du développement.

Il n'est évidemment pas possible de prévoir tous les problèmes qui peuvent arriver à un moment donné. Il est aussi difficile de prévoir les besoins qui peuvent apparaître à un moment donné après la mise en service de l'application. Il n'est donc pas possible de concevoir des systèmes fermés, capables de s'adapter et de répondre à tous types de situations. En contre partie, même si la fermeture peut être perçue à première vue comme un inconvénient, cette approche est intéressante si on souhaite concevoir des systèmes maîtrisables. En fait, un système fermé est complètement contrôlable, car tous les états dans lesquels il peut passer sont déterminés à l'avance.

A partir de cette définition nous pouvons constater que, tous les systèmes qui nécessitent que le développeur écrive explicitement le code permettant d'adapter l'application à une situation donnée, peuvent être considérés comme des systèmes fermés. OSGi est un exemple de système, où le développeur est responsable de prévoir, à l'avance, toutes les situations qui peuvent survenir à l'exécution. Le programmeur doit donc développer son application dans cette optique. L'adaptabilité dans OSGi est illustrée dans la suite de cette section.

2.1 Exemple d'adaptabilité dans OSGi

Dans OSGi, les applications peuvent être vues comme un ensemble d'instances de composants qui s'exécutent au dessus du framework OSGi. Par nature, les instances peuvent apparaître et disparaître à l'exécution. Le développeur doit prendre en considération cette propriété, et doit faire en sorte que les composants développés, soient conscients de la nature volatile des services qu'ils utilisent. Dans la suite, nous illustrons ceci à travers un exemple d'application.

L'application est constituée de deux instances de composants, un client et un serveur. Le serveur gère une imprimante qui peut être utilisée par les clients pour imprimer des rapports. L'imprimante peut devenir indisponible soudainement. Lorsque c'est le cas, l'instance qui la représente devient invalide. Comme le montre le code suivant, le serveur doit dés-enregistrer le service d'impression.

```
printerServiceReg.unregister();
```

Le client doit également être conscient de la disparition du service d'impression, et doit réagir en le libérant et en évitant de l'appeler. En pratique, le client doit, par exemple, écouter les événements émis par le framework OSGi. Lorsqu'un événement indiquant la disparition du service d'impression est reçu, le client doit libérer le service. Ceci est illustré par l'exemple suivant.

```
public void serviceChanged(ServiceEvent event)
{
    if(event.getType()==event.UNREGISTERING)
        printerService=null;
}
```

Inversement, lorsque le service d'impression devient disponible, le client reçoit un événement du framework. Comme illustré dans le code suivant, le client doit récupérer la référence au service d'impression pour pouvoir l'utiliser.

```
ServiceReference serviceRef =
    bundleContext.getServiceReference("prnService");
IPRNService ser =
    (IPRNService) bundleContext.getService(serviceRef);
```

2.1.1 Discussion

L'exemple, malgré sa simplicité et malgré les facilités supportées par le framework OSGi (notamment le mécanisme des événements) montre le travail supplémentaire auquel le développeur doit faire face. Si la liaison entre le client et le serveur est réifiée et, par conséquent, manipulable, on peut facilement imaginer un mécanisme qui gère le problème de disparition des services, d'une manière complètement transparente et automatique.

A l'inverse, le développeur d'une telle application a une maîtrise complète de son exécution, car aucune intervention, non-autorisée explicitement, n'est possible après la mise en service de l'application. Le développeur peut également spécialiser et limiter les solutions d'adaptabilité selon ses besoins.

3 SOLUTIONS DE RECONFIGURATION PARTIELLEMENT OUVERTES

Du fait de l'impossibilité d'imaginer et de prévoir toutes les adaptations nécessaires, à un moment donné, certains systèmes prévoient explicitement un ensemble de points d'ouverture. Au sein du système, ces points peuvent être vus comme des trous qui peuvent être remplis, à n'importe quel moment, par de nouvelles fonctionnalités. L'ouverture réside surtout dans le fait que les fonctionnalités destinées à être branchées au système original, ne sont pas prédéfinies.

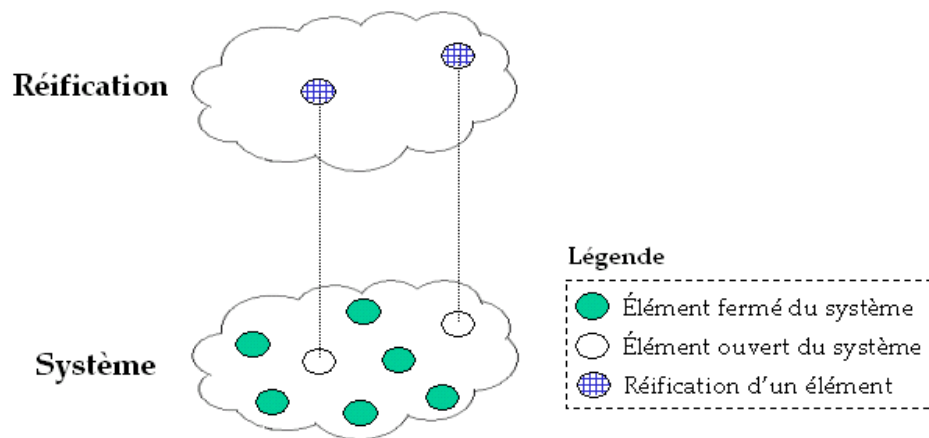


Figure 12. Solutions de reconfiguration partiellement ouvertes

Comme le montre la Figure 12, les éléments ouverts du système, sont réifiés pour qu'ils puissent être manipulés. Les systèmes à plugins sont un exemple typique de cette catégorie. Nous présentons dans la suite OpenVL, un exemple de système à plugins. OpenVL utilise les plugins pour supporter l'évolution dynamique.

3.1 OpenVL

OpenVL [LK03] (Open Volume Library) est un système de traitement des données volumétriques, développé à l'université de Stony Brook (USA). La flexibilité et l'extensibilité ont été fixées dès le départ, comme les objectifs principaux de ce système. OpenVL fournit une API standard et uniforme pour l'accès, le stockage et le traitement des données. Il supporte différents types de formats de fichiers et différents mécanismes de stockage grâce à

des plugins dynamiques. Il dispose d'une classe d'utilitaires pour réaliser, entre autres, des opérations sur les vecteurs et les matrices.

3.1.1 Architecture du système

La nature des données à stocker et à traiter n'est pas figée. Les fonctionnalités supportées par le système, peuvent être étendues en créant de nouveaux plugins. Ces plugins peuvent être développés à tout moment par les utilisateurs et intégrés, dynamiquement, au système. Comme illustré par la Figure 13, deux types de plugins peuvent être distingués :

- ❑ Les plugins du système : ils font partie du noyau du système, et sont nécessaires pour son fonctionnement.
- ❑ Les plugins utilisateur : ils peuvent être ajoutés selon les besoins. Par exemple, en fonction des nouvelles données à traiter, des nouveaux formats, des nouveaux traitements, des nouvelles présentations, etc.

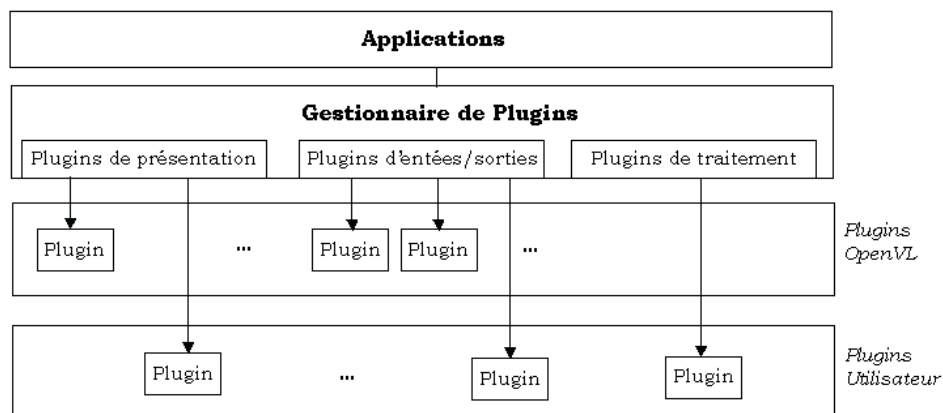


Figure 13. Architecture à plugins de OpenVL

3.1.2 Plugins dynamiques

Les plugins dynamiques, qui assurent l'extensibilité de OpenVL, sont construits comme des bibliothèques dynamiques (DLLs). Ils sont chargés dynamiquement si les fonctionnalités qu'ils implémentent sont demandées. OpenVL fournit un composant pour gérer les plugins (Gestionnaire des plugins). Ce composant a plusieurs responsabilités, comme la recherche de tous les plugins installés dans le système, la création d'une base de données des plugins disponibles, le chargement des plugins appropriés lorsque des fonctionnalités sont demandées, le déchargement des plugins non utilisés, etc.

Trois parties du système sont concernées par l'ajout des plugins (Figure 13) : la partie présentation qui se charge de l'organisation des données sous un certain format, la partie entrées/sorties qui se charge de la lecture des données à partir des fichiers clients, et la partie

traitement qui fournit des algorithmes pour le traitement des données. Un plugin doit être destiné exclusivement à l'une de ces trois parties. Il doit implémenter une interface spécifique à la partie visée.

3.1.3 Discussion

OpenVL est un système qui supporte l'évolution dynamique. Il utilise les plugins dynamiques pour assurer cette évolution. Comme nous l'avons expliqué, uniquement trois parties du système sont censées être extensibles, et supporter par conséquent l'ajout de nouveaux plugins. Ceci coïncide exactement avec notre définition des solutions de reconfiguration partiellement ouvertes. Les points d'extension sont bien définis. Par contre, les éléments destinés à être inclus dans ces points ne sont pas fixés à l'avance. Ceci permet des possibilités d'adaptation plus riches, car, à tout moment après la mise en service du système, les utilisateurs peuvent créer de nouvelles fonctionnalités et les brancher aux points appropriés.

4 SOLUTIONS DE RECONFIGURATION OUVERTES

Contrairement aux systèmes des catégories précédentes, les systèmes ouverts considèrent tous les éléments du système, comme des entités manipulables et adaptables (Figure 14). Ceci permet de concevoir des systèmes flexibles et très riches en termes d'adaptabilité. En contre partie, et puisque les modifications qui peuvent être appliquées ne sont pas prédéfinies, et peuvent affecter tous les éléments du système, le comportement de ce dernier peut devenir incontrôlable.

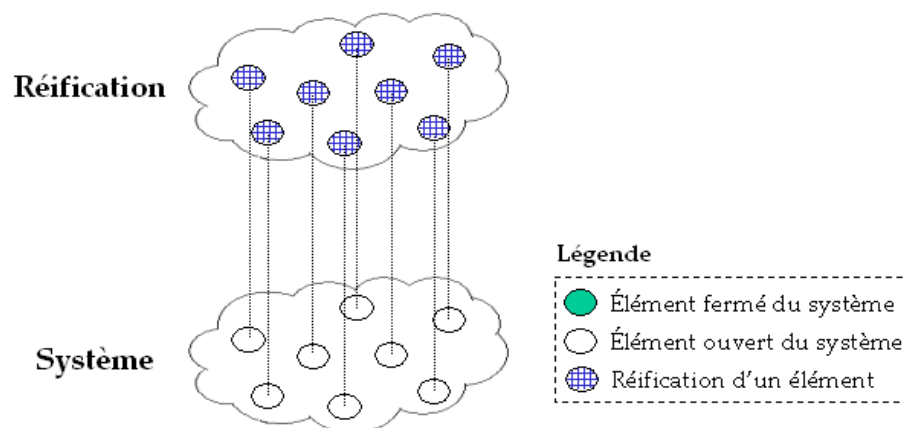


Figure 14. Solutions de reconfiguration ouvertes

La réflexion est typiquement l'un des moyens utilisés pour concevoir des systèmes ouverts. Comme nous l'avons expliqué dans le chapitre précédent, un système réflexif est

fondamentalement basé sur la réification de ses éléments. L'ensemble des éléments réifiés, constitue alors les points d'ouverture du système, qui peuvent être adaptés. Dans la suite, nous présentons Iguana/J, une solution de reconfiguration dynamique basée sur la réflexion.

4.1 Iguana/J

Iguana/J [RC00, RC02] est une plate-forme développée, au début des années 2000, par le Distributed Systems Group au Trinity College de Dublin. Elle a été conçue pour supporter des adaptations dynamiques non anticipées des applications Java. La plate-forme est une extension de la machine virtuelle de Java. L'objectif de l'extension était d'ajouter la réflexion comportementale, pour supporter l'interception et la gestion des opérations, comme la création des objets, les invocations de méthodes, la lecture et l'écriture des attributs. Grâce à la capacité d'interception, de nouveaux comportements peuvent être ajoutés aux applications Java, sans avoir besoin de les modifier et de les recompiler. Initialement, Iguana était proposé comme une architecture réflexive indépendante d'un langage particulier [Gow97]. Elle a été d'abord implémentée pour le langage C++ (Iguana/C++) puis pour le langage Java (Iguana/J).

4.1.1 Architecture du système

La séparation des préoccupations [TD+00, ATB01] est l'un des principaux objectifs visés par les concepteurs de Iguana/J. Cette séparation est matérialisée par une architecture réflexive, supportant deux niveaux : un niveau de base et un méta-niveau. La Figure 15 illustre la relation entre ces deux niveaux.

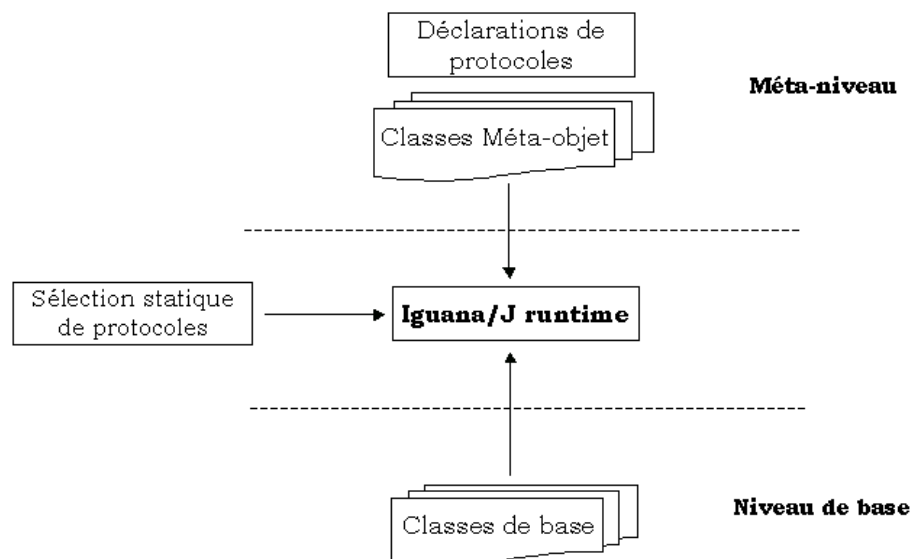


Figure 15. Architecture du système Iguana/J

Le niveau de base reflète les applications concrètes. Le méta-niveau reflète leur image. Il est composé d'un ensemble de classes et de protocoles qui permettent d'associer ces classes à des opérations de la JVM (création d'instances, exécution de méthodes...).

Comme le montre la Figure 15, le modèle de programmation de Iguana/J regroupe quatre éléments : les classes de base, les classes méta-objet, les déclarations de protocoles, et la sélection statique des protocoles.

4.1.1.1 Classes de base

La programmation du niveau de base n'a rien de différent par rapport à la programmation Java standard. Les classes et les interfaces de l'application peuvent être développées sans prendre en considération la plate-forme Iguana/J.

4.1.1.2 Classes méta-objet

Les classes méta-objet peuvent être créées en sous classant les classes appropriées de la plate-forme Iguana/J, et en redéfinissant les méthodes de réification, définies par ces classes.

```
class TraceExecution extends MExecute {
    Object execute(Object o, Object[] args, Method m) {
        System.out.println("Avant appel " + m.getName());
        result = m.invoke(o, args);
        System.out.println("Après appel " + m.getName());
    }
}
```

L'exemple précédent illustre le code d'une classe méta-objet appelée *TraceExecution*. Cette classe étend la classe de base *MExecute* (exécution des méthodes), définie par la plate-forme Iguana/J. Les instances de la classe *TraceExecution* permettent de réifier les appels de méthodes. La classe *TraceExecution* redéfinit la méthode *execute* pour insérer la sémantique nécessaire (afficher un message avant et après l'appel).

4.1.1.3 Déclarations de protocoles

Les déclarations de protocoles spécifient les opérations de la JVM à réifier, et les classes méta-objet utilisées pour leur réification. L'exemple suivant exprime qu'il faut réifier l'exécution des méthodes. Il déclare, également, que la réification doit être faite par la classe méta-objet *TraceExecution*.

```
protocol TraceProtocole {
    reify Execution: TraceExecution;
}
```

Les déclarations de protocoles sont écrites dans des fichiers séparés. Ces fichiers sont ensuite compilés en des classes Java, en utilisant le compilateur de protocoles, fourni par

Iguana/J. Dans les déclarations, uniquement les catégories comportementales peuvent être attachées à des classes méta-objet. Les catégories comportementales concernent les aspects qui expriment le comportement des applications, comme l'appel des méthodes, la création et la destruction des objets, etc. Les catégories structurelles, d'un autre côté, concernent la structure des objets.

Les catégories comportementales supportées par Iguana/J, correspondent aux opérations réifiées, résumées par le Tableau 1 (paragraphe 4.1.5).

4.1.1.4 Sélection des protocoles

La sélection d'un protocole, à associer à une classe ou à une interface de base, se fait par une déclaration statique, dans un fichier de sélection de protocoles. Ce fichier est utilisé par le moteur d'exécution de Iguana/J pour détecter le chargement, par la machine virtuelle de Java, des classes et des interfaces qui doivent être associées à un protocole. Chaque ligne du fichier de sélection, associe une classe ou une interface de base à un protocole. Un protocole peut être associé à plusieurs classes ou interfaces. Par contre, une classe ou une interface de base, ne peut être associée qu'à un seul protocole.

```
class MaClasseDeBase ==> TraceProtocole;  
interface MonInterfaceDeBase ==> TraceProtocole;
```

Dans la déclaration précédente, par exemple, une classe et une interface de base ont été associées au protocole *TraceProtocole*.

Les associations entre les classes (ou interfaces) du niveau de base et les protocoles, sont déclarées dans des fichiers séparés. Elles ne sont pas mélangées avec le code du niveau de base, ou celui du méta-niveau. Cette séparation assure une grande flexibilité : il n'est pas nécessaire d'intégrer dans le niveau de base, des références vers le méta-niveau. Également, le code du niveau de base ne nécessite pas d'être instrumenté, pour qu'il soit connecté au méta-niveau.

4.1.2 Dérivation et composition des protocoles

Un seul protocole peut être associé à une classe à un moment donné. Cependant, plusieurs protocoles peuvent être composés, pour former un nouveau protocole qui les représente tous. Ceci est possible grâce au mécanisme de dérivation de protocoles. Dans l'exemple suivant, le protocole *P3* est dérivé à partir des deux protocoles *P1* et *P2*. *P3* hérite alors des déclarations de *P1* et de *P2*. Il peut également définir de nouvelles déclarations. Contrairement au mécanisme d'héritage dans les langages objets, la surcharge des déclarations n'est pas possible. Cette restriction a pour objectif de garantir que les protocoles dérivés, ne suppriment pas le comportement, introduit par leur super-protocole.


```
protocol P1 {
    reify Creation: MonCreate1;
    reify Invocation: MonExecute1;
}

protocol P2 {
    reify Invocation: MonExecute2;
}

protocol P3 : P1, P2 {
    reify Creation: MonCreate3;
    reify StateRead: MonRead3;
}
```

Dans l'exemple précédent, le protocole *P3* va contenir des méta-objets pour gérer trois catégories d'opérations. Il va gérer la création en utilisant la composition de deux méta-objets (instances de *MonCreate1* et *MonCreate3*). La composition des méta-objets utilise une simple structure chaînée. Lorsqu'une opération est interceptée, elle est passée tout au long de la chaîne des méta-objets. Chacun d'eux est responsable de transmettre l'opération interceptée au méta-objet suivant. Il peut, également, exécuter un code avant et après le passage de l'opération interceptée.

Le protocole *P3* va aussi gérer l'invocation en utilisant la composition de deux méta-objets, instances de *MonExecute1* et de *MonExecute2*. Enfin, il va gérer la lecture des attributs en utilisant un méta-objet, instance de la classe méta-objet *MonRead3*.

4.1.3 Association des protocoles aux objets

Lorsqu'un protocole est associé à une classe de base, tous les objets de cette classe sont concernés par le protocole. En d'autres termes, l'association déclenche la création d'un ensemble de méta-objets, et leur connexion aux objets de la classe.

Même si le protocole est associé dynamiquement à la classe, tous les objets, soit ceux qui existaient, soit ceux qui vont être créés, se voient affecter automatiquement au protocole associé à leur classe. Si un ancien protocole existe, il est automatiquement remplacé par le nouveau protocole (un seul protocole peut être associé à la classe).

Iguana/J permet aussi d'associer un protocole à un seul objet, indépendamment des autres objets de la même classe. Ceci peut être réalisé en utilisant un mécanisme explicite d'association. Une méthode statique de la classe *Meta*, définie par Iguana/J, permet au programmeur de réaliser, ou de changer l'association, entre un objet particulier et un protocole. Dans le code suivant, par exemple, on demande d'associer le protocole *TraceProtocole* à un objet particulier. Les autres objets de *MaClasse* ne sont pas affectés par cette association.

```
MaClasse obj = new MaClasse() ;  
Meta.associate(obj, "TraceProtocole") ;
```

Il est important de noter que, même s'il est possible d'associer un protocole à un objet particulier, il est nécessaire de prendre en considération les règles de méta-typage des protocoles. En particulier, le nouveau protocole doit être dérivé du protocole associé à la classe.

4.1.4 Rôles de développement

Pour créer une application, dynamiquement adaptable, Iguana/J prévoit trois rôles de développement complémentaires.

- ❑ **Développeur du niveau de base** : il crée les classes, les interfaces et les ressources responsables de fournir les fonctionnalités attendues du système.
- ❑ **Développeur du méta-niveau** : il crée les classes méta-objets, utilisées pour réifier les classes et les interfaces du niveau de base.

Ces deux premiers développeurs sont censés être indépendants. Idéalement, aucun des deux n'a besoin de connaître le développement réalisé par l'autre.

- ❑ **Intégrateur du système** : il est responsable de fusionner les éléments fournis par les deux premiers développeurs. Il se charge ainsi de construire le système global. L'intégration consiste à créer les déclarations de sélection, qui permettent de lier les deux niveaux.

4.1.5 Modifications supportées

Iguana/J supporte sept types de modifications. Ces modifications concernent principalement le comportement des applications. Le Tableau 1 résume les opérations de base qui peuvent être interceptées, et par conséquent, modifiées. Il montre également la classe méta-objet qui peut être associée à chaque opération interceptée, et la méthode qu'il faut implémenter pour changer la sémantique de cette opération.

Opération interceptée	Classe méta-objet associée	Méthode associée
Création d'objets	ie.tcd.iguana.MCreate	create()
Suppression d'objets	ie.tcd.iguana.MDelete	delete()
Appel de méthodes	ie.tcd.iguana.MSend	send()
<i>Dispatch</i> de méthodes	ie.tcd.iguana.MDispatch	dispatch()
Exécution de méthodes	ie.tcd.iguana.MExecute	execute()
Lecture d'attributs	ie.tcd.iguana.StateRead	read()
Écriture d'attributs	ie.tcd.iguana.StateWrite	write()

Tableau 1. Modifications supportées par Iguana/J

Remarque : le *dispatch* concerne la liaison statique ou dynamique. C'est le processus utilisé par le compilateur (*dispatch* statique) ou par la machine d'exécution (*dispatch* dynamique), pour sélectionner les méthodes à appeler, en fonction du nombre et des types d'arguments.

4.1.6 Discussion

Comme nous l'avons cité dans le chapitre précédent, le langage Java ne supporte que la notion d'introspection, qui permet de consulter les informations des entités réifiées. Cependant, l'intercession, qui permet d'agir sur ces informations, n'est pas supportée par le langage. Egalement, seules les informations de structure sont prises en compte. Les informations comportementales, quant-à-elles, sont complètement ignorées. Iguana/J a principalement étendu la machine virtuelle de Java, pour compenser ces limitations. L'utilisation des protocoles méta-objets et la désignation explicite de plusieurs rôles de développement, facilitent la compréhension et l'utilisation de l'architecture. Cependant, cette séparation n'est pas toujours évidente, et notamment, lorsqu'on souhaite attacher un méta-objet à un objet particulier. Dans ce cas, le programmeur du niveau de base, doit avoir une connaissance du méta-niveau.

Lorsqu'un appel de méthode est intercepté, le méta-objet responsable de cette interception peut exécuter un code avant de rediriger l'appel vers la méthode originale. La redirection de l'appel se fait en appelant la méthode *invoke()*, définie dans l'API de réflexion de Java [Bou00]. Cependant, l'utilisation de cette méthode affecte considérablement les performances des applications. Nous avons remarqué que dans la version la plus récente de Iguana/J, la méthode *proceed()* a été ajoutée pour éviter ce problème de performance. La méthode *proceed()* est héritée par tous les méta-objets. Elle permet de rediriger les opérations interceptées, vers leurs destinations d'origine.

Nous pensons que dans l'idéal, les applications doivent être développées sans aucune considération de leur future reconfiguration. Nous croyons également que cet objectif est très

difficile à atteindre. L'application doit fournir des moyens, qui permettent au système de reconfiguration d'interagir avec elle. Le lien entre l'application concrète et son méta-niveau peut être réalisé de plusieurs manières différentes. Comme le montre la Figure 16, plusieurs possibilités peuvent être considérées :

- ❑ Cas 1 : Le programmeur réalise lui-même le lien, en codant des appels vers le méta-niveau. Par exemple, lorsqu'une méthode est appelée, on peut insérer un appel, pour informer le méta-niveau de cet événement.
- ❑ Cas 2 : Le programmeur ne réalise pas le lien explicitement, il ne se préoccupe pas de la présence d'un méta-niveau. Les applications sont ensuite instrumentées pour inclure le lien vers le méta-niveau. L'instrumentation peut agir, soit sur le code source de l'application, soit sur le code binaire. Ceci peut être décidé selon la disponibilité du code source, et selon les technologies d'instrumentation disponibles. La programmation orientée aspect (AOP) [EAB02, PRS04], est l'une des solutions qui peuvent être utilisées pour réaliser le lien causal entre une application et son méta-niveau.
- ❑ Cas 3 : Une autre approche consiste à intervenir plutôt au niveau de la machine d'exécution. En modifiant cette machine, on donne la possibilité d'intercepter certaines opérations (appels de méthodes, accès aux attributs, modification des attributs, etc.). La modification de la machine virtuelle permet aussi d'ajouter les structures de données et les traitements nécessaires, permettant d'augmenter les capacités de la machine (pour supporter l'intercession par exemple). Iguana/J est basé sur cette troisième approche.

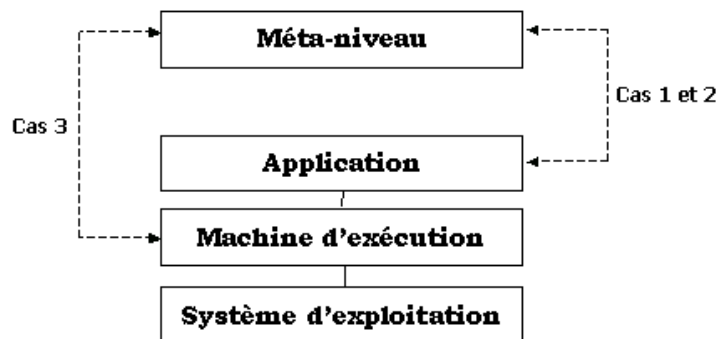


Figure 16. Connexion entre le niveau de base et le méta-niveau

Par rapport aux deux premières approches, nous pensons que le choix d'une méthode ou d'une autre, peut se faire selon la complexité du code de connexion vers le méta-niveau, à ajouter dans le code de base. Si ce code est simple, la première solution peut s'imposer, car elle ne nécessite pas des outils d'instrumentation. Dans le cas où le code de connexion est plus complexe, l'utilisation d'un outil d'instrumentation est plus adéquat. La deuxième solution a

aussi l'avantage de la transparence, car elle ne nécessite pas la participation du programmeur.

La troisième approche permet d'automatiser et de traiter, d'une manière transparente, la connexion entre l'application et son méta-niveau. Toutes les applications, s'exécutant sur une machine d'exécution augmentée, bénéficient systématiquement des capacités de réflexion. L'inconvénient majeur, est que l'application reste liée à la machine d'exécution modifiée. Par conséquent, elle ne peut pas être livrée, indépendamment de cette machine (si on souhaite garder sa réflexivité).

Nous avons, jusque là, analysé quelques solutions de reconfiguration par rapport à la propriété d'ouverture. Dans la suite de ce chapitre, nous étudions d'autres approches qui ont adressé différentes granularités de changement. Nous présentons des solutions procédurales, puis modulaires, ensuite orientées objet, et pour finir, nous mettons l'accent sur les solutions orientées composant.

5 SOLUTIONS DE RECONFIGURATION PROCEDURALES

Les solutions procédurales s'intéressent au remplacement dynamique d'une procédure par une autre. Le choix de la procédure comme unité de modification est motivé, en général, par le fait que la procédure constitue l'unité basique pour l'abstraction du code [Lee83].

5.1 DYMOS

DYMOS [Lee83] signifie DYnamic MODification System. Il a été développé en 1983 par *Insup Lee* dans ses travaux de thèse, à l'université de Wisconsin Madison. Il fournit un environnement intégré et complet pour le développement des applications et pour leur modification dynamique.

DYMOS adresse les applications écrites en StarMod [COO88], un langage de programmation concurrent, similaire à Modula [MODULA]. Il permet le remplacement dynamique des procédures. Puisque StarMod est un langage modulaire, DYMOS supporte également le remplacement des modules en exécution.

Le programmeur modifie et recompile le code source des méthodes et des modules, qui doivent être dynamiquement adaptés. Il demande ensuite au système de réaliser le remplacement. L'adaptation est réalisée par un processus de modification dynamique, qui est exécuté en parallèle avec les processus des utilisateurs.

5.1.1 Architecture du système

Comme illustré par la Figure 17, DYMOS est constitué de cinq éléments : un interpréteur de commandes, un gestionnaire de code source, un éditeur, un compilateur et un support d'exécution.

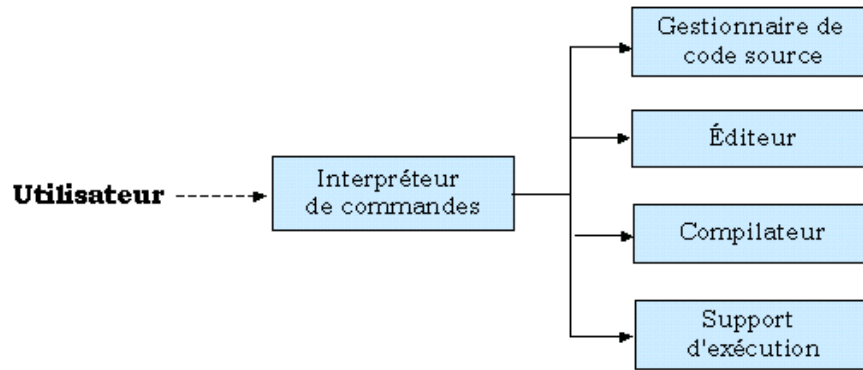


Figure 17. Architecture de DYMOS

□ **Interpréteur de commandes**

Il joue le rôle d'interface entre l'utilisateur et le système. Il reçoit et analyse les commandes de l'utilisateur. Il appelle ensuite le composant du système, qui est concerné par la commande.

□ **Gestionnaire de code source**

Son rôle est de stocker et de retrouver le code source d'une procédure ou d'un module. Le code source d'une application est organisé dans une structure hiérarchique, créée à la compilation, et modifiée à chaque adaptation. Chaque nœud de la structure représente un module ou une procédure. Deux versions peuvent être maintenues à chaque nœud : la version *courante* et la version *prévisionnelle*. La version courante correspond à celle qui est en exécution, et la version prévisionnelle correspond à celle qui va remplacer la version courante. Une seule version prévisionnelle peut être maintenue à un moment donné.

□ **Editeur**

Il permet d'éditer le code des procédures et des modules stockés dans le système. Il permet également de créer de nouvelles versions prévisionnelles, et de les ajouter à la structure, maintenue par le gestionnaire de code source.

□ **Compilateur**

Il permet de compiler le code source des modules et des procédures en un code binaire. Il supporte la compilation séparée. Lorsqu'un module est recompilé, le compilateur tente d'assigner aux variables et aux procédures externes les mêmes adresses que dans la version précédente du module. Ceci permet de minimiser l'impact sur le reste de l'application. Si les adresses nouvellement assignées sont différentes des anciennes, les modules et les procédures qui utilisent la version recompilée, doivent être recompilés à leur tour pour maintenir la cohérence de l'application. Lorsqu'une procédure est recompilée, elle est toujours affectée à la même adresse. Ses variables locales sont assignées indépendamment de l'ancienne version, car, elle ne peuvent pas être partagées.

□ Support d'exécution

Le support d'exécution consiste notamment en un processus de modification qui reçoit les requêtes de l'éditeur de commandes. Il charge ensuite les nouvelles versions de procédures ou de modules. Une commande de modification a la forme suivante :

```
update P, Q, M when P, R, T idle
```

Cette commande signifie qu'il faut adapter les deux procédures P et Q , aussi bien que le module M . L'adaptation ne doit se faire que si les procédures P , R et T sont inactives.

5.1.2 Discussion

L'adaptation dynamique dans DYMOSS consiste à remplacer les procédures ou les modules d'une application, en exécution. DYMOSS gère automatiquement les dépendances entre les procédures et les modules. Une commande d'adaptation d'une procédure ou d'un module, déclenche automatiquement l'adaptation des procédures et/ou des modules, dont elle dépend. Pour que l'adaptation dynamique soit efficace, il faut minimiser les liens entre les différents éléments de l'application. Si une application n'est pas bien conçue de ce point de vue, l'adaptation d'une seule procédure peut déclencher l'adaptation de toute l'application.

DYMOSS donne la possibilité à l'utilisateur de spécifier une condition à vérifier avant de réaliser l'adaptation. Cette condition permet d'exprimer quelles procédures doivent être inactives, pour que l'adaptation puisse avoir lieu.

5.2 PODUS

PODUS [SF93], acronyme de "Procedure-Oriented Dynamic Update System", est un système qui a adressé l'adaptation des procédures en exécution. Proposé en 1993, PODUS permet l'adaptation incrémentale des procédures en mémoire. Il a résulté de plusieurs travaux de *Mark Segal* et *Ophir Frieder* [SF89, FS91, SF93]. Plusieurs versions de la même procédure peuvent coexister. Chacune est chargée dans un espace mémoire séparé, avec sa propre table de liaisons. Tous les appels entre les procédures sont indirects et passent forcément par cette table.

5.2.1 Adaptation des procédures

Une procédure ne peut être changée que si elle est inactive. Une procédure est inactive si :

- elle n'est pas en cours d'exécution,
- aucune des procédures, susceptibles d'être appelées par sa nouvelle version, n'est en cours d'exécution. Ceci est nécessaire car, comme nous l'expliquerons dans la suite, l'adaptation d'une procédure entraîne automatiquement l'adaptation de

toutes les procédures que sa nouvelle version doit appeler. Les procédures susceptibles d'être appelées sont déterminées en analysant le code de la nouvelle version de la procédure.

Chaque procédure comprend deux parties :

- la spécification du comportement (l'interface),
- l'implémentation.

L'adaptation d'une procédure se traduit par le changement de la liaison entre sa spécification et son implémentation. Si la modification doit aussi affecter la spécification, une *inter-procédure* est nécessaire pour transférer les appels et retourner les résultats.

5.2.2 Utilisation des inter-procédures

PODUS permet que la nouvelle version d'une procédure, utilise une signature différente de l'ancienne. Pour garder la compatibilité entre les différentes procédures de l'application et la nouvelle version, le concept des *inter-procédures* a été introduit. Une inter-procédure est une procédure, écrite par le programmeur, et destinée à remplacer l'ancienne version dans le même espace mémoire. La Figure 18 illustre l'interaction entre une procédure de l'application et une nouvelle version d'une procédure. Comme le montre la Figure 18, la procédure P_i tente d'appeler la procédure P_j qui a été adaptée par une nouvelle version P_j' , ayant une signature différente. Une inter-procédure prend alors la place de la procédure P_j dans le même espace mémoire. Son rôle est de recevoir et de rediriger vers P_j' , les appels initialement envoyés vers P_j . L'inter-procédure est responsable, éventuellement, de récupérer le résultat retourné par P_j' , de le transformer en cas de besoin, avant de le transférer à P_i .

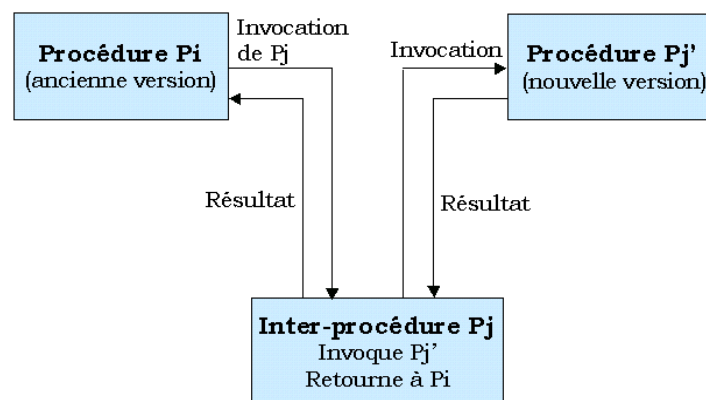


Figure 18. Utilisation des inter-procédures

5.2.3 Architecture du système

Comme illustré par la Figure 19, PODUS comporte deux composants principaux : un *shell* et un *processeur d'adaptation*. Le shell (*ush*) fournit un ensemble de commandes, permettant à l'utilisateur de charger, d'exécuter et d'adapter dynamiquement les procédures d'une application. Le processeur d'adaptation (*pup*) contrôle le processus dans lequel s'exécute l'application. Il fournit aussi les primitives nécessaires pour réaliser l'adaptation. Le processeur d'adaptation reçoit et exécute les commandes, envoyées à travers le shell. Il renvoie éventuellement des notifications.

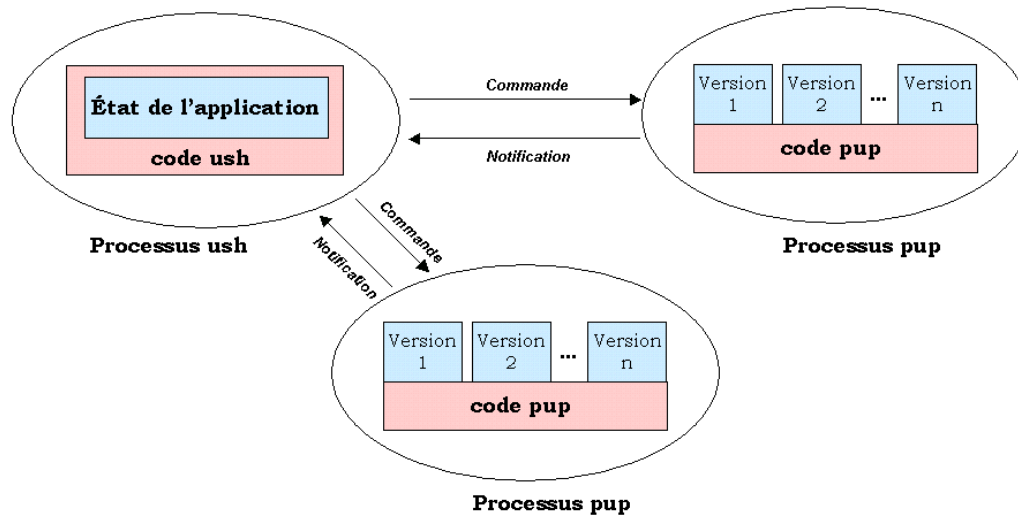


Figure 19. Architecture de PODUS

L'utilisateur peut lancer l'adaptation à tout moment. Une fois la requête d'adaptation reçue, le système commence d'abord par déterminer l'ensemble S des procédures actives. Une procédure P_i ne peut être adaptée que si elle ne fait pas partie de l'ensemble S . Une fois adaptée, la procédure P_i est marquée comme étant une nouvelle version.

Les procédures actives sont déterminées en parcourant la pile d'exécution et le graphe d'appel entre les procédures. Une telle information permet de déterminer le moment où une procédure peut être adaptée. Cependant, il n'est pas possible dans tous les cas de déterminer l'ensemble des procédures actives, car les appels des procédures peuvent dépendre d'un facteur externe (application de contrôle par exemple). C'est ce que les concepteurs appellent *dépendances sémantiques*, par opposition aux *dépendances syntaxiques*, reliant directement les procédures entre elles. Les dépendances syntaxiques peuvent être déterminées à partir du code de l'application. En contre partie, les dépendances sémantiques ne peuvent pas être déterminées automatiquement. L'utilisateur doit les spécifier, explicitement, avant de lancer l'adaptation. Quand une procédure est adaptée, le système adapte automatiquement toutes les procédures, syntaxiquement ou sémantiquement liées à la procédure adaptée.

□ Exemple d'utilisation

Imaginons que l'on a dans une application une procédure *trier_Entiers* qui trie des entiers. On souhaite, par exemple, manipuler des réels au lieu des entiers. Il faut donc adapter la procédure *trier_Entiers* par une procédure qui fait le tri des réels (*trier_Reels*). Puisque la spécification de la procédure a changé, une inter-procédure est nécessaire. Le code de l'inter-procédure peut avoir la forme suivante :

```
interprocedure trier_Entiers(data : array of integer);
var
  real_data : array of real;
{
  real_data := convert_to_real(data);
  trier_Reels(real_data);
  data := convert_to_integer(real_data);
}
```

L'inter-procédure précédente est nécessaire pour que les procédures existantes puissent continuer à s'exécuter normalement. Une procédure qui tente d'appeler *trier_Entiers* tombe sur l'inter-procédure qui la remplace. Cette dernière transforme d'abord le tableau des entiers, reçu en paramètres, en un tableau de réels. Elle appelle, ensuite, la procédure de tri des réels (*trier_Reels*). Enfin, elle retourne le résultat à l'appelant, après l'avoir transformé.

5.2.4 Discussion

PODUS est un système qui supporte l'adaptation dynamique des procédures. Il ne se contente pas seulement d'assurer l'adaptation de l'implémentation des procédures, mais il fournit en plus, un mécanisme qui permet d'adapter même leur spécification (le mécanisme des inter-procédures). Une procédure ne peut être adaptée que si elle n'est pas active. Si une application n'est pas bien modélisée, ou si une procédure doit s'exécuter pour longtemps, l'adaptation risque d'être fortement retardée.

Les inter-procédures constituent un bon moyen pour l'adaptation transparente des applications. En contre-partie, leur invocation systématique, à chaque appel, peut jouer un rôle important dans la dégradation des performances du système. Ceci est d'ailleurs le problème des techniques d'interposition en général (aspects, proxys, contenaires...).

6 SOLUTIONS DE RECONFIGURATION MODULAIRES

Plusieurs solutions de reconfiguration considèrent le module comme unité de modification. Un tel choix est justifié par plusieurs raisons :

- les langages de programmation, utilisés pour développer les applications dynamiquement adaptables, sont de nature modulaire,

- le plus souvent, dans ces langages, le module constitue l'unité d'abstraction et d'encapsulation des structures de données, et des procédures qui les manipulent,
- le module constitue également, le plus souvent, l'unité de compilation séparée.

Dans la sous-section suivante, nous présentons Polyolith, un système de reconfiguration modulaire, que nous avons étudié dans cette thèse.

6.1 Polyolith

Polyolith [Pur94] est un système développé à l'université de Maryland. Sa première version a vu le jour en 1985. Il définit un langage de programmation, modulaire, basé sur le langage C. Polyolith propose des solutions pour porter plus facilement, une application développée pour un environnement, vers un autre environnement. Dans une application distribuée, le développeur doit s'occuper des unités qui encapsulent le code applicatif, mais il doit aussi s'occuper du contexte d'utilisation de ce code. Ceci inclut par exemple la communication entre les unités, le *marshalling* des données avant de les envoyer sur le réseau, la gestion des erreurs de communication, l'adaptation de l'application en cas de besoin, etc. Les unités de l'application contiennent ainsi, à la fois, du code fonctionnel et du code non-fonctionnel, ce qui complique le développement et réduit les possibilités de réutilisation. L'idée qui est derrière Polyolith est de proposer une approche qui permet au développeur de s'occuper uniquement de la logique applicative, le reste est pris en charge automatiquement par le système.

6.1.1 Objectifs du système

Pour répondre aux besoins que nous avons décrits dans le paragraphe précédent, Polyolith a fixé plusieurs objectifs :

- Les modules et l'architecture de l'application doivent être indépendants. Les développeurs doivent avoir la possibilité d'exprimer quels modules utiliser dans l'architecture, sans prendre en considération leurs détails d'implémentation. Parallèlement, l'implémentation des modules doit être indépendante de l'architecture dans laquelle les modules seront utilisés.
- La topologie de l'application doit être indépendante de l'implémentation des modules. Il faut avoir la possibilité d'exprimer, sur quelle machine les modules doivent être exécutés, sans modifier leur implémentation.
- Il faut pouvoir exprimer, comment les modules interagissent, indépendamment de leur implémentation.

Pour répondre à ces objectifs, Polyolith fournit un langage d'interconnexion des modules (MIL) [FH+93] pour exprimer l'architecture des applications. Ce langage déclaratif, est indépendant de tout langage de programmation. La sous-section suivante donne une courte description de ce langage.

6.1.2 Langage d'interconnexion de modules (MIL)

Le rôle du MIL est d'exprimer et d'organiser l'architecture des applications. La construction de base du langage est de la forme suivante :

```
module nom { déclarations }
```

Cette construction permet de déclarer un module avec un nom spécifique. Les déclarations décrivent la vue externe du module en termes de ressources fournies et requises. L'exemple suivant montre comment les modules peuvent être déclarés.

```
module main {
  use interface lookup : PATTERN=string
}

module demo {
  define interface lookup : PATTERN=string
    : RETURNS=↑{ string ; integer }
}
```

L'exemple précédent montre la déclaration de deux modules : *main* et *demo*. Le module *main* déclare qu'il utilise une interface nommée *lookup*. L'expression *PATTERN=string* signifie que l'interface *lookup* a un argument qui s'appelle *PATTERN* de type *string*. Il est à noter que le terme *interface*, au sens de Polyolith, correspond au terme *méthode* dans la terminologie orientée objet. Le deuxième module défini dans l'exemple (*demo*) déclare qu'il fournit une interface nommée *lookup*, ayant un argument de type *string*, et retournant un tableau de couples, de types *string* et *integer* respectivement.

Pour exécuter l'application, il faut définir un troisième module qui regroupe les deux premiers. Ce module joue alors le rôle d'une glue, qui assemble les deux autres modules, pour former une application exécutable. Le code de ce module a la forme suivante :

```
module {
  tool main
  tool demo
  bind main.lookup demo.lookup
}
```

L'instruction *tool* permet d'instancier les modules. L'instruction *bind*, quant-à-elle permet d'interconnecter les interfaces des deux instances.

Une application, spécifiée en MIL, se présente comme un graphe. Les nœuds du graphe correspondent aux modules et les arcs représentent les interconnexions entre les interfaces des modules. Pour contrôler la topologie de l'application, le MIL permet d'attacher des informations aux nœuds du graphe. Ces informations permettent d'exprimer, sur quelle machine (ou processeur), doit s'exécuter un module donné.

6.1.3 Le bus logiciel

Un bus est un *intergiciel* (middleware) [Ber96] qui définit les mécanismes d'interaction, et qui encapsule les détails de communication entre les modules. Le changement du mode d'interaction, peut être obtenu en changeant le bus. Aucune modification sur les modules n'est alors nécessaire. Pour construire un bus, principalement deux décisions doivent être prises :

- Comment les processus démarrés par le bus peuvent communiquer.
- Comment les requêtes, exprimées par un module développé dans un langage de programmation particulier, peuvent être traduites et mises en correspondance sur les méthodes abstraites du bus.

Polylith supporte plusieurs types de bus, développés dans des travaux indépendants [PSW91]. Un développeur d'applications ne doit pas normalement s'occuper des bus. Les décisions qui concernent l'interaction entre les modules développés et le bus logiciel, et les décisions qui concernent la création des binaires pour un bus particulier, sont prises en charge par un système de packaging fourni par Polylith. En plus de la gestion de la communication entre les modules, le bus est aussi responsable de la reconfiguration dynamique des applications. Plusieurs situations peuvent se présenter en fonction de la localisation des modules.

Si les modules doivent s'exécuter dans le même processus, et s'ils sont développés dans le même langage de programmation, le code binaire généré est identique à celui généré par un compilateur normal. La Figure 20, montre la communication entre deux modules dans le même processus.

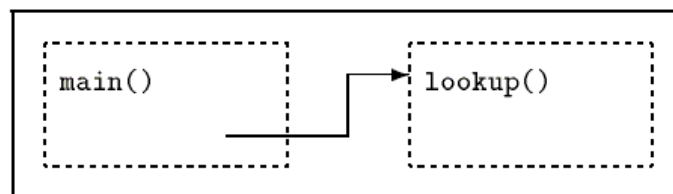


Figure 20. Communication directe entre modules

Si les modules sont développés dans deux langages de programmation différents, un mécanisme de traduction (stub) est nécessaire entre les deux. L'interaction se présente alors comme illustré par la Figure 21.

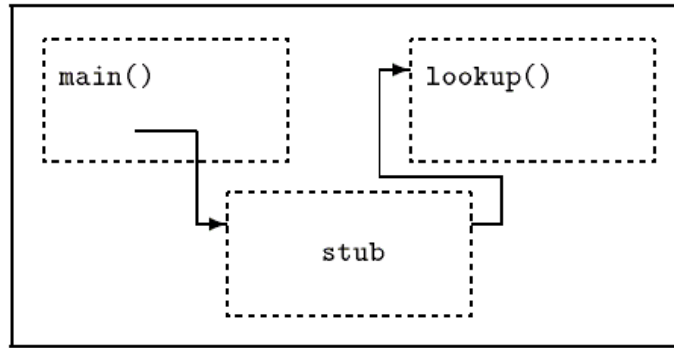


Figure 21. Communication indirecte entre modules

Dans le cas où les modules doivent s'exécuter sur des processeurs différents, distribués à travers le réseau, plusieurs processus doivent être gérés. Grâce à la description MIL de l'application, le bus initialise les processus nécessaires, sur les processeurs appropriés. Le schéma d'interaction est illustré par la Figure 22.

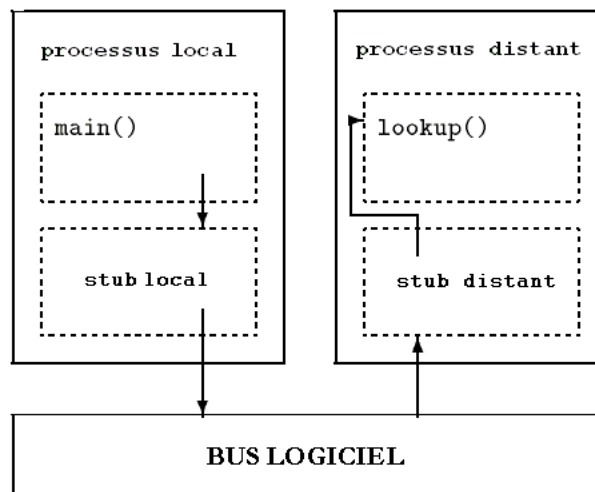


Figure 22. Communication entre modules distribués

6.1.4 Support de reconfiguration dynamique

Polyolith supporte trois types de modifications dynamiques. Ces modifications concernent l'architecture de l'application, la localisation des modules et les modules eux-mêmes. Le bus logiciel est responsable de réaliser dynamiquement ces modifications.

- **Reconfiguration de l'architecture** : altère l'architecture de l'application en ajoutant ou en supprimant des modules, ou en changeant les canaux de communication (interconnexions) entre les modules en cours d'exécution.

- ❑ **Reconfiguration de la topologie** : correspond à la migration des modules d'une machine à une autre.
- ❑ **Reconfiguration des modules** : implique le remplacement d'un module de l'application par un autre, en gardant la même architecture et le même emplacement du module.

Polyolith fournit aussi des méthodes qui permettent de bloquer les requêtes, envoyées sur les canaux de communication. Il prend également le soin de mettre en attente ces requêtes. Les canaux de communication sont gérés par le bus logiciel, leur blocage est transparent aux modules. La participation des modules n'est donc pas requise pour cette opération.

D'un autre côté, la reconfiguration topologique et modulaire nécessite la participation des modules, notamment pour la capture et la restauration de leur état d'exécution.

❖ **Transfert d'état**

Polyolith fournit un support automatique pour capturer et restaurer l'état d'un processus (en particulier la pile). En contre-partie, la sauvegarde et la restauration des segments mémoires (*heap*) et des données statiques, nécessite l'assistance du programmeur. Deux méthodes doivent être fournies par le module, pour la sauvegarde et la restauration de son état. Ces méthodes doivent s'appeler respectivement *encode()* et *decode()*. Le code de sauvegarde de l'état de la pile avant la reconfiguration, et celui de sa restauration après la reconfiguration, est généré automatiquement par le compilateur. La sauvegarde de toute autre donnée, doit être programmée à la main.

L'encodage de l'état d'un module se fait dans une représentation abstraite, indépendante de la machine. Ceci permet de faire migrer le module, vers une autre machine, même avec une architecture différente.

❖ **Points de reconfiguration**

La reconfiguration ne peut pas s'effectuer à n'importe quel moment. Le programmeur a la possibilité de spécifier à l'avance les points auxquels la reconfiguration est possible (points de reconfiguration). Ces points doivent correspondre à des états cohérents de l'exécution des modules. Ils sont spécifiés dans le code même des modules.

❖ **Algorithme de reconfiguration**

Avant d'effectuer la reconfiguration, le système bloque d'abord les canaux de communication du module, et lui demande de sauvegarder son état. Le module n'est pas obligé de s'arrêter et de réagir instantanément. Il peut continuer son exécution jusqu'à aboutir à un état cohérent, et par conséquent, à un point de reconfiguration. L'algorithme de reconfiguration peut être résumé par les étapes suivantes :

- ❑ bloquer les canaux de communication du module, pour l'isoler de l'application,
- ❑ demander au module d'encoder son état,

- ❑ le module attend le prochain point de reconfiguration pour réagir,
- ❑ le système crée le nouveau module, l'initialise à partir de l'état encodé auparavant et ensuite le connecte à l'application,
- ❑ le système débloque les canaux de communication.

6.1.5 Discussion

Nous avons présenté dans cette section Polyolith, un exemple de solution de reconfiguration modulaire. Les applications peuvent être spécifiées en utilisant un langage d'interconnexion de modules (MIL). Ce langage permet, en particulier, de séparer l'implémentation des modules, de l'architecture dans laquelle ils doivent être utilisés. Les développeurs doivent se préoccuper uniquement de la logique applicative, délégrant ainsi au système les détails de communication, de gestion des erreurs et de reconfiguration. Polyolith fournit un bus logiciel qui se charge de ces détails, et qui permet à des modules, développés dans des langages de programmation différents, d'interagir.

Le bus logiciel se charge de la reconfiguration dynamique des applications. Les modules peuvent être ajoutés, supprimés et remplacés dynamiquement. Le bus est également responsable de modifier les interconnexions entre les modules, de bloquer les canaux de communication et d'assurer la préservation des requêtes envoyées sur les canaux bloqués. Polyolith adresse particulièrement les applications distribuées. Il supporte la migration des modules, d'un site d'exécution à un autre.

L'état d'exécution des modules est aussi considéré dans Polyolith. Le transfert d'état est un problème très complexe; il est cependant nécessaire de le prendre en considération, pour assurer un minimum de cohérence dans la reconfiguration. L'une des questions les plus délicates, est la spécification de ce qu'est exactement un état. La réponse à cette question doit être faite, ou au moins guidée par le développeur. Polyolith assure une certaine automatisation dans le transfert de l'état (état de la pile). Le développeur doit participer en fournissant des opérations pour l'encodage et le décodage du reste de l'état.

La reconfiguration dans Polyolith doit être décidée par un administrateur humain, et déclenchée manuellement. Ceci constitue une limitation si les applications adressées nécessitent beaucoup d'interventions de reconfiguration. Avec l'utilisation de mécanismes d'observation et de raisonnement automatiques, cette limitation aurait pu être évitée.

7 SOLUTIONS DE RECONFIGURATION ORIENTEES OBJET

7.1 Les classes dynamiques de Java

Des travaux ont été réalisés à l'université de Californie, pour augmenter la capacité des applications Java, à supporter l'évolution dynamique [MP+00]. Même si Java fournit

plusieurs mécanismes puissants comme l'héritage, les interfaces et la liaison dynamique, il ne supporte pas l'évolution dynamique du code. Dans l'approche proposée, les applications Java peuvent évoluer, sans être arrêtées, en remplaçant dynamiquement leurs classes.

Une classe Java a un cycle de vie qui peut prendre l'un des trois états : statique (non chargée), chargée et active. Ces états sont illustrés par la Figure 23.

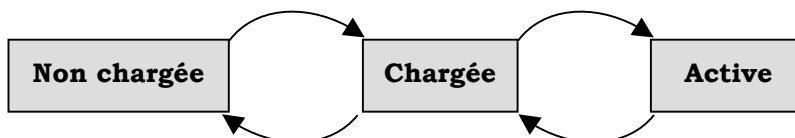


Figure 23. Cycle de vie d'une classe Java

Une classe est considérée comme statique si elle n'est pas encore chargée dans la machine virtuelle de Java (JVM). Lorsqu'une référence à une classe est rencontrée, la classe est chargée. Enfin, lorsque la classe est instanciée ou lorsqu'une invocation statique est faite (une méthode statique de la classe est en exécution), la classe entre alors dans l'état actif. L'approche proposée vise le remplacement des classes actives. Le remplacement doit prendre en compte les instances existantes de la classe, et doit préserver la cohérence de l'application. Pour proposer une solution générale, et pour supporter le remplacement de n'importe quelle classe Java, la JVM a été modifiée. En particulier, un chargeur de classes évolué a été proposé, et des modifications ont été introduites à quelques structures de données de la JVM. Le chargeur de classes évolué permet de définir plusieurs fois la même classe, dans la même application.

7.1.1 Validité des types ("Type safety")

Remplacer une classe C peut avoir un grand impact sur la validité des types (Type Safety) [OGC00]. Le remplacement peut concerner le type ou l'implémentation de la classe. Des méthodes peuvent être ajoutées, modifiées ou supprimées. Des attributs peuvent être ajoutés ou supprimés. Le type peut être par conséquent modifié. De la même manière, ajouter ou supprimer des super-classes ou modifier la liste des interfaces implémentées par la classe, modifie la liste des types dans lesquels une instance de C peut être affectée ou convertie (*castée*). Dans l'exemple suivant, la classe D contient une référence vers la méthode m de la classe C . Si la nouvelle version de la classe C ne contient plus la méthode m , une violation est constatée, et l'application est alors dans un état incohérent.

```
// classe 'C' initiale
public class C {
    public void m() { };
}

// classe 'C' modifiée
public class C {
}

// classe 'D' dépend de 'C'
public class D {
    public void p() {
        C c = new C();
        c.m();
    };
}
```

En général, deux catégories de violation de type peuvent être distinguées, dans une application Java. Les *violations statiques* et les *violations dynamiques*. Le premier type de violation peut être détecté à la compilation. Par exemple, lorsqu'une méthode dans une classe *C*, référence un attribut *D.x*, et que la classe *D* ne contient pas un attribut nommé *x*, la référence vers *x* est invalide. Le compilateur est capable de détecter cette violation. Une violation dynamique apparaît à l'exécution. Elle échappe au contrôle du compilateur, et peut être observée si une référence doit être liée à une valeur d'un type différent. Par exemple, supposons que *o* soit une instance de la classe *C*, et que *C* n'implémente pas l'interface *I*. Si *o* est converti à une variable de type *I*, comme le montre le code suivant, une violation de type dynamique est observée.

```
C o = new C();
I v;
v = (I) o;
```

La vérification dynamique des types, est réalisée par la JVM pour certaines opérations comme l'affectation ou la conversion.

7.1.2 Remplacement sûr des classes

L'approche a pour objectif de remplacer dynamiquement les classes. Des mécanismes sont alors nécessaires pour vérifier que la nouvelle version de la classe garantit la validité des types. La vérification des types, réalisée dynamiquement par la JVM, n'est pas suffisante. En effet, le remplacement d'une classe peut avoir lieu après l'opération d'affectation, et par conséquent, les erreurs éventuelles de typage ne peuvent pas être détectés par la JVM. Pour permettre le remplacement d'une classe par n'importe quelle autre classe, sans restriction, un mécanisme est nécessaire pour contrôler la validité des types, à chaque fois qu'un objet est référencé. Le contrôle systématique des types à chaque fois qu'un objet est référencé, permet d'éviter les erreurs de type qui échappent au contrôle de la JVM. En contre partie, un tel mécanisme systématique, est très coûteux en termes de performances.

Pour assurer la validité des types, sans causer la dégradation des performances, la solution choisie, dans cette approche, consiste à placer des contraintes sur les nouvelles classes, destinées à remplacer les anciennes. Ceci en réduisant l'ensemble des modifications autorisées. Les contraintes imposées permettent d'éliminer la probabilité des violations dynamiques. Par conséquent, l'approche se contente de définir des mécanismes pour la vérification statique des types, avant le remplacement des classes.

Les modifications doivent respecter les deux conditions suivantes :

- ❑ Un attribut ou une méthode, ne peut être supprimé d'une classe C , que si aucune autre classe P ne dépend de cet attribut ou de cette méthode.
- ❑ Un type de C (super-classe ou interface implémentée par C) ne peut être supprimé, que si aucune autre classe P ne dépend de ce type.

De nouvelles méthodes ou de nouveaux attributs peuvent être ajoutés à la nouvelle classe sans restriction. De nouvelles super-classes ou interfaces peuvent aussi être ajoutées. Il est également possible de modifier le corps des méthodes existantes.

Dans la suite de cette section, nous présentons brièvement le chargeur de classes standard de Java. Puis, nous expliquons comment il a été étendu pour supporter le remplacement dynamique des classes.

7.1.3 Chargeur de classes de Java

La JVM établit la résolution des références des classes, en exécution, en utilisant le mécanisme de chargeur de classes [Tra01]. Un chargeur est responsable de localiser la définition des classes (fichier .class), et de les charger dans la JVM. Une classe Java peut être ainsi identifiée par son nom et par son chargeur. La JVM définit un chargeur de classes *système* qui est le chargeur par défaut. Il est utilisé pour charger les classes systèmes et aussi les classes des applications. Les programmeurs peuvent définir des chargeurs de classes personnalisés, en étendant le chargeur par défaut. La JVM peut être amenée à appeler le chargeur de classes par défaut, ou un chargeur personnalisé, selon le cas.

Quand une classe C a une référence vers une classe P . La JVM demande d'abord au chargeur de C (chargeur particulier ou chargeur par défaut) de charger P (en appelant la méthode `loadClass()`). Si la classe P n'est pas trouvée, la requête de chargement est déléguée au chargeur parent, et ainsi de suite, jusqu'à ce que la requête soit reçue par le chargeur par défaut, parent de tous les autres chargeurs. Si en arrivant à la racine des chargeurs, la classe n'est pas trouvée, une exception est alors levée.

7.1.4 Chargeur de classes dynamiques

Le nouveau chargeur défini, étend le chargeur de classes par défaut. Deux nouvelles méthodes ont été ajoutées : `reloadClass()` et `replaceClass()`. L'interface du nouveau chargeur est la suivante :

```
public class DynamicClassLoader extends ClassLoader {
    public Class reloadClass(String newc);
    public final int replaceClass(String oldc, String newc);
}
```

La méthode `reloadClass()` permet de recharger une classe déjà chargée. La méthode `replaceClass()` quant-à-elle, recharge une nouvelle version d'une classe, et déclenche la mise à jour des instances existantes de cette classe. Ces deux méthodes sont basées sur plusieurs autres méthodes natives, qui interfacent avec les structures de données internes de la JVM.

Le nouveau chargeur nécessite le support de la JVM pour recharger la définition d'une classe, pour chercher et mettre à jour les classes qui en dépendent, et pour chercher et mettre à jour les instances de la classe rechargée.

7.1.5 Adaptation des instances

Lorsqu'une classe est remplacée, plusieurs stratégies peuvent être appliquées à ses instances :

- ❑ *Barrière de version* : le remplacement de la classe ne peut se faire que si toutes ses instances ont expiré. Dans ce cas, le problème de l'adaptation des instances ne se pose pas. Cependant, les possibilités de remplacement, ainsi que la flexibilité, sont fortement réduites.
- ❑ *Partitionnement passif* : les instances créées avant le remplacement ne sont pas affectées. Les instances créées après le remplacement, respectent la nouvelle définition de la classe. Plusieurs versions de la même classe doivent alors être maintenues simultanément, ce qui risque de causer des ambiguïtés.
- ❑ *Partitionnement actif* : il est similaire au partitionnement passif. La seule différence est que l'utilisateur a la possibilité de sélectionner quelle instance doit garder la définition initiale de la classe, et quelle instance doit être adaptée selon la nouvelle définition. Cette solution est très compliquée car elle nécessite le traitement des instances au cas par cas, ce qui n'est pas une tâche triviale.
- ❑ *Adaptation globale* : elle consiste à adapter toutes les instances de la classe remplacée selon sa nouvelle définition.

C'est cette dernière stratégie qui a été choisie et implémentée dans cette approche. Après le remplacement d'une classe, ses instances sont alors localisées dans le *tas*, et après les avoir verrouillées, le système procède à leur adaptation selon la nouvelle définition de la classe.

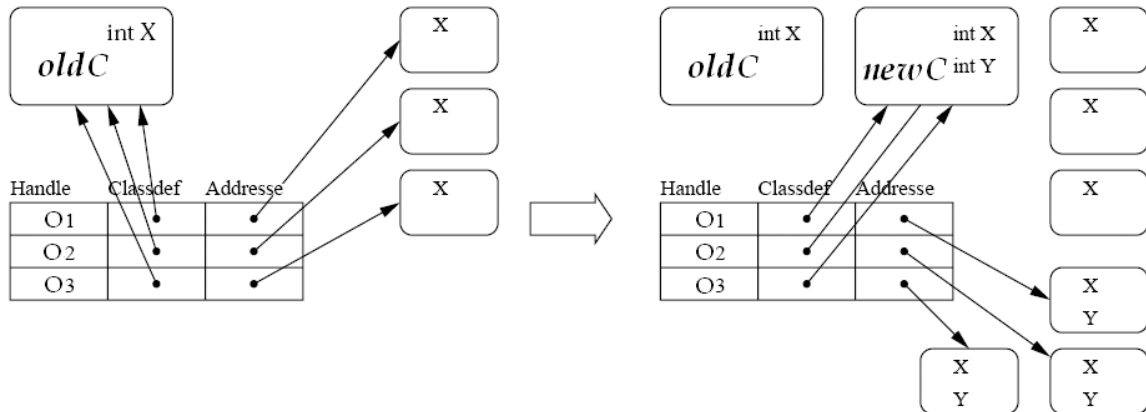


Figure 24. Adaptation des instances d'une classe

Comme le montre la Figure 24, l'adaptation consiste à créer pour chaque instance une nouvelle instance de la nouvelle classe, ensuite à copier les valeurs des attributs (X dans la figure) de l'ancienne instance vers la nouvelle, et enfin à changer les références vers les nouvelles instances (l'ancienne instance peut ainsi être libérée par le ramasse-miettes de Java).

Les classes dépendantes de la classe remplacée, y compris les sous-classes, doivent également être adaptées. Toutes les références vers l'ancienne classe, doivent pointer, après le remplacement, vers la nouvelle classe.

7.1.6 Discussion

La solution que nous avons présentée dans cette section, se situe dans la lignée des approches qui ont adressé le problème d'adaptation dynamique, avec l'objet comme granularité de base. La machine virtuelle de Java (JVM) a l'avantage de charger les classes, seulement si elles sont nécessaires. Cependant, une fois une classe chargée, elle ne peut plus être chargée à nouveau avec une nouvelle définition. Cette limitation constitue un obstacle devant l'évolution dynamique des applications.

Les classes dynamiques de Java, ont été proposées comme solution, pour dépasser les limitations de la JVM, et pour supporter le rechargement dynamique des classes déjà chargées. Nous avons montré, à travers cette section, la complexité d'une telle tâche, et les nombreux aspects qu'il faut prendre en compte pour la rendre possible.

Concrètement, la solution est basée sur la définition d'un chargeur de classes évolué, et sur quelques modifications au niveau de la JVM. Le chargeur de classes proposé définit, en particulier, une méthode qui permet de remplacer les classes déjà chargées. Le remplacement prend en considération les instances existantes de la classe, et assure leur adaptation en fonction de la nouvelle version de la classe. Il assure également l'adaptation des références que détiennent les autres classes, vers la classe remplacée. Les méthodes actives (les

méthodes en cours d'exécution) restent quand-à-elles, un problème délicat. Comme nous l'avons expliqué dans le chapitre précédent, l'une des techniques pour pouvoir remplacer les méthodes actives, est d'utiliser des points de reconfiguration, à placer explicitement dans le code. Ces points permettent de savoir à quel point exactement il faut remplacer l'objet ou la classe, et à quel point exactement il faut continuer l'exécution de la nouvelle version de la méthode. La solution que nous avons présentée dans cette section, ne supporte pas explicitement les points de reconfiguration. Par conséquent, il n'est pas possible, même en analysant la pile d'exécution, de trouver une correspondance entre deux points dans l'ancienne et dans la nouvelle méthode.

Comme dans le cas Iguana/J, les classes dynamiques de Java ont eu recours à la modification de la JVM. Une telle modification nécessite la maîtrise de beaucoup de détails techniques, et une connaissance profonde de la JVM. En effet le moindre détail ignoré peut avoir un grand impact sur les applications et peut causer leur déstabilisation. De plus, la solution reste figée pour une JVM unique, et son adaptation pour de nouvelles versions de la JVM, ayant des spécifications techniques différentes, n'est pas toujours garantie.

Un travail similaire a été réalisé pour le langage C++, dans une collaboration entre le laboratoire AT&T et le Dartmouth College (USA) [HG98]. L'objectif du projet était de développer une extension au dessus du langage C++, pour supporter l'ajout et la modification des fonctionnalités dynamiquement. Le choix de la classe (et par conséquent de l'objet) comme unité de remplacement, avait pour objectif la préservation de l'intégrité sémantique des applications, et le raisonnement homogène en termes d'objets. Après le remplacement d'une classe, la question de base concerne les instances de cette classe. Parmi les quatre stratégies que nous avons présentées précédemment, c'est la deuxième stratégie (partitionnement passif) qui a été adoptée. Cela signifie que les instances existantes continuent à s'exécuter normalement, jusqu'à leur fin naturelle, et les nouvelles instances sont créées selon la nouvelle définition de la classe. Cette stratégie est la plus simple à mettre en œuvre car elle ne nécessite aucune modification sur les instances existantes. Cependant, les problèmes qui peuvent résulter de la présence de deux versions de la même classe en mémoire, n'ont pas été clairement abordés par les concepteurs du système.

Le choix du partitionnement passif comme stratégie, a été motivé par le fait que l'ancienne et la nouvelle version de la classe, peuvent avoir des structures de données radicalement différentes. Par conséquent, le problème de transfert d'état de l'ancienne instance vers la nouvelle, d'une manière automatique, n'est pas solvable dans toute sa généralité.

Nous avons jusque là présenté plusieurs solutions de reconfiguration dynamique, adressant l'adaptation des applications en exécution, à différents niveaux de granularité. Dans la suite, nous nous concentrons sur des solutions orientées composant.

8 SOLUTIONS DE RECONFIGURATION ORIENTEES COMPOSANT

Dans les définitions que nous avons présentées dans le chapitre précédent, et qui concernaient le développement orienté composant, l'une des propriétés importantes que nous avons soulignée, était la substituabilité des composants. Nous avons aussi constaté, après l'analyse de quelques modèles de composants industriels, que la reconfiguration dynamique, qui comprend entre autres, le remplacement des composants en exécution, n'était pas un critère déterminant dans ces modèles.

D'un autre coté, plusieurs modèles de composants, académiques en général, ont été développés explicitement dans l'optique de supporter la reconfiguration dynamique. Nous avons étudié quelques uns de ces modèles. Les sections suivantes sont dédiées à leur présentation.

Certains travaux, parmi ceux que nous présentons, sont issus de la communauté des architectures logicielles [SG96, OM+98]. A partir du moment où les unités de base manipulées sont souvent les composants et les connecteurs qui les lient, nous analysons ces travaux au même titre que les modèles de composants, en nous concentrant sur l'aspect de reconfiguration dynamique.

8.1 K-Components

K-Components [DC01] est un projet développé au Distributed Systems Group au Trinity College de Dublin (comme Iguana). C'est un modèle de composants, destiné spécialement au développement des applications sensibles au contexte. Les composants sont spécifiés dans un langage nommé K-IDL, un sous ensemble du langage de définition d'interfaces (IDL-3) [IDL3], développé par l'OMG [OMG]. Le comportement architectural des applications est spécifié dans un langage séparé, nommé ACDL (langage de description des contrats d'adaptation). Le modèle est fortement basé sur la réflexion architecturale, que nous avons présentée dans le chapitre précédent. Toute application (correspondant au niveau de base dans une conception réflexive), développée en K-Components, possède alors un méta-niveau qui reflète son architecture.

8.1.1 Construction des composants

Les composants sont spécifiés en utilisant un langage de définition d'interfaces. Un composant peut explicitement déclarer qu'il fournit des interfaces, ou qu'il a besoin d'autres interfaces, fournies par d'autres composants. Les clauses *emits* et *consumes*, qui permettent de déclarer respectivement des sources et des puits d'événements dans IDL-3, sont utilisées pour spécifier les événements d'adaptation, reçus ou émis par le niveau de base. Les squelettes des composants sont générés par le compilateur de l'IDL-3. Les composants peuvent être primitifs ou composites. Les composants composites détiennent leur propre méta-modèle d'architecture.

En plus des interfaces fonctionnelles, un composant peut aussi avoir des interfaces de reconfiguration. Ces interfaces, utilisées par le protocole de reconfiguration, permettent par exemple, de traverser le graphe de configuration (représentant l'application), de changer les connexions, et de lire et écrire l'état d'exécution des instances de composants.

8.1.2 Les connecteurs

Dans une application, les composants sont reliés par des connecteurs. Ces connecteurs sont considérés comme des entités de première classe au même titre que les composants. Ils sont caractérisés par des interfaces fournies et requises. Ils sont générés à partir de la description IDL-3 des composants. Comme les composants, chaque connecteur fournit une interface de reconfiguration. Cette interface définit principalement deux opérations (*link_component* et *unlink_component*) qui permettent de connecter et de déconnecter les connecteurs et les composants.

8.1.3 Le méta-modèle de l'architecture

Le méta-modèle de l'architecture réifie les propriétés architecturales des applications. Comme toute approche réflexive, la question fondamentale qui se pose, concerne la désignation des entités de base qu'il faut réifier, et par conséquent représenter au méta-niveau. K-Components, comme toutes les approches qui tentent d'assurer un développement propre, préconise la séparation des préoccupations. Cela signifie que le code d'adaptation doit être séparé du code de l'application. Plusieurs questions sont alors posées : comment le code d'adaptation doit être représenté, quels sont les mécanismes nécessaires pour intégrer le code d'adaptation dans l'application et comment faire interagir les deux codes, comment gérer l'intégrité de l'application durant et après la reconfiguration dynamique, etc. ?

8.1.3.1 Le graphe de configuration

L'architecture d'une application est réifiée comme un graphe typé. La Figure 25 montre un exemple de graphe. Les nœuds du graphe représentent les composants. Chaque nœud est caractérisé par l'interface du composant qu'il représente, et étiqueté par son implémentation. Les arcs du graphe représentent les connecteurs entre les composants. Un arc peut avoir des propriétés de reconfiguration qui le caractérisent. Une propriété exprime, par exemple, la capacité du connecteur de changer son protocole de communication, ou son attachement à un ensemble d'intercepteurs, etc. Le nœud racine du graphe a une propriété particulière : il représente le point d'entrée de l'application. Il correspond, par exemple, à la procédure `main()` dans une application C++ ou Java.

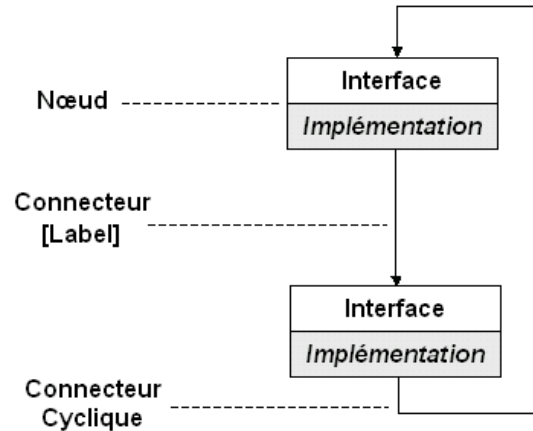


Figure 25. Exemple de graphe

Le graphe supporte les cycles. Cependant, ces cycles doivent être mentionnés explicitement comme des connecteurs cycliques. Le stockage et la gestion du graphe de configuration est de la responsabilité d'un élément spécifique du méta-niveau : *le gestionnaire de configuration*.

8.1.3.2 Identification de l'architecture des applications

Nous avons expliqué que la description des composants doit être donnée explicitement dans un sous-ensemble du langage IDL-3. La question qui reste posée concerne l'architecture des applications. Au lieu de forcer le développeur à décrire explicitement l'architecture dans un ADL (langage de description d'architecture) [Gar00], le système se charge de la déterminer automatiquement. Le calcul de l'architecture, et par conséquent du graphe de configuration, se fait à partir des fichiers sources C++, contenant l'implémentation des composants. Le gestionnaire de configuration analyse les fichiers sources C++, de la même manière que le compilateur C++ génère le graphe de dépendances. Chaque fichier est analysé, pour déterminer de nouvelles dépendances, donc de nouveaux connecteurs. L'analyse se termine en arrivant aux composants feuilles (ils n'ont pas de dépendances).

8.1.4 Reconfiguration dynamique

La reconfiguration dynamique est modélisée comme une transformation conditionnelle de graphes. La transformation est explicitement spécifiée dans des *contrats d'adaptation*. La manipulation du graphe est guidée par un ensemble de règles. Ces règles définissent quant et comment le graphe doit être transformé.

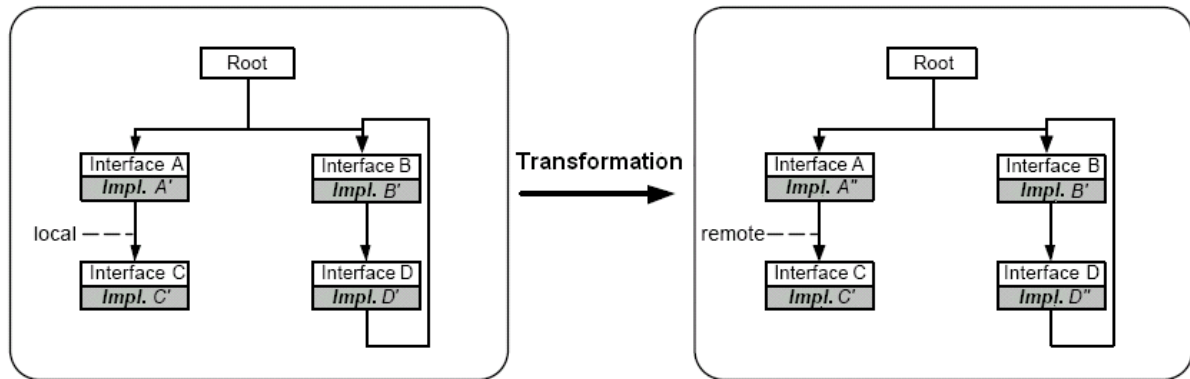


Figure 26. Exemple de transformation de graphe

Les nœuds et les arcs, qui représentent respectivement les composants et les connecteurs, constituent la partie préservée du graphe, qui ne peut pas être affectée par la transformation. Uniquement les propriétés, attachées aux composants et aux connecteurs peuvent être modifiées durant la transformation. Par conséquent, le modèle de reconfiguration dynamique est limité aux deux opérations suivantes :

- ❑ Remplacement d'une implémentation d'un composant par une autre implémentation : la même interface doit être maintenue.
- ❑ Modification des stratégies du connecteur.

Le système ne supporte alors pas l'ajout de nouveaux composants, la suppression de composants existants et même le changement dynamique des connexions entre composants.

La Figure 26 montre un exemple de reconfiguration dynamique. L'implémentation du composant *A*, ainsi que celle du composant *D*, ont été modifiées. Le mode de communication du connecteur entre *A* et *C* a été également modifié.

Le protocole de reconfiguration se décompose en plusieurs étapes :

- ❑ Appeler l'opération de reconfiguration pour déclencher la transformation du graphe.
- ❑ Identifier la nouvelle configuration, qui doit être obtenue. Ensuite envoyer un message pour geler les composants et les connecteurs qui sont impliqués par la transformation.
- ❑ Réaliser la transformation du graphe. La transformation consiste soit à changer les stratégies des connecteurs, soit à remplacer l'implémentation des composants.

Ce dernier cas se résume en plusieurs points :

- déconnecter le composant dont l'implémentation doit être remplacée,

- créer le nouveau composant, suivant la nouvelle implémentation,
 - transférer l'état de l'ancien vers le nouveau composant. Comme nous l'avons expliqué précédemment, chaque composant doit fournir deux méthodes qui permettent de lire et d'écrire son état,
- La dernière étape de la reconfiguration consiste à envoyer un message aux composants et aux connecteurs qui ont été gelés, pour reprendre leur exécution normale.

8.1.5 Les contrats d'adaptation

Les contrats d'adaptation sont utilisés pour spécifier les transformations d'un graphe de configuration. Ils peuvent être utilisés pour fournir la logique d'adaptation qui permet de construire des applications auto-reconfigurables. Les contrats d'adaptation appellent les opérations de reconfiguration sur le méta-modèle. La reconfiguration effective des applications concrètes, résulte de la reconfiguration du méta-modèle. Les contrats sont représentés, en exécution, par des objets au méta-niveau. Ils sont déployés et gérés par le gestionnaire de configuration. L'exemple suivant illustre la déclaration d'un contrat d'adaptation.

```
configuration contract MonContrat
{
  transaction {
    if (C.EventX) {
      replace("C", "C'");
      replace("B", "B'");
      throw B.EventY;
    }
  }
}
```

Un contrat d'adaptation contient une série de règles conditionnelles, qui servent à transformer le graphe de configuration. Il permet d'exprimer des contraintes pour le bon fonctionnement de l'application. Les conditions associées aux règles portent sur les éléments de l'application (composants et connecteurs). Le mécanisme *d'événements d'adaptation* est utilisé pour remonter les informations provenant de ces éléments. Dans l'exemple précédent, le contrat est concerné par un événement, nommé *EventX*, issu du composant *C*. A l'occurrence de cet événement, les implémentations des deux composants *C* et *B* sont remplacées. Egalement, un événement nommé *EventY*, provenant du composant *B*, est déclenché.

8.1.6 Discussion

Dans cette section, nous avons présenté K-Components, un modèle de composants qui traite la reconfiguration dynamique comme un aspect de premier plan. Il est destiné au

développement des applications sensibles au contexte, ce qui explique la position importante qu'il réserve à l'adaptabilité des applications. K-Components exploite principalement la réflexion architecturale pour supporter la reconfiguration dynamique des applications. Il permet d'assurer grâce à la réflexion, une séparation des préoccupations entre l'application concrète qui fournit les services de base, et entre le système qui est responsable de son adaptation.

❖ Automatisation

L'auto-reconfiguration est l'une des caractéristiques importantes, mise en avant par K-Components. Les besoins des utilisateurs en termes de reconfiguration dynamique, peuvent être exprimés dans des contrats d'adaptation explicites. Le système se charge au bon moment, et sous les bonnes conditions, de répondre à ces besoins, en appliquant les contrats.

Egalement, l'utilisation des événements d'adaptation a l'avantage de découpler les contrats d'adaptation (au méta-niveau) des composants et des connecteurs (au niveau de base). Les événements d'adaptation sont un moyen indispensable pour créer un système auto-reconfigurable, dans lequel le code de l'adaptation et le code fonctionnel sont séparés. Cette séparation permet également de manipuler facilement les contrats d'adaptation, et permet de les charger et de les décharger dynamiquement.

La découverte de l'architecture des applications est la responsabilité d'un outil. Cet outil analyse les fichiers sources C++ de l'application, et exploite la description IDL-3 des composants. Ceci a l'avantage de décharger le programmeur de cette tâche. Cependant, les concepteurs de K-Components n'expriment pas clairement si l'outil est complètement automatisé, ou s'il nécessite une assistance du programmeur. Egalement, il n'est pas clair s'il y a des contraintes sur le code C++ à écrire. Par exemple, qui est responsable d'instancier les composants, qui est responsable d'établir les connexions, etc ?

❖ Opérations de reconfiguration

Les opérations de reconfiguration sont limitées au remplacement de l'implémentation des composants, et aux changements des propriétés des connecteurs. Il n'est pas possible de faire évoluer dynamiquement les applications, en ajoutant de nouveaux composants. Il n'est également pas possible de retirer des composants de l'architecture, de supprimer ou de rediriger des connecteurs. Le problème de transfert d'état est très complexe, il n'a pas été suffisamment abordé dans K-Components. Le programmeur doit prévoir des opérations pour lire et écrire l'état d'un composant, il n'est pas clairement exprimé comment cet état doit être exprimé, et dans quel format il doit être représenté. Le système ne prévoit pas, non plus, des facilités pour aider les programmeurs à définir les opérations de transfert d'état, à partir de descriptions de haut niveau par exemple.

❖ Règles d'adaptation

Dans les règles d'adaptation, les conditions portent uniquement sur les noms des événements reçus. Parfois, il est souhaitable de pouvoir exprimer des conditions complexes

sur les propriétés des composants ou des connecteurs. Une solution serait d'attacher aux nœuds, qui représentent les composants dans le graphe de configuration, des méta-données qui caractérisent les propriétés qui sont jugées pertinentes. Par conséquent, des conditions plus riches peuvent être exprimées au niveau des règles. Il est également intéressant de pouvoir exprimer des événements en provenance des ressources matérielles (bande passante, mémoire, processeur, disque, etc.), au lieu de se limiter uniquement aux événements envoyés par les composants et les connecteurs.

❖ Gestion des messages

K-Components n'aborde pas clairement le problème des messages envoyés pendant la reconfiguration. En effet, si nous partons de l'hypothèse que les composants ne sont pas conscients de la reconfiguration, et que cette dernière peut avoir lieu d'une manière transparente, il est nécessaire d'assurer la préservation des messages envoyés pendant la reconfiguration, ce qui constitue un problème délicat.

8.2 DCUP

SOFA/DCUP (Dynamic Component UPdating) [PBJ97] est un modèle de composants, développé dans le cadre du projet SOFA [SOFA], à l'université Charles (Prague). DCUP est un modèle hiérarchique, supportant le remplacement dynamique des composants. Parmi les objectifs majeurs, tracés par ses concepteurs, nous pouvons citer la transparence de l'adaptation dynamique, la prise en compte de l'état d'exécution pendant l'adaptation, et le support de téléchargement automatique et dynamique des composants. Une application DCUP est une hiérarchie de composants interconnectés. C'est un composant de gros grain, construit par assemblage de sous-composants à l'aide d'un mécanisme de composition.

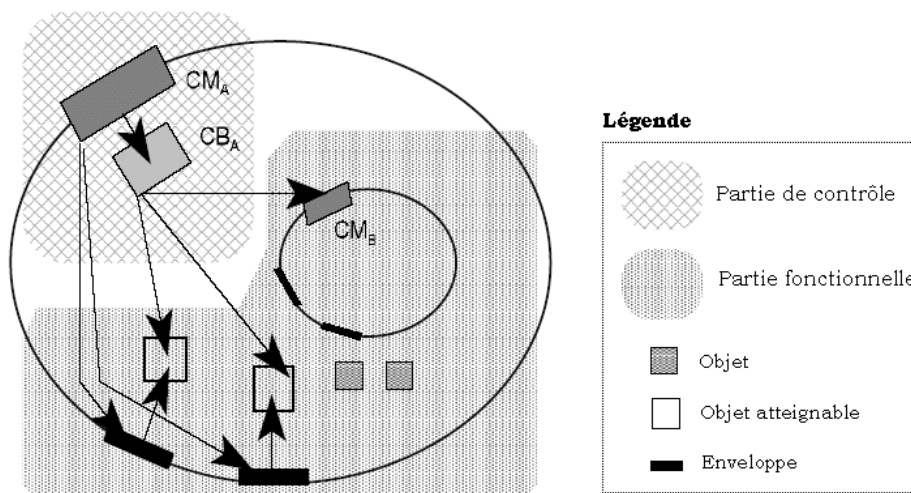


Figure 27. Structure d'un composant DCUP

8.2.1 Structure d'un composant

La Figure 27 présente la structure typique d'un composant DCUP. Un composant est caractérisé par un ensemble d'objets (instances de classes). Il fournit deux types d'opérations :

- ❑ Les opérations de contrôle : elles permettent de contrôler le composant et de gérer son cycle de vie.
- ❑ Les opérations fonctionnelles : elles caractérisent les services fonctionnels que le composant est responsable de fournir.

Du point de vue de l'adaptation, un composant est constitué d'une partie permanente et d'une partie remplaçable, comme le montre la Figure 28.

Chaque composant contient deux objets particuliers : *le gestionnaire du composant (CM)* et *le constructeur du composant (CB)*. Le premier implémente l'interface *ComponentManagerInterface*. Il constitue le cœur de la partie permanente. Le constructeur du composant, quant à lui, implémente l'interface *ComponentBuilderInterface*. Il constitue la clé de la partie remplaçable.

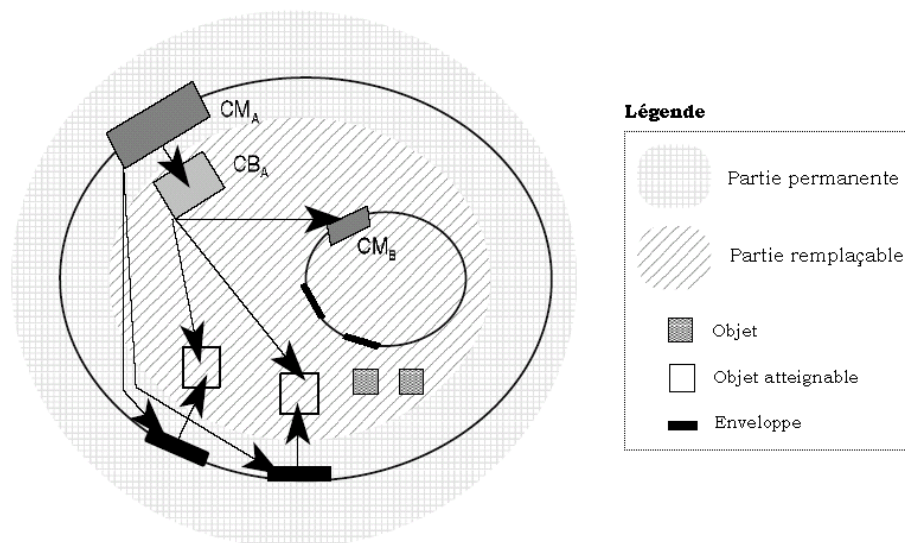


Figure 28. Partie permanente et partie remplaçable

En plus de ces deux objets particuliers, un composant contient également des objets fonctionnels et des sous-composants. Chaque sous-composant contient à son tour un gestionnaire et un constructeur. Un objet fonctionnel doit implémenter l'interface *Reachable* pour se déclarer atteignable. Dans ce cas, une enveloppe est associée à cet objet. Comme le montre la Figure 28, l'enveloppe, qui appartient à la partie permanente, permet au composant père d'accéder à l'objet.

8.2.2 Adaptation dynamique

L'adaptation dynamique d'un composant revient à changer sa partie remplaçable par une nouvelle version. Le cycle de vie d'un composant passe par plusieurs versions PR_1, PR_2, \dots, PR_n , où PR_i est la i -ème version de la partie remplaçable du composant. A toute version PR_i de la partie remplaçable, est associée une version CBI du constructeur.

Deux opérations sont assurées par le constructeur :

- ❑ `onArrival()` : elle se charge de construire le composant. Elle crée les objets fonctionnels et les initialise, éventuellement, à partir de l'état externalisé par les objets de l'ancienne version. Cette opération crée également les sous-composants de la partie remplaçable, et établit les interconnexions nécessaires à l'intérieur de cette partie.
- ❑ `onLeaving()` : elle arrête tous les *threads* [WO97] en exécution dans la partie remplaçable et sauvegarde l'état des objets considérés comme "*importants*". Elle finit par détruire tous les objets fonctionnels et tous les sous-composants de la partie remplacée.

L'adaptation d'un composant est contrôlée par son gestionnaire. Ce dernier, commence d'abord par appeler l'opération `onLeaving()` du constructeur de l'ancienne version. Il charge ensuite le constructeur associé à la nouvelle version. Enfin, il demande au nouveau constructeur de construire la nouvelle version du composant, en appelant l'opération `onArrival()`. Ce processus d'adaptation est regroupé dans la méthode `updateComponent()`, illustrée par le code suivant :

```
public class CManager {  
  
    updateComponent(...) {  
        oldBuilder.onLeaving();  
        newBuilder = new ComponentBuilder();  
        newBuilder.onArrival();  
    }  
}
```

Le processus d'adaptation d'un composant comprend plusieurs étapes :

- ❑ Le gestionnaire du composant bloque toutes les communications entrantes, et envoie une requête de mise à jour au constructeur du composant.
- ❑ Le constructeur du composant arrête l'exécution de la partie remplaçable. Il capture l'état du composant, puis détruit la partie remplaçable.
- ❑ Le gestionnaire du composant télécharge et instancie la nouvelle version du constructeur, puis il détruit l'ancienne.

- ❑ Le constructeur du composant crée la nouvelle structure du composant. Cela consiste à créer les objets fonctionnels et les sous-composants. L'état de l'ancienne version, précédemment sauvegardé, est réinjecté dans la nouvelle version.
- ❑ Quand la nouvelle version du composant est complètement construite, le gestionnaire débloque les communications, précédemment bloquées.

8.2.3 Interactions entre composants et gestion des références

Un objet fonctionnel peut avoir besoin des fonctionnalités fournies par les autres composants. Il lui faut alors un moyen pour avoir les références des objets fonctionnels assurant ces fonctionnalités. Deux situations peuvent se présenter (Figure 29), selon le sens de la communication dans la hiérarchie des composants :

- ❑ *Références descendantes* : uniquement les références vers les sous-composants directs sont autorisées. Par exemple, un objet $O1$ peut demander une référence vers un objet $O2$, appartenant au sous-composant S (et accessible). Pour demander la référence, $O1$ appelle la méthode $CManager_s.bindToReachableObject(O2)$. Le gestionnaire du sous-composant S crée une enveloppe pour l'objet $O2$ et retourne sa référence à l'objet $O1$. L'enveloppe, appartenant à la partie permanente, joue le rôle de médiateur entre les deux objets.
- ❑ *Références montantes* : le composant parent est responsable de fournir les références nécessaires aux objets qu'il contient. Au moment de la création d'un composant C , son constructeur indique à son gestionnaire les besoins et les services fournis par le composant. Le gestionnaire prend la responsabilité de fournir les références nécessaires.

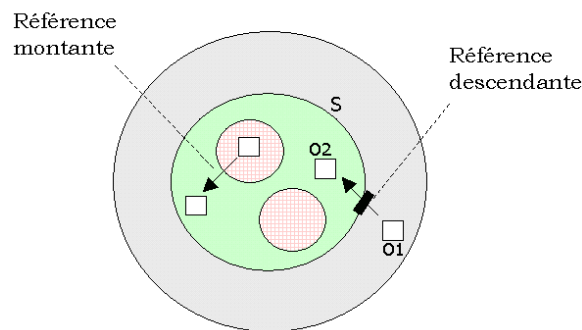


Figure 29. Références descendantes et montantes

8.2.4 Nommage des composants

Chaque composant de l'application est identifié par un nom unique. Ce nom permet de localiser le composant dans la hiérarchie de l'application, pour l'adapter par exemple. De la même manière, chaque objet atteignable, appartenant au composant, est aussi identifié par

un nom unique. Ce nom permet aux clients de demander la liaison vers l'objet en question. Deux espaces de nommage, par conséquent, sont supportés :

- ❑ *L'espace de nommage des composants* : il en existe un seul par application. Il répertorie les noms de tous les composants appartenant à l'application.
- ❑ *L'espace de nommage des objets* : il existe un espace par composant. Il répertorie les noms de tous les objets atteignables, appartenant au composant.

8.2.5 Les enveloppes et les objets accessibles

Comme expliqué précédemment, une enveloppe constitue le médiateur entre l'objet fonctionnel qu'elle représente, et les objets qui l'utilisent. En cas d'adaptation, elle est responsable de verrouiller l'accès à l'objet qu'elle enveloppe.

Si l'une des opérations sur l'objet doit retourner une référence d'un autre objet accessible, appartenant au même composant, l'enveloppe demande au gestionnaire du composant, de créer une enveloppe pour l'objet à retourner, et retourne, comme réponse, la référence de cette nouvelle enveloppe.

Les objets accessibles sont les seuls objets visibles de l'extérieur du composant. Un objet accessible doit explicitement implémenter l'interface *Reachable*.

```
public interface Reachable {  
    String getName();  
    void setName(String name);  
}
```

8.2.6 Modes d'adaptation

Les différentes versions de composants, sont maintenues par un *fournisseur de composants*. Deux modes d'adaptation sont supportés :

- ❑ Le mode *push* : l'initiative d'adaptation est faite par le fournisseur.
- ❑ Le mode *pull* : l'adaptation est décidée et déclenchée par le composant lui-même.

La coordination de l'adaptation est déléguée à un objet spécifique appelé *Updater*. A chaque composant, est associé un *Updater*, responsable de son adaptation. Que ce soit en mode *push* ou *pull*, l'adaptation est déclenchée en appelant la méthode *handleUpdateMessage()* de l'objet *Updater*. Ce dernier reçoit la nouvelle version du composant à partir du fournisseur, puis appelle la méthode *updateComponent()* sur le composant en question, pour démarrer effectivement l'adaptation.

8.2.7 Discussion

Nous avons présenté dans cette section DCUP, un modèle de composants hiérarchique. Un assemblage de composants est considéré comme un composant, ce qui permet de construire, plus facilement, des applications de grande complexité. L'adaptation dynamique, est l'une des propriétés mise en avant par DCUP. Plusieurs mécanismes ont été explicitement mis en œuvre, dans le modèle de composants, pour la supporter.

Les enveloppes sont utilisées pour éviter la liaison directe entre les objets qui communiquent. C'est un mécanisme qui permet de contrôler plus facilement le processus d'adaptation. Grâce à ce mécanisme, le système permet de garantir une certaine forme de cohérence. Ceci en bloquant les communications avant l'adaptation, et en les débloquent après. Cependant, un surcoût doit être payé à cause de l'indirection des appels, ce qui peut dégrader considérablement les performances des applications.

La séparation conceptuelle entre les interfaces fonctionnelles et les interfaces de contrôle, d'un côté, et entre la partie permanente et la partie remplaçable, d'un autre côté, est une idée très intéressante. Elle aide à comprendre l'architecture du système, et permet d'affecter des responsabilités spécifiques, à des éléments séparés. Cependant, ceci implique clairement, que seul le changement d'implémentation des composants est supporté. L'interface du composant, comprise dans la partie permanente, ne peut alors pas être changée.

Le transfert d'état, de l'ancienne version vers la nouvelle, est pris en compte dans le processus d'adaptation. D'après les concepteurs, l'état est déterminé par les objets qualifiés d'importants. Aucune explication n'a été donnée sur comment considérer qu'un objet est important, comment et qui doit spécifier cette information. [PBJ98] explique que l'état d'un composant est représenté par l'état des objets qu'il contient, et que l'état d'un objet est caractérisé par l'ensemble des valeurs des attributs qu'il contient. La responsabilité d'externalisation, et d'initialisation de l'état, est laissée à l'objet lui-même.

DCUP a l'avantage de définir une architecture complète, décrivant les applications en exécution, et les fournisseurs de ces applications. Ceci permet aux fournisseur d'assurer la maintenance des applications, en réalisant, dynamiquement, les adaptations nécessaires. Conceptuellement, les composants peuvent déclencher eux-même l'adaptation. Cependant, le système ne fournit aucun moyen explicite, permettant de définir la logique d'adaptation. Cette logique est nécessaire pour que les composants puisse raisonner, et prendre des décisions d'adaptation.

Les opérations d'adaptation sont réduites au remplacement de l'implémentation des composants. Si pour des raisons particulières, on souhaite ajouter un composant, ou changer une connexion à un certain niveau, le seul moyen possible est de redéployer tout le composant englobant le niveau en question. Ceci est très coûteux, et peut impliquer l'adaptation de toute l'application, si le changement doit intervenir au plus haut niveau de la hiérarchie des composants.

9 SOLUTIONS DE RECONFIGURATION ISSUES DU MONDE INDUSTRIEL

Dans les sections précédentes, nous nous sommes principalement intéressés aux approches académiques. Dans le monde industriel, plusieurs solutions ont été développées, pour supporter l'administration et la reconfiguration des applications et des équipements. Nous illustrons dans la suite, les fondements et les principes de base de ces solutions.

9.1 SNMP

SNMP (Simple Network Management Protocol) [CF+90] est un standard utilisé dans l'administration des équipements réseaux, comme les ponts, les concentrateurs et les routeurs. Sa première version a été développée en 1988. Elle a été suivie par deux autres versions, respectivement en 1993 et 1999, pour introduire des améliorations et corriger des problèmes de sécurité. SNMP est basé sur le modèle Manager/Agent. Ce modèle regroupe plusieurs éléments : le manager (station d'administration), les agents, les bases de données de gestion (MIB), les objets administrés et le protocole réseau qui assure la communication entre les différents éléments. Chaque équipement nécessite d'être modélisé sous la forme d'un objet administrable. Cet objet contient un ensemble d'attributs, caractérisant l'équipement, et pouvant être consultés et modifiés.

9.1.1 Architecture du système

SNMP est un protocole réseau qui permet aux équipements matériels, d'être administrés à distance, à partir d'une station d'administration, souvent appelée *le manager*. La Figure 30 montre l'architecture d'administration SNMP. Trois éléments principaux peuvent être distingués dans cette architecture : le manager, les agents et la MIB.

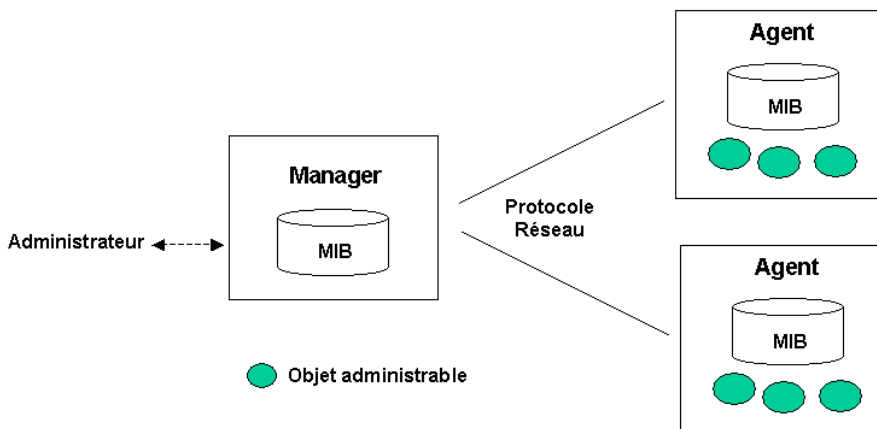


Figure 30. Architecture d'administration SNMP

9.1.1.1 Le manager

Le manager joue le rôle d'interface entre les applications d'administration, et les équipements réseau à administrer. Il permet ainsi, comme le montre la Figure 31, de recevoir les requêtes des applications d'administration, et de les envoyer sur le réseau. Le manager maintient une base de données de gestion. Cette base sera décrite par la suite.

9.1.1.2 Les agents

Un agent joue le rôle d'interface entre le manager, et un ou plusieurs objets administrables. Il est responsable de répondre aux requêtes envoyées par les applications d'administration, et de gérer les événements éventuels qui peuvent se produire sur les équipements réseau contrôlés.

9.1.1.3 La MIB

Pour administrer un équipement, ses caractéristiques doivent être représentées et stockées, sous un format compréhensible, à la fois par le manager et par les agents. Ces caractéristiques peuvent concerner des aspects matériels, comme la vitesse d'un ventilateur, par exemple. Elles peuvent aussi concerner des aspects applicatifs, comme les tables de routage. Les données, modélisant les caractéristiques des équipements, sont stockées dans une structure appelée la MIB.

Comme nous avons expliqué, les équipements à administrer, sont modélisés sous forme d'objets administrables. C'est justement la MIB qui organise et qui stocke ces objets. La MIB est organisée hiérarchiquement, de la même façon que l'arborescence des domaines Internet. Elle contient une partie commune à tous les agents SNMP, une partie commune à tous les agents SNMP d'un même type de matériel et une partie spécifique à chaque constructeur. Elle peut contenir des valeurs simples ou des tableaux de valeurs.

9.1.2 Le protocole de communication

Il définit l'ensemble des commandes qui permettent aux applications d'administration, de communiquer avec les agents. La Figure 31 montre le lien entre une application d'administration, et un agent SNMP. Elle illustre les différentes couches de communication, sur lesquelles SNMP est basé.

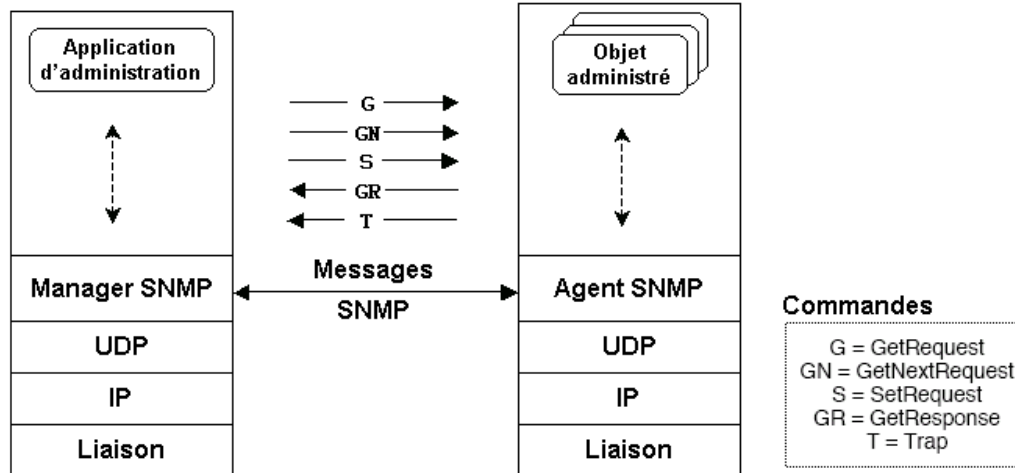


Figure 31. Protocole de communication

Le protocole de communication, entre le manager et les agents, se résume à cinq commandes. Ces commandes sont expliquées dans le Tableau 2.

Commande	Action
get-request	Le Manager SNMP demande une information à un agent, l'agent répond avec une requête de type <i>get-reponse</i> .
get-next-request	Le Manager SNMP demande l'information suivante à l'agent.
set-request	Le Manager SNMP demande à un agent de mettre à jour une information dans la MIB. L'équipement associé doit réagir en fonction de la modification.
get-reponse	L'agent SNMP répond à une requête de type <i>get-request</i> ou <i>set-request</i> .
trap	L'agent SNMP envoie une alerte au manager, pour l'informer d'une situation particulière concernant l'appareil.

Tableau 2. Commandes du protocole SNMP

En résumé, les commandes *get-request*, *get-next-request* et *set-request* sont toutes émises par le manager, à destination d'un agent, et attendent toutes une réponse de type *get-response*, de la part de l'agent. La commande *trap* représente une alerte. Elle est toujours émise par l'agent à destination du manager, et n'attend pas de réponse.

9.1.3 Discussion

SNMP est l'un des protocoles, les plus utilisés pour d'administration des équipements réseau. Il n'est pas figé pour un nombre particulier d'équipements. Il peut être étendu pour administrer toutes sortes d'appareils. Son succès vient particulièrement de sa simplicité. Comme nous l'avons expliqué, son architecture est basée sur peu d'éléments, et le nombre de commandes qu'il supporte est très réduit, ce qui simplifie son utilisation.

L'idée fondamentale derrière SNMP, est de modéliser les équipements à administrer, sous forme d'objets administrables, pour pouvoir les interroger et les modifier. Toutes les modifications appliquées à ces objets, se traduisent par des actions sur les équipements modélisés.

SNMP en tant que tel, n'est pas approprié à l'administration des applications pour différentes raisons. D'un côté, le nombre de commandes qui peuvent être échangées entre une application d'administration et une application administrée, est réduit et figé. D'un autre côté, la sémantique même des commandes ne suffit pas à administrer des applications complexes. Les commandes sont limitées uniquement à consulter et à modifier les valeurs des attributs.

Cependant, SNMP permet de confirmer plusieurs idées :

- ❑ Les entités à administrer ou à reconfigurer, doivent être modélisées dans un format bien spécifique. Ce format, compréhensible par le système de reconfiguration, permet à ce dernier d'interagir avec les entités modélisées.
- ❑ Lors de la modélisation, seules les propriétés qui peuvent être intéressantes pour la reconfiguration, doivent être prises en compte. Il est alors nécessaire d'abstraire les entités à administrer, et de décider quelle propriété doit être modélisée.
- ❑ Il faut séparer le système de reconfiguration des applications à reconfigurer. Ceci permet d'identifier clairement les responsabilités de chaque partie. Pour assurer le lien entre les différentes parties, un protocole d'interaction doit également être clairement exprimé.

9.2 JMX

JMX (Java Management Extensions) [JMX] est une extension du langage Java, développée par Sun Microsystems. Elle a été introduite pour supporter l'administration de différents types de ressources. Une ressource peut être une application, un objet, un service, un appareil, etc. La seule condition imposée est que la ressource soit modélisée ou instrumentée, sous forme d'un objet Java administrable. JMX définit une architecture d'administration, une API pour développer les applications, et un ensemble de services d'administration.

9.2.1 Architecture de JMX

L'architecture proposée par JMX est basée sur trois niveaux : le niveau *instrumentation*, le niveau *agent*, et le niveau *adaptateur*. Le premier niveau définit comment modéliser ou instrumenter les ressources, pour que les applications d'administration, basées sur JMX, puissent les contrôler. L'instrumentation est basée sur des objets particuliers appelés les *MBeans* (Manageable Beans). Le niveau agent, quant-à-lui, spécifie comment implémenter les agents, qui contrôlent les MBeans, et les rend accessibles aux applications d'administration. Le niveau adaptateur fournit les mécanismes nécessaires, pour que les applications d'administration distribuées, puissent communiquer avec les agents. La Figure 32 montre le lien entre les trois niveaux.

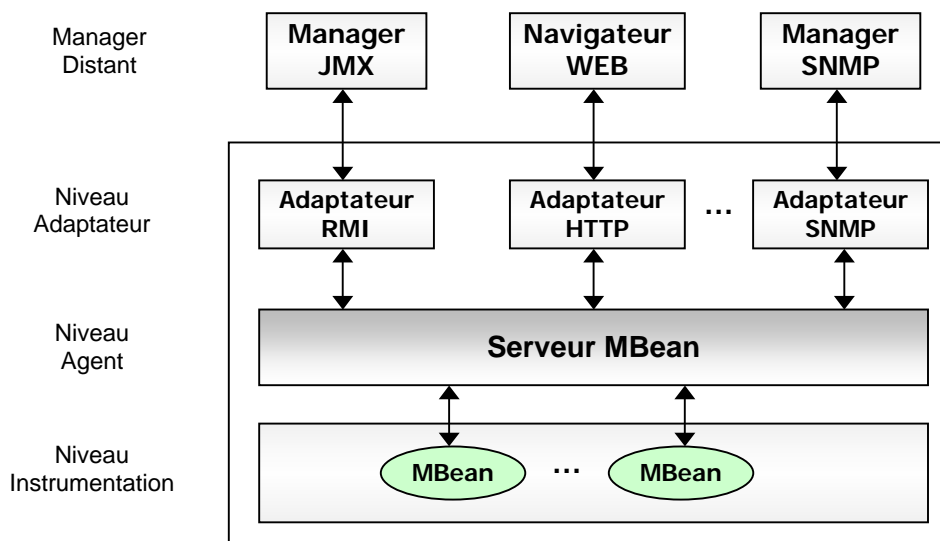


Figure 32. Niveaux de l'architecture JMX

9.2.2 Instrumentation des ressources

Une ressource administrable (classe, application patrimoniale, méthode, etc.) doit être développée en Java, ou, doit offrir une enveloppe Java. Elle doit être instrumentée sous forme d'un ou de plusieurs MBeans. Concrètement, un MBean est un objet Java qui suit une certaine sémantique. Il expose les informations de gestion sous forme d'attributs et d'opérations pour les manipuler. Il existe quatre types de MBean : standard, dynamique, modèle et ouvert.

9.2.2.1 MBean standard

Les MBeans standards constituent la plus simple forme d'instrumentation. Ils sont les plus appropriés à la gestion des informations statiques. Un ensemble de conventions lexicales doit être respecté lors de la mise en œuvre des MBeans. Le code suivant montre un exemple d'interface d'administration d'un MBean standard.

```
public abstract interface MyServerMBean
{
    public boolean isServerStarted( );
    public int getPort( );
    public void setPort(int port);
    public void startServer( );
    public void stopServer( );
}
```

L'interface précédente doit être implémentée pour créer un MBean standard.

```
public class ServerInfo implements MyServerMBean {
    ...
}
```

L'interface du MBean doit avoir un nom suffixé par le mot "*MBean*". Elle doit définir les opérations de consultation et/ou de modification des attributs du MBean. Elle doit également définir les opérations nécessaires pour administrer les ressources.

9.2.2.2 MBean dynamique

Dans les MBeans dynamiques, les attributs et les opérations de manipulation des ressources instrumentées, ne sont exposés qu'en exécution. Ce type de MBean est alors plus approprié à l'instrumentation des ressources dont l'ensemble de propriétés n'est pas statique. La classe d'un MBean dynamique doit implémenter l'interface prédéfinie "*javax.management.DynamicMBean*". En utilisant les MBeans dynamiques, les attributs et les opérations de manipulation des ressources, peuvent évoluer dans le temps en fonction des besoins. Les méthodes de l'interface prédéfinie permettent de les découvrir dynamiquement. Le Tableau 3 donne un aperçu des différentes méthodes de l'interface à implémenter.

Méthode	Description
<code>getMBeanInfo</code>	Retourne un objet de type <code>MBeanInfo</code> contenant la définition des attributs et des méthodes d'administration du MBean.
<code>getAttribute</code>	Permet de lire la valeur d'un attribut dont le nom est passé en argument.
<code>getAttributes</code>	Identique à <code>getAttribute</code> mais elle concerne un ensemble d'attributs
<code>setAttribute</code>	Permet d'écrire la valeur d'un attribut dont le nom et la valeur sont passés en argument.
<code>setAttributes</code>	Identique à <code>setAttribute</code> mais elle concerne un ensemble d'attributs
<code>invoke</code>	Permet d'exécuter la méthode dont le nom est passé en paramètre avec les paramètres correspondants.

Tableau 3. Interface des MBeans dynamiques

9.2.2.3 MBean modèle

Un MBean modèle est un MBean dynamique qui offre plus de flexibilité de développement. La différence principale est que le développement de la classe d'implémentation du MBean modèle est plus simple. Une classe nommée *RequiredModelMBean*, implémentant l'interface *ModelMBean*, qui caractérise un MBean modèle, est fournie par défaut. Cette classe peut être étendue, et ses méthodes peuvent être surchargées, en cas de besoin, pour obtenir la classe d'implémentation du MBean modèle. Il est également possible de créer un objet de type *ModelMBeanInfo* et de l'affecter à une instance de la classe par défaut. Une alternative à la création de cet objet serait de fournir un fichier XML [XML00] qui décrit les propriétés du MBean.

9.2.2.4 MBean ouvert

Un MBean ouvert est lui aussi un MBean dynamique. Il est par contre restreint d'accepter et de retourner un nombre limité de types de données. L'utilisation d'un nombre restreint de types, permet d'éliminer le besoin de chargement de classes. Les types autorisés, lors du développement d'un MBean ouvert, sont :

- ❑ les types primitifs (*int*, *boolean*, *float*, etc.),
- ❑ les classes qui enveloppent les types primitifs : (*Integer*, *Boolean*, *Float*, etc.)
- ❑ les tableaux des types précédents.

Un MBean ouvert doit implémenter la méthode *getMBeanInfo()*, qui retourne un objet de type *OpenMBeanInfo* (sous-type de *MbeanInfo*). Cet objet fournit des méta-données supplémentaires, décrivant le MBean ouvert.

9.2.2.5 Modèle de notification

Généralement, la communication dans un système d'administration basé sur JMX est initiée par l'application d'administration. L'agent, à la réception d'une requête, se charge de la transférer vers le MBean concerné. Il se charge, ensuite, de récupérer la réponse du MBean, et de la transférer vers l'application d'administration. Certaines situations, par contre, peuvent arriver d'une manière aléatoire et complètement imprévisible. Le mécanisme de notification permet à l'application d'administration (et même à l'agent), d'être informée lorsque certains événements surviennent.

Le MBean qui est susceptible d'envoyer des événements, doit implémenter l'interface *NotificationBroadcaster*. Cette interface définit des méthodes pour ajouter et supprimer les auditeurs, intéressés par les événements envoyés. Les éléments intéressés par un type d'événement, doivent implémenter l'interface *NotificationListener*, et doivent souscrire auprès du MBean qui est susceptible d'envoyer l'événement. Les auditeurs implémentent la méthode *handleNotification*, qui est appelée par le MBean, au moment de la notification.

9.2.3 Le niveau agent

Un agent JMX est une entité d'administration qui s'exécute dans une machine virtuelle Java. Elle agit comme une liaison entre la station d'administration et les MBeans. Un agent est constitué d'un *serveur de MBeans*, gérant un ensemble de MBeans, et d'un ensemble de *services* de base implémentés sous forme de MBeans.

9.2.3.1 Serveur de MBeans

Un serveur de MBeans est une base qui contient et gère un ensemble de MBeans. C'est le cœur du niveau agent. Toutes les opérations de gestion exécutées sur les MBeans sont faites à travers le serveur, dans lequel ils sont enregistrés. Le serveur est aussi responsable, à la réception d'une requête de la station d'administration, de rechercher le MBean concerné, et de lui transférer la requête. La gestion au niveau du serveur de MBeans concerne, par exemple, l'instanciation des MBeans, l'enregistrement et le dés-enregistrement des instances créées. Un mécanisme de notification est supporté pour envoyer des notifications, après l'enregistrement et le dés-enregistrement des instances de MBeans.

9.2.3.2 Services de l'agent

Un serveur de MBeans fournit un ensemble de services, implémentés sous forme de MBeans. Parmi les services fournis, nous pouvons citer :

- ❑ Le service *Timer* : envoie des notifications à des moments spécifiques, d'une manière simple ou répétitive, à des intervalles réguliers.
- ❑ Le service *Monitor* : permet d'observer des attributs d'un type Java, et d'envoyer des événements lorsque des conditions spécifiques sont vérifiées.

- ❑ Le service *M-Let* : permet de télécharger, d'instancier et d'enregistrer des MBeans distants, en utilisant l'URL où ils se trouvent.

9.2.4 Niveau adaptateur

C'est le troisième niveau de l'architecture JMX. Il définit au moins un adaptateur ou un connecteur. Il permet aux applications d'administration distribuées à travers le réseau, de se connecter et d'utiliser les agents JMX. Les applications d'administration peuvent, grâce à ce niveau :

- ❑ Consulter et modifier les attributs des MBeans enregistrés.
- ❑ Appeler les opérations d'administration des MBeans enregistrés.
- ❑ Instancier et enregistrer de nouveaux MBeans.
- ❑ S'enregistrer et recevoir des notifications envoyées par les MBeans.

Les connecteurs sont utilisés pour connecter les applications d'administration compatibles avec la spécifications JMX. Tandis que les adaptateurs sont utilisés pour permettre la communication avec d'autres types d'applications. Leur rôle est de transformer les opérations des MBeans et de leur serveur, dans une représentation compatible avec les applications d'administration concernées. Par exemple, des adaptateurs sont nécessaires pour permettre à des applications en HTML ou en SNMP, de communiquer avec des agents JMX.

9.2.5 Discussion

Cette section a été dédiée à présenter JMX, l'une des solutions industrielles d'administration qui commence à être largement utilisée. Malgré que JMX ne soit pas limité à l'administration des équipements, ses principes fondateurs sont fortement inspirés de SNMP. Les idées d'instrumentation, de contrôle, d'interaction et de partage de responsabilités ont été reprises de SNMP et étendues.

JMX, comme son nom l'indique, est fortement lié au langage Java. Les conventions lexicales, et notamment les suffixes imposés aux interfaces, et les préfixes imposés aux méthodes de consultation et de modification des attributs, sont très similaires à ceux que nous avons vues dans le modèle JavaBeans. Les applications développées en Java peuvent être facilement instrumentées pour répondre à la spécification JMX, et être par conséquent administrables. L'administration n'est pas limitée aux applications Java. Tout autre type d'application, et même les équipements matériels, peuvent être encapsulés ou associés à des objets Java, pour bénéficier de l'administration.

Contrairement à SNMP, l'ensemble des commandes, qui peuvent être échangées entre une station d'administration et des agents représentant les entités à administrer, n'est pas figé. L'utilisateur peut définir librement l'interface d'administration appropriée à son

application. L'utilisation des MBeans dynamiques, permet d'administrer des applications, dont l'ensemble des propriétés concernées par l'administration varie dans le temps.

Comme dans SNMP, l'administration dans JMX consiste à consulter et à modifier des attributs, et à appeler des opérations, définies par l'utilisateur sur les objets administrables. Les solutions d'administration que nous avons vues auparavant, vont jusqu'à la définition d'algorithmes complexes pour le remplacement des entités à l'exécution, en prenant en charge l'intégrité des applications. Comparé à ces solutions, JMX reste une solution d'administration de haut niveau, qui décrit comment instrumenter les applications, pour qu'elles puissent être contrôlées par des stations d'administration.

10 SOLUTIONS DE RECONFIGURATION MATERIELLES

Les systèmes de reconfiguration dynamique que nous avons présentés dans ce chapitre, sont tous basés sur des solutions logicielles. D'autres systèmes, en contre-partie, s'attaquent au problème de reconfiguration dynamique en fournissant une unité de calcul (CPU) et des périphériques redondants [Rey83]. Ces éléments redondants sont destinés à être chargés avec la nouvelle version de l'application à reconfigurer. Lorsque la reconfiguration devient nécessaire, le nouveau matériel est activé pour prendre le rôle de l'ancien, qui, à son tour, est désactivé. Dans la suite, nous présentons brièvement deux systèmes commerciaux, qui utilisent une solution matérielle pour supporter la reconfiguration dynamique.

10.1 ACARS

L'approche matérielle, est employée dans ACARS [ACARS]. Un système de messagerie aérienne développé et maintenu par la société ARINC, Inc. [ARINC]. ACARS utilise un routeur pour établir la communication entre les avions en vol et un aéroport. Le routeur communique son état, périodiquement, ou à la demande, à un autre routeur, considéré comme secondaire. Pour préparer la reconfiguration, la nouvelle version de l'application, s'exécutant sur le routeur principal, est chargée dans le routeur secondaire. Au moment de la reconfiguration, le routeur secondaire demande au routeur principal de lui transmettre son état, avec lequel il peut s'initialiser. Le déclenchement effectif de la reconfiguration se traduit par le changement des rôles des deux routeurs. Le routeur secondaire prend le relais et devient principal. Le routeur principal se désactive, devient secondaire, et attend une nouvelle reconfiguration pour reprendre le relais.

10.2 SCP

SCP (Service Control Point) [Smi95, NS98] est un système commercial, développé par le laboratoire Bellcore (BELL COmmunications Research). SCP fournit des services pour l'utilisation d'environ un millier de cartes à haute vitesse, de recherche et d'appel pour des

opérateurs téléphoniques. Le domaine d'utilisation du système SCP, le contraint à être disponible à tout moment, pour répondre aux requêtes des utilisateurs.

Pour atteindre les performances et le degré de disponibilité requis, SCP est conçu avec des éléments matériels et des moyens de communications redondants. Deux copies de l'application s'exécutent simultanément sur les deux configurations matérielles. Des routeurs reçoivent les requêtes, et les transmettent vers la configuration matérielle active. Comme dans ACARS, avant la reconfiguration, la nouvelle version de l'application est chargée dans la configuration matérielle inactive. La reconfiguration consiste à demander aux routeurs, au moment propice, de rediriger les requêtes qu'ils reçoivent vers la nouvelle configuration matérielle, afin qu'elles soient traitées par la nouvelle version de l'application.

La technique de reconfiguration utilisée dans SCP n'est pas complètement sûre. Une faible perte, concernant les requêtes reçues, peut être observée pendant la reconfiguration. Cependant, ceci est négligeable comparé au gain de temps par rapport à l'option d'arrêter et de redémarrer le système. Cette dernière option nécessite, d'après les concepteurs de SCP, deux semaines de planification, et huit heures d'indisponibilité pour réaliser la reconfiguration.

10.3 Discussion

Le grand obstacle devant l'utilisation d'une approche de reconfiguration matérielle, est clairement le coût élevé, nécessaire pour la mise en œuvre des solutions. L'approche est typiquement utilisée dans les systèmes qui nécessitent du matériel redondant pour répondre à des problèmes classiques, comme la tolérance aux pannes, par exemple. Ceci est typiquement le cas des systèmes de télécommunication. Nous pensons qu'il n'est pas pratique de doubler les ressources matérielles, simplement pour assurer que la reconfiguration des applications, soit faite d'une manière dynamique.

11 SYNTHÈSE ET CONCLUSION

11.1 Synthèse des approches étudiées

Ce chapitre a été dédié à l'exploration de quelques approches de reconfiguration dynamique. Nous avons d'abord expliqué le trait marquant de chaque approche, avant de l'illustrer avec un exemple de système concret. Selon le support de reconfiguration utilisé, il est possible de classer les solutions que nous avons survolées en deux catégories : les solutions *matérielles* et les solutions *logicielles*. La première catégorie est en général basée sur l'utilisation de matériel redondant pour supporter la reconfiguration dynamique. Elle est appliquée dans des domaines critiques comme les télécommunications par exemple. C'est une solution très coûteuse, qui ne peut être envisagée que dans le cas où le contexte d'application utilise déjà du matériel redondant pour d'autres raisons, comme la reprise sur panne par

exemple. Les solutions logicielles, d'un autre coté, sont celles que nous avons le plus explorées, car elles sont les plus utilisées en général. Ces solutions sont issues soit du monde industriel, soit du monde académique.

Les travaux que nous avons étudiés, se sont intéressés à la reconfiguration sous différents angles. L'objectif, en général ciblé, est de trouver des solutions qui permettent de modifier les constituants d'une application :

- sans l'arrêter,
- en la perturbant le moins possible,
- en assurant un niveau acceptable de qualité de service,
- en préservant l'intégrité de l'application,
- en impliquant les acteurs humains le moins possible,
- et en évitant la dégradation des performances.

La résolution d'une équation, comportant tous ces paramètres, n'est pas une tâche triviale. Souvent, quelques éléments de l'équation, sont négligés au détriment des autres.

Le Tableau 4 résume les systèmes que nous avons présentés dans ce chapitre.

Propriété Système	Développeur	Granularité de modification	Statut	Modifications supportées à l'exécution	Remarques
OpenVL	Université de Stony Brook	Plugin	Projet de recherche	Remplacement/ajout de plugins	Basé sur C++
Iguana/J	Trinity College de Dublin	Divers	Projet de recherche	Modification du comportement des applications Java	Basé sur Java - Modification de la JVM - Approche réflexive
DYMOS	Insup Lee	Procédure/ Module	Projet de recherche	Remplacement de procédures (actives ou non) ou de modules	Basé sur StarMod
PODUS	Mark Segal et Ophir Frieder	Procédure	Projet de recherche	Remplacement de procédures non actives	Basé sur C - Modification de la signature d'une procédure
Polyolith	Université de Maryland	Module	Projet de recherche	Remplacement/ajout/ Suppression/migration de modules, modification des interconnexions	Basé sur C - Supporte les points de reconfiguration
Classes dynamiques de Java	Université de Californie	Classe/Objet	Projet de recherche	Remplacement des classes/objets	Basé sur Java - Modification de la JVM

Propriété Système	Développeur	Granularité de modification	Statut	Modifications supportées à l'exécution	Remarques
Classes dynamiques de C++	Laboratoire AT&T et Dartmouth College	Classe	Projet de recherche	Remplacement des classes	Basé sur C++
K-Components	Trinity College de Dublin	Composant	Projet de recherche	Remplacement de l'implémentation des composants, changer le type de communication	Basé sur C++ et IDL-3 - Approche réflexive - supporte l'auto-reconfiguration
DCUP	Université Charles	Composant	Projet de recherche	Remplacement de l'implémentation des composants	Basé sur C++ - Modèle hiérarchique
SNMP	SNMP Research	-	Projet industriel	Administration des équipements réseau	Propose une architecture d'administration
JMX	Sun Microsystems	-	Projet industriel	Administration des équipements et des applications - Support d'instrumentation	Basé sur Java - Etend les idées de SNMP
ACARS	ARINC, Inc.	Routeur	Projet industriel	Remplacement d'une application entière	Système de messagerie aérienne
SCP	Bellcore	Machine	Projet industriel	Remplacement d'une application entière	Système de télécommunication

Tableau 4. Résumé des systèmes présentés

11.2 Conclusion sur les approches étudiées

Les approches que nous avons présentées, montrent la complexité à laquelle il faut faire face, pour mettre en œuvre un système de reconfiguration dynamique. La complexité est plus importante si les éléments constituant les applications sont de faible granularité, ou ne sont pas suffisamment isolés. Dans les systèmes qui ont adressé le remplacement des modules, par exemple, des détails de bas niveau, comme l'assignation des adresses par le compilateur, doivent être pris en compte. Dans les systèmes orientés composant, les liens explicites entre les composants, facilitent le raisonnement sur l'architecture, et simplifient la mise en œuvre des solutions de reconfiguration. Ceci explique en partie notre choix d'explorer la reconfiguration dynamique des applications orientées composant.

Nous pouvons conclure d'après l'étude de certains systèmes que :

- Les applications à reconfigurer et les systèmes responsables de leur reconfiguration, doivent être clairement séparés. Ceci permet de raisonner sur chaque niveau indépendamment de l'autre. Le développeur des applications, doit se préoccuper uniquement de la logique fonctionnelle des applications qu'il

développe. La responsabilité de reconfiguration doit être déléguée au système de reconfiguration, qui ne doit s'occuper que de cette tâche.

- Pour développer des solutions de reconfiguration générales, et de haut niveau, sans rentrer dans des détails de bas niveau, qui risquent d'être très complexes, les applications doivent être construites dans une optique de reconfiguration. Elles doivent, en particulier, répondre aux exigences du système de reconfiguration afin qu'elles bénéficient des fonctions de reconfiguration.

11.3 Aspects liés aux systèmes de reconfiguration

Parmi les aspects les plus importants, abordés en général par les différents systèmes de reconfiguration que nous avons étudiés, nous pouvons citer :

- **Transfert d'état** : il est considéré comme une condition fondamentale, pour assurer, en partie, la cohérence des applications reconfigurées. Les entités en exécution (objets, instances de composants, etc.) ont un état qui évolue depuis la mise en service de l'application. Cet état correspond à une certaine sémantique englobée par les entités de l'application (solde d'un compte, historique, valeur d'un paramètre, etc.). Si en remplaçant une entité, son état n'est pas transféré vers l'entité remplaçante, la sémantique représentée par l'application peut être perdue. Plusieurs questions sont liées au problème de transfert d'état :
 - **Responsabilité du transfert** : pour assurer la fonction de transfert d'état dans un système de reconfiguration, il faut d'abord définir le responsable de cette fonction. La responsabilité concerne, d'un côté, la spécification de ce qu'est exactement l'état. Et d'un autre côté, elle concerne les mécanismes nécessaires à mettre en œuvre pour réaliser effectivement le transfert.
 - **Transfert à la charge du développeur** : le développeur doit fournir par exemple des méthodes pour encoder et décoder l'état.
 - **Transfert à la charge du système** : le système doit déterminer, encoder et décoder l'état. Même si l'encodage et le décodage peuvent être automatisés avec des mécanismes de bas niveau, comme dans le cas des classes dynamiques de Java, la détermination des propriétés constituant l'état doit être guidée par le développeur. Ceci vient du fait, que seul le développeur d'une application, est capable de désigner les propriétés importantes à prendre en considération.
 - **Etendue du transfert** : deux types de transfert d'état peuvent être considérés :

- **Transfert faible** : il concerne les valeurs des structures de données appartenant aux entités remplacées. Par exemple, les variables, les tableaux, etc.
 - **Transfert fort** : il concerne le flot d'exécution. Lorsque le remplacement implique du code, les processus en exécution peuvent être affectés. Le transfert fort doit garantir que les processus affectés par le changement, reprennent leur exécution, dans le nouveau code, à partir d'un point cohérent.
- **Préservation des communications** : au moment de la reconfiguration, des communications peuvent être en cours, au départ ou à destination des entités à remplacer. Le système de reconfiguration doit garantir que ces communications ne soient pas perdues. Les solutions qu'il faut mettre en œuvre dépendent fortement du modèle de communication. Par exemple, une communication synchrone ne peut pas être traitée comme une communication asynchrone, et vice-versa.
- **Facilité d'utilisation** : nous pensons que la réussite d'une solution de reconfiguration dépend essentiellement de sa facilité d'utilisation. Un bon système de reconfiguration doit garantir une interface d'utilisation simple et intuitive. Il doit également, dans la mesure du possible, automatiser les vérifications nécessaires avant de procéder à la reconfiguration. Par exemple, pour remplacer une entité par une autre, il faut disposer des moyens et des informations nécessaires, qui permettent de vérifier que la nouvelle entité est compatible avec celle qui doit être remplacée.
- **Automatisation** : le système doit être suffisamment intelligent pour raisonner sur certaines situations, décider et déclencher les reconfigurations appropriées. Une telle capacité nécessite, d'un côté, l'utilisation d'un ensemble de mécanismes permettant de détecter les situations nécessitant une intervention, et d'un autre côté, la spécification de la logique qui permet au système de raisonner et d'agir.

Le chapitre suivant explique notre première contribution concrète, dans le cadre de cette thèse. Il permet d'illustrer la progression chronologique de notre travail, depuis l'étude de l'état de l'art, jusqu'à la proposition d'une approche générale de reconfiguration dynamique.

CHAPITRE 4

EXPERIMENTATIONS PRELIMINAIRES

PRÉAMBULE.

Nous avons mené, après l'étude de l'état de l'art, deux expérimentations préliminaires. L'objectif de la première était la mise en œuvre d'une solution de reconfiguration pour le modèle JavaBeans. La deuxième quant à elle adresse le modèle OSGi. Dans ce chapitre, nous décrivons ces deux expérimentations, et nous mettons en évidence les différences les plus significatives entre les deux solutions mises en œuvre. Nous terminons par une synthèse qui illustre, d'un côté, les limitations des deux solutions, et d'un autre côté, les besoins à satisfaire pour développer une solution plus générale.

1 INTRODUCTION

Ce chapitre présente les expérimentations que nous avons réalisées après l'étude de l'état de l'art. Ces expérimentations sont la pierre de base de DYVA, notre machine de reconfiguration dynamique. Elles nous ont aidé à consolider et à mettre en œuvre les différents concepts que nous avons présentés dans les deux chapitres précédents.

Au lieu de définir un nouveau modèle de composants, et de le doter de capacités d'adaptation, nous avons préféré baser notre travail sur des modèles existants. Ce choix est justifié par le fait que notre intérêt, dans cette thèse, est porté sur la problématique de reconfiguration dynamique, et pas sur les modèles de composants en tant que tels. D'un côté, le choix d'un modèle existant nous permet de nous attaquer directement au fond du problème auquel nous nous intéressons. D'un autre côté, il nous force à raisonner sur les limitations du modèle choisi en terme d'adaptabilité, et sur les mécanismes nécessaires pour combler les limitations potentielles.

Notre première contribution pratique dans cette thèse était l'intégration d'un support de reconfiguration dynamique au modèle JavaBeans. Ce modèle a été essentiellement choisi pour sa simplicité, et pour sa facilité d'utilisation. Nous avons ensuite réalisé un travail similaire pour le modèle OSGi. Cette deuxième expérimentation sur un autre modèle avait pour objectif, d'un côté, la validation des idées de la première expérimentation, et d'un autre

coté, l'exploration des différences entre les deux. Ce chapitre présente d'abord ces deux expérimentations. Puis il illustre la solution que nous avons mis en œuvre pour le problème de changement d'interfaces. Nous concluons ce chapitre par une synthèse des différents travaux que nous présentons.

2 RECONFIGURATION DYNAMIQUE DANS LE MODELE JAVA BEANS

Le modèle JavaBeans ne propose pas un support explicite pour la reconfiguration dynamique. Pour pouvoir reconfigurer dynamiquement une application développée en JavaBeans, le développeur doit prévoir l'infrastructure nécessaire pour assurer cette reconfiguration. Le travail que nous avons réalisé dans le contexte du modèle JavaBeans avait pour objectif de décharger le développeur de la responsabilité de reconfiguration dynamique.

2.1 Support de développement : la BeanBox

Les composants JavaBeans, après leur construction, peuvent être assemblés soit par programmation, soit en utilisant des outils d'assemblage graphique. La BeanBox est l'un des outils développés par Sun Microsystems pour cet effet. Outre le fait que c'est un logiciel libre, son architecture est très simple et facilement extensible. Nous nous sommes donc appuyés sur cet environnement pour faciliter la mise en œuvre de notre support de reconfiguration dynamique. La Figure 33 montre un aperçu de la BeanBox.

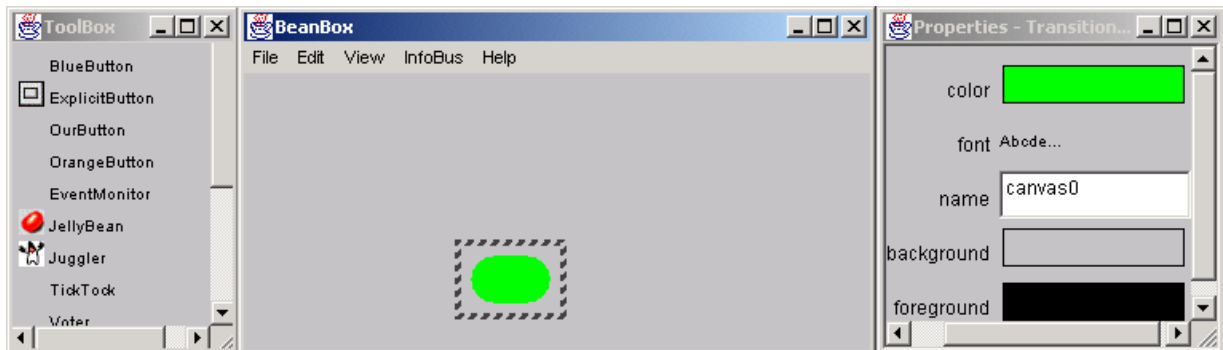


Figure 33. Aperçu de la BeanBox

2.1.1 Architecture de la BeanBox

Nous avons dans un premier temps étudié l'architecture de la BeanBox pour pouvoir l'étendre. Notre étude était principalement basée sur l'analyse du code source de la BeanBox, constitué de 42 classes Java. La Figure 34 montre l'architecture que nous avons extraite. Dans cette architecture, nous avons pris en compte uniquement les classes qui ont un lien direct avec l'extension que nous avons faite.

Lorsque la BeanBox est lancée, trois éléments sont automatiquement créés : la palette des composants (classe "ToolBox"), la fenêtre d'assemblage (classe "BeanBox") et la fiche de propriétés (classe "PropertySheet").

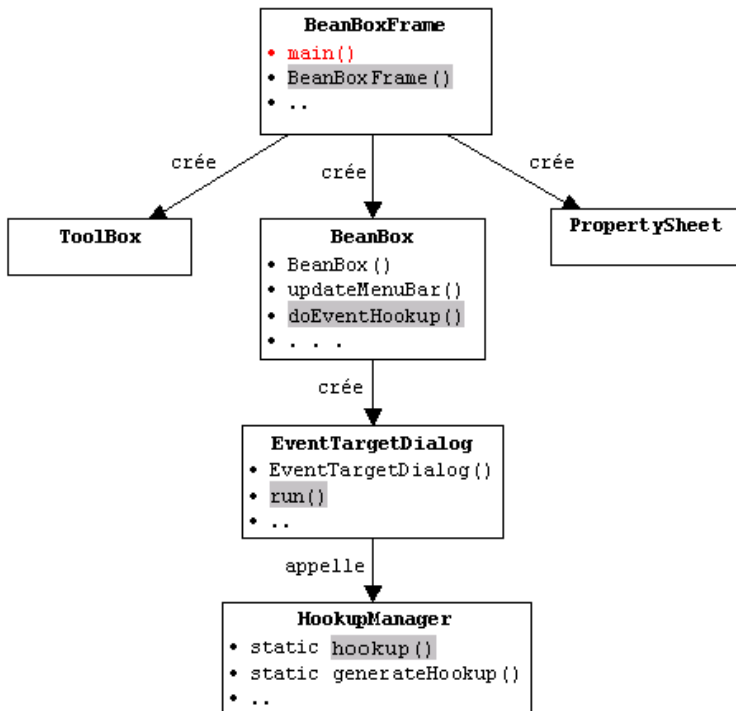


Figure 34. Architecture de la BeanBox

2.1.2 Assemblage des applications

Lors de l'assemblage graphique des applications, l'assembleur doit spécifier, pour connecter une instance de composant 'A' à une instance 'B', l'événement que 'A' doit envoyer, et la méthode de 'B' qui doit être appelée pour notifier l'événement à 'B'. Comme le montre la Figure 34, la méthode "doEventHookup()" de la classe "BeanBox" lance le processus d'assemblage, et crée une instance de la classe "EventTargetDialog". Cette instance représente l'interface qui permet de faire la sélection de la méthode à appeler pour notifier les événements. Après la validation de la sélection, la méthode statique "hookup()" de la classe "HookupManager" est invoquée pour effectuer la connexion entre 'A' et 'B'. La méthode "hookup()" crée un adaptateur qui permet de connecter les deux instances source et cible.

Pour illustrer le fonctionnement de la BeanBox, nous prenons un exemple d'application de télémétrie très simple. Notre application est composée de deux instances de Beans (*Probe* (capteur) et *Handler* (analyseur)), la première récolte les informations issues d'un appareil de

mesure. Elle envoie ensuite ces informations à la deuxième instance qui se charge de les analyser. Le code de l'adaptateur généré pour connecter *Probe* et *Handler* est le suivant :

```
public class Hookup implements telem.ProbeListener {
    public void setTarget(telem.Handler t) {
        target = t;
    }
    public void notify(telem.ProbeEvent arg0) {
        target.receive(arg0);
    }
    private telem.Handler target;
}
```

L'adaptateur généré automatiquement s'interpose entre les deux instances connectées. Il permet aussi d'adapter les noms des méthodes si la méthode appelée par *Probe* ne correspond pas à celle implémentée par *Handler*. Ceci est illustré par la figure suivante :

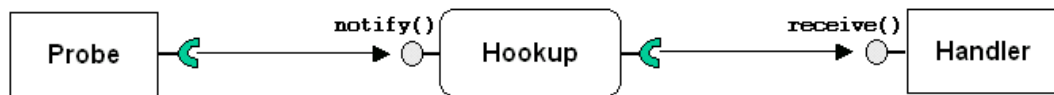


Figure 35. Architecture de l'application

2.2 Extension de la BeanBox : la DBeanBox

Pour supporter la reconfiguration dynamique, nous avons étendu la BeanBox [KBC02-a, KBC02-b]. L'objectif de l'extension était d'ajouter, d'un côté, les mécanismes nécessaires à la reconfiguration, et d'un autre côté, l'interface et les algorithmes qui se chargent de piloter cette reconfiguration.

2.2.1 Mécanismes de reconfiguration

L'adaptateur qui s'interpose entre deux instances de composants joue un rôle central dans la reconfiguration. Toutes les communications transitent par les adaptateurs. En augmentant leur capacité à gérer ces communications, il est possible d'augmenter la capacité de la BeanBox à supporter la reconfiguration dynamique. Dans Polyolith par exemple, nous avons expliqué que la reconfiguration dynamique est de la responsabilité d'un bus logiciel qui s'interpose en quelque sorte entre les modules. D'une manière identique, les adaptateurs doivent être capables de contrôler les communications, et de répondre aux requêtes de reconfiguration.

2.2.1.1 Adaptateur étendu

Les adaptateurs générés par la DBeanBox supportent la reconnexion dynamique. Ils permettent aussi de gérer les messages envoyés au moment de la reconnexion.

2.2.1.2 Gestion des reconnexions

L'adaptateur étendu implémente deux méthodes "`setTargetMethod()`" et "`setTargetObject()`". Ces deux méthodes permettent respectivement de modifier l'instance vers laquelle l'adaptateur est connecté, et aussi de modifier le nom de la méthode qui doit être appelée.

2.2.1.3 Gestion des messages

Avant de réaliser une opération de reconfiguration, il est nécessaire de verrouiller les canaux de communication impliqués dans l'opération. Ceci est nécessaire pour assurer des opérations de reconfiguration transparentes et éviter l'obligation de la participation des différentes instances dans la reconfiguration. Le verrouillage des canaux de communication permet d'éviter de perdre les messages envoyés pendant la reconfiguration.

Un adaptateur supporte deux états : *actif* et *passif*. Les deux méthodes "`activate()`" et "`passivate()`", implémentées par chaque adaptateur permettent de passer d'un état à l'autre. Les concepts liés à ces deux états sont expliqués dans les points suivants :

- ❑ Etat actif : à la réception d'un message, l'adaptateur l'envoie directement à sa destination.
- ❑ Etat passif : les messages reçus sont estampillés et stockés dans une file d'attente.
- ❑ Passage de l'état passif à l'état actif : avant de passer à l'état actif, l'adaptateur doit attendre que sa file d'attente soit vide. La gestion des files d'attente et le traitement des messages stockés est expliqué dans ce qui suit.
- ❑ Files d'attente des messages : il est important de noter que chaque canal de communication est matérialisé par un adaptateur. Ces adaptateurs sont séparés et peuvent changer d'état d'une manière indépendante. Nous avons alors mis en place une file d'attente par adaptateur. L'utilisation d'une file d'attente par adaptateur au lieu d'une file d'attente globale est un simple choix d'implémentation. Nous l'avons adopté pour déléguer à chaque adaptateur la responsabilité de traduire les requêtes, sauvegardées dans sa file d'attente, si l'instance cible est remplacée par une instance ayant une interface différente.

L'absence d'une file d'attente globale complique fortement la tâche de traitement des messages après l'activation des adaptateurs. Tout message reçu doit être estampillé avec sa date d'arrivée. Tout adaptateur doit

implémenter deux méthodes `getOldStamp()` et `sendOldEvent()`. Ces deux méthodes sont utilisées par le gestionnaire d'activation.

- Gestionnaire d'activation : c'est le responsable de l'activation des adaptateurs. Il demande à tous les adaptateurs qui doivent passer à l'état actif de lui retourner leur plus ancienne estampille. Il demande ensuite à l'adaptateur qui a l'estampille la plus ancienne d'envoyer le message qui correspond à cette estampille. Les adaptateurs sont activés lorsque tous les messages sont consommés.

2.2.2 Interface et algorithmes de reconfiguration

L'interface de reconfiguration permet à l'utilisateur d'intervenir sur une application en exécution, pour la reconfigurer. L'interface de reconfiguration de la DBeanBox offre plusieurs services, comme illustré par la Figure 36.

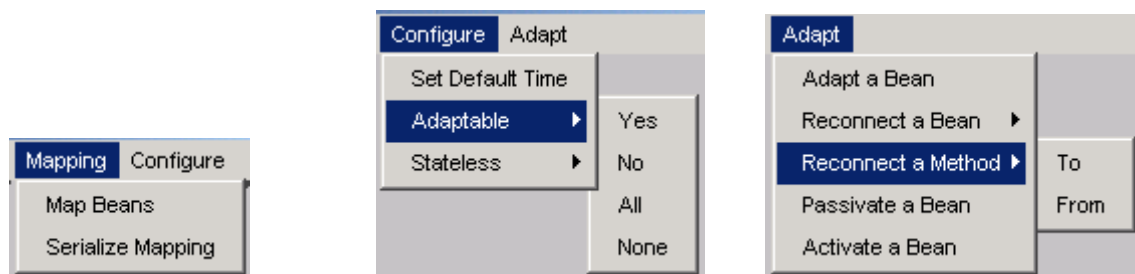


Figure 36. Interface de reconfiguration de la DBeanBox

2.2.2.1 Mapping

Une instance de Bean peut être remplacée par une autre instance d'un même Bean. Dans ce cas, les deux instances ont la même structure et possèdent les mêmes interfaces. Cependant, si la nouvelle instance est issue d'un Bean différent, elle peut avoir une structure et une interface différente. Dans ce dernier cas, il est nécessaire de définir les règles de correspondance entre l'ancien et le nouveau Bean.

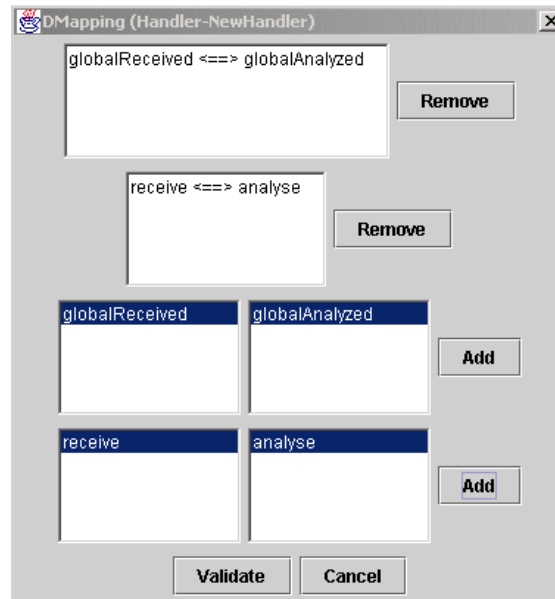


Figure 37. Mapping de l'état et des opérations

Par exemple, dans l'application de télémétrie, nous souhaitons remplacer l'instance *Handler* par une nouvelle instance *NewHandler* d'un autre Bean. Il est nécessaire de définir le mapping entre ces deux instances. Comme le montre la Figure 37, la propriété *globalReceived* de *Handler* a été mise en correspondance avec la propriété *globalAnalyzed* de *NewHandler*. Egalement, la méthode *receive()* de *Handler* a été mise en correspondance avec la méthode *analyse()* de *NewHandler*. La découverte des propriétés constituant l'état et les méthodes implémentées par chaque Bean se fait par la DBeanBox en utilisant l'introspection de Java.

Plusieurs stratégies peuvent être considérées par rapport à la portée du mapping :

- ❑ *Mapping par instance* : la correspondance ne s'applique qu'aux instances sélectionnées explicitement. Par exemple, le mapping défini entre l'instance '*BiIi*' (instance *i* du Bean *Bi*) et l'instance '*BjIj*' ne s'applique qu'à ces deux instances.
- ❑ *Mapping par composant* : la correspondance s'applique à toutes les instances. Par exemple, le mapping défini entre l'instance '*BiIi*' et l'instance '*BjIj*' est valable pour toutes les instances des mêmes composants. N'importe quelle instance du Bean '*Bi*' sera mise en correspondance, suivant la même définition, avec n'importe quelle instance du Bean '*Bj*'.

Le service de mapping que nous avons mis en œuvre pour la DBeanBox est très réduit. Il permet de faire correspondre deux propriétés, obligatoirement de même type, même si leurs noms sont différents. Il permet aussi de faire correspondre deux méthodes, obligatoirement de même signature, même si leurs noms sont différents. Par rapport à la portée du mapping, c'est la deuxième stratégie qui a été adoptée (mapping par composant).

❖ Anti-symétrie du mapping

Il est important de noter que dans le cas général, le mapping n'est pas symétrique. L'exemple présenté par la Figure 37 est très élémentaire. Dans des cas plus complexes, il est parfois nécessaire de définir des formules qui permettent de calculer la valeur d'une propriété à partir de celle qui lui correspond. Ces formules s'appliquent en général dans un sens mais pas forcément dans l'autre.

2.2.2.2 Configuration

Ce service permet de modifier des propriétés (annotations) attachées aux Beans et à leurs instances. Par exemple, une instance peut être déclarée comme adaptable ou non. Seules les instances déclarées adaptables peuvent être remplacées dynamiquement. Ceci permet de limiter la portée des opérations de reconfiguration. De la même façon, un Bean peut être déclaré à état ou non. Cette information est importante au moment du remplacement d'une instance d'un Bean. Elle permet de savoir s'il faut transférer l'état de l'ancienne instance vers la nouvelle ou non.

2.2.2.3 Reconfiguration dynamique

Cette partie de l'interface définit les différentes opérations de reconfiguration dynamique que la DBeanBox supporte.

□ Mettre une instance à l'état "passif"

Cette opération se traduit par le verrouillage de tous les canaux de communication à destination de l'instance en question. La méthode `passivate()` est appelée sur tous les adaptateurs sources. La Figure 38 montre la passivation de l'instance 'C' à travers la passivation des deux adaptateurs sources 'H1' et 'H2'.

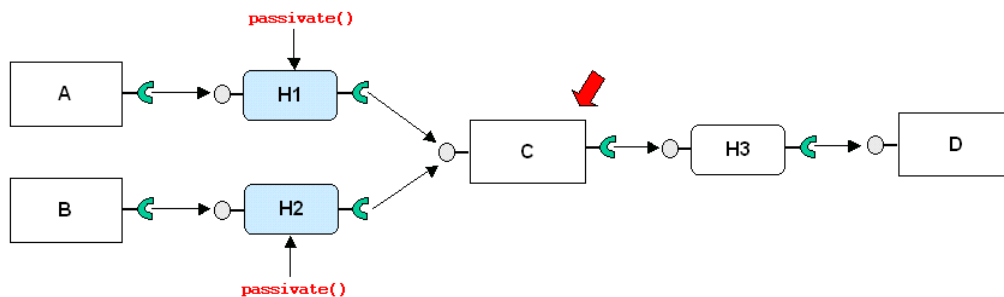


Figure 38. Passivation d'une instance

□ **Mettre une instance à l'état "actif"**

Cette opération se traduit par le déverrouillage des canaux de communication précédemment verrouillés. Le gestionnaire d'activation se charge de synchroniser cette opération.

□ **Reconnexion sélective**

Cette opération permet de reconnecter un canal de communication particulier entre deux instances. Les autres canaux ne sont pas affectés. Comme illustré par la Figure 39, la reconnexion peut affecter la source ou la destination du canal de communication.

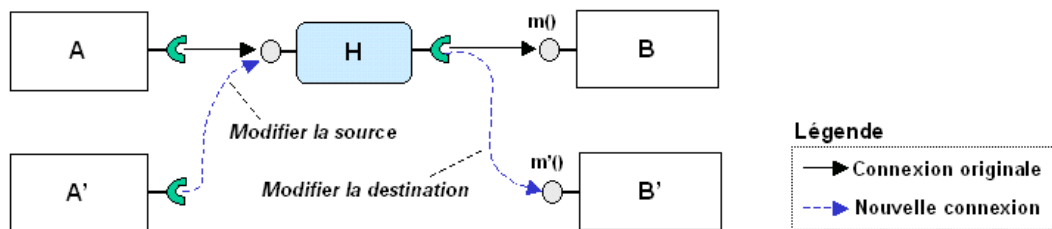


Figure 39. Reconnexion dynamique

L'algorithme de reconnexion dans le cas de changement de la source (cas1) ou de la destination (cas2) se présente comme suit :

Bloquer le canal de communication concerné par la reconnexion en appelant la méthode `passivate()` de l'adaptateur.

Etablir la reconnexion :

Cas 1 : remplacer la référence de l'instance et de la méthode cibles dans l'adaptateur (remplacer `B` par `B'` et `m` par `m'`).

Cas 2 : supprimer l'adaptateur de la liste des auditeurs de l'ancienne instance (`A`) et l'ajouter dans la liste des auditeurs de la nouvelle instance (`A'`).

Ouvrir le canal de communication et traiter les messages en attente.

Il est important de préciser que dans le deuxième cas, la nouvelle et l'ancienne instance (`A` et `A'`) doivent avoir la même interface d'écoute (l'interface d'écoute est celle qui doit être implémentée par les auditeurs des événements).

□ **Reconnexion totale**

Contrairement à la reconnexion sélective, la reconnexion totale permet de reconnecter tous les canaux de communication entre deux instances. D'une manière analogue au cas précédent, la reconnexion peut affecter la source ou la destination des canaux de communication. L'algorithme de reconnexion est basé sur la reconnexion sélective. Par

exemple, pour reconnecter une instance A déjà connectée à B , vers B' , il suffit de faire la reconnexion sélective de tous les canaux de communication, reliant A à B , vers B' .

□ Remplacement des instances

La DBeanBox permet de remplacer une instance de Bean par une autre instance. Si les deux instances sont de même Bean, il suffit que l'instance à remplacer soit déclarée adaptable. Dans le cas contraire, il est nécessaire qu'une correspondance entre les deux instances soit déjà définie pour que la requête de remplacement puisse être acceptée. L'algorithme de remplacement d'une instance A par une nouvelle instance B est le suivant :

1. Bloquer tous les canaux de communication à destination de A (mettre A à l'état passif).
2. Si les deux instances sont à état, transférer l'état de l'ancienne instance vers la nouvelle. Les informations de correspondance entre les propriétés sont exploitées à ce stade.
3. Reconnecter B à la place de A (toutes les connexions qui pointaient vers A doivent pointer vers B , toutes les connexions qui partaient de A , doivent partir de B). Les informations de correspondance entre les méthodes sont exploitées à ce stade.
4. Débloquer les canaux de communications auparavant bloqués (mettre B à l'état actif).

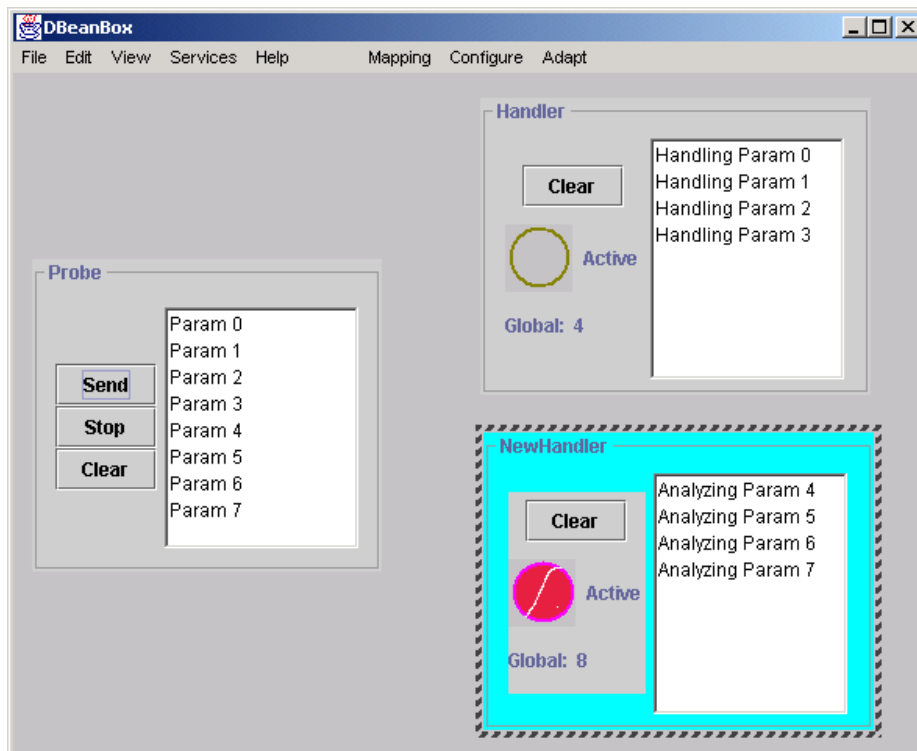


Figure 40. Remplacement dynamique des instances

La Figure 40 montre l'interface de la *DBeanBox* et le remplacement de l'instance *Handler* par l'instance *NewHandler*. Cette nouvelle instance implémente un algorithme différent pour analyser et réagir aux informations collectées par l'instance *Probe*.

2.2.3 Mise en œuvre de la *DBeanBox*

Pour mettre en œuvre la *DBeanBox*, nous avons étendu plusieurs classes de la *BeanBox*. Les classes qui ont été étendues ont une relation directe avec l'architecture des applications. La Figure 41 illustre l'architecture de la *DBeanBox*. Les classes les plus importantes sont :

- *DBeanBox* : étend la classe *BeanBox*. Elle définit essentiellement les trois méthodes suivantes :
 - *updateMenuBar()* : définit les nouveaux menus liés aux opérations de reconfiguration.
 - *doEventHookup()* : lance l'opération d'assemblage. Elle crée une instance de *DEventTargetDialog* au lieu de *EventTargetDialog*.
 - *handleNewMenus()* : traite les nouveaux menus de reconfiguration.
- *DEventTargetDialog* : étend la classe *EventTargetDialog*. Elle redéfinit la méthode suivante :
 - *run()* : lorsque la méthode cible (de l'instance cible) est sélectionnée, cette méthode se charge d'appeler la méthode statique *hookup()* définie dans la classe *DHookupManager*.
- *DHookupManager* : étend la classe *HookupManager*. Elle redéfinit les deux méthodes suivantes :
 - *hookup()* : appelle la méthode *generateHookup()* pour créer un adaptateur. Elle se charge ensuite de le compiler et de l'instancier.
 - *generateHookup()* : génère la classe Java qui implémente l'adaptateur utilisé dans l'assemblage.

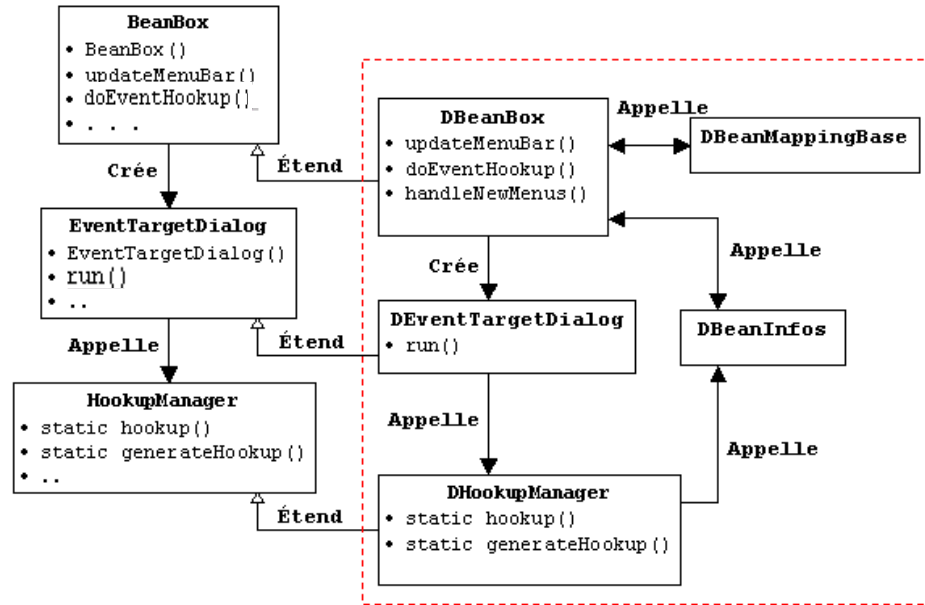


Figure 41. Architecture de la DBeanBox

Nous avons présenté dans cette section notre première contribution pratique dans le cadre de cette thèse. Nous discutons brièvement dans la section suivante un travail similaire que nous avons réalisé pour le modèle OSGi. Nous mettons surtout l'accent sur les différences les plus importantes entre les deux travaux que nous avons réalisés.

3 RECONFIGURATION DYNAMIQUE DANS LE MODELE OSGI

Dans cette section, nous présentons brièvement notre travail sur la reconfiguration dynamique pour le modèle OSGi [KC+02]. Comme nous l'avons expliqué précédemment, le framework OSGi permet de déployer dynamiquement des "bundles" (unités de déploiement). Il permet également de les activer, de les désactiver, de les mettre à jour et de les retirer de l'application. Les bundles sont utilisés pour encapsuler les composants. La grande capacité d'OSGi de supporter et de gérer le déploiement dynamique, permet de construire des applications évolutives et flexibles. Cependant, toute la puissance d'OSGi se concentre sur la gestion des unités de déploiement. La gestion des instances de composants est laissée à la charge du développeur. Par exemple, lorsqu'un bundle est déployé dynamiquement, l'application est responsable de récupérer les références des instances de composants encapsulées dans le bundle déployé. Lorsqu'un bundle est retiré dynamiquement, l'application doit réagir en libérant les instances encapsulées par le bundle retiré. En cas de besoin d'utilisation d'une instance au lieu d'une autre, il faut tout programmer explicitement.

Le travail que nous avons mis en œuvre donne la possibilité à l'utilisateur d'agir dynamiquement sur une application, selon les besoins, sans être obligé de tout prévoir et

programmer à l'avance. Les opérations et les algorithmes de reconfiguration que nous avons mis en œuvre sont identiques à ceux de la DBeanBox. Nous insistons donc dans cette section uniquement sur les différences les plus importantes entre les deux travaux. Notamment sur la forme de la solution mise en œuvre et sur la gestion des communications pendant la reconfiguration dynamique.

3.1 OSGiAdaptor : service OSGi dédié à la reconfiguration

Contrairement à la DBeanBox, notre solution de reconfiguration pour OSGi est indépendante de toute implémentation de la spécification OSGi. Cette solution se présente comme un service OSGi : *OSGiAdaptor*. Ce service peut être déployé et manipulé comme tout autre service. L'idée appliquée pour supporter la reconfiguration dynamique est très similaire à celle de la DBeanBox. Elle consiste à insérer des *proxys* dynamiques [DPC] entre les demandeurs et les fournisseurs de fonctionnalités.

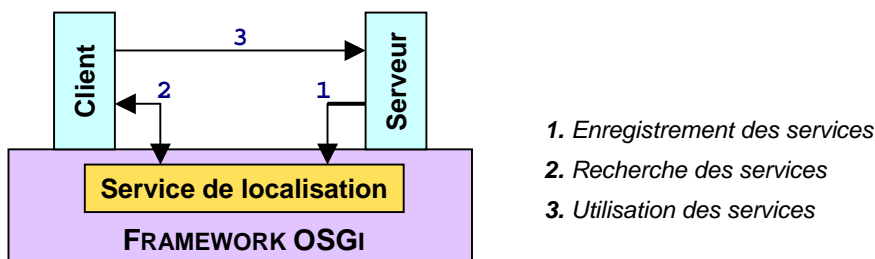


Figure 42. Enregistrement et recherche de services

OSGi est basé sur l'interaction par services. Ceci signifie que les fournisseurs et les utilisateurs de fonctionnalités ne se connaissent pas mutuellement. Un mécanisme de localisation constitue la partie centrale du système (Figure 42). Il joue le rôle d'intermédiaire entre les fournisseurs et les utilisateurs de fonctionnalités au sein d'une application. Les instances de composants l'utilisent pour enregistrer les services fournis d'un côté, et pour rechercher les services requis d'un autre côté.

OSGiAdaptor est basé sur un autre service, que nous avons développé, qui s'appelle OSGiServiceLookup (OSL). Ce service fournit un mécanisme de recherche plus élaboré au dessus du mécanisme de localisation original d'OSGi. Cette section est dédiée à OSGiAdaptor, OSL sera décrit dans la section suivante.

Comme le montre la Figure 43, avec OSGiAdaptor, l'enregistrement des services se fait d'une manière standard. Par contre les instances qui ont besoin de services, doivent s'adresser à OSL au lieu de les demander au service de localisation d'OSGi.

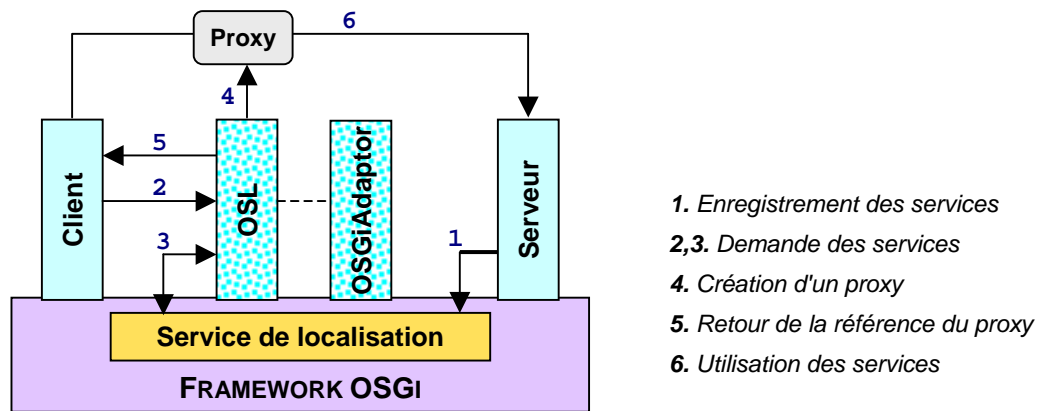


Figure 43. Utilisation du service OSL

A la réception d'une demande de service, OSL recherche le service demandé en utilisant le mécanisme de localisation d'OSGi. Il crée ensuite un proxy dynamique pour ce service et retourne la référence du proxy au demandeur. Il est possible que le client demande un service non disponible. Dans ce cas OSL tente de chercher des services compatibles. Ceci est expliqué plus en détail dans la section suivante. Nous expliquons également les mécanismes intégrés dans le proxy pour supporter la reconfiguration dynamique.

3.2 Gestion des communications

L'une des différences les plus significatives entre la DBeanBox et OSGiAdaptor réside dans la gestion des communications pendant la reconfiguration dynamique. Dans le premier cas, la communication est basée sur des événements. Des files d'attente estampillées nous ont suffi pour gérer les messages envoyés pendant la reconfiguration. Dans le cadre d'OSGi, la communication est basée sur des appels de méthodes. Sa nature synchrone était nettement plus compliquée à gérer.

Un proxy peut avoir l'un des deux états suivants :

- Etat actif : les appels sont directement redirigés vers leur destination.
- Etat passif : chaque appel intercepté est bloqué sur le *thread* [WO97] de l'appel précédent.

Pour passer de l'état passif à l'état actif, les appels précédemment bloqués sont libérés pour qu'ils puissent atteindre leur destination.

Le code suivant montre une vue simplifiée du thread qui permet de bloquer les appels lorsque le proxy est à l'état passif.

```

class WaitingThread extends Thread {
    ...
    public void waitUntilActivation() {
        try {
            synchronized(pred) {
                pred.wait(); // attendre le thread précédent
            }
            synchronized(this) {
                notify(); // libérer le thread en attente
            }
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

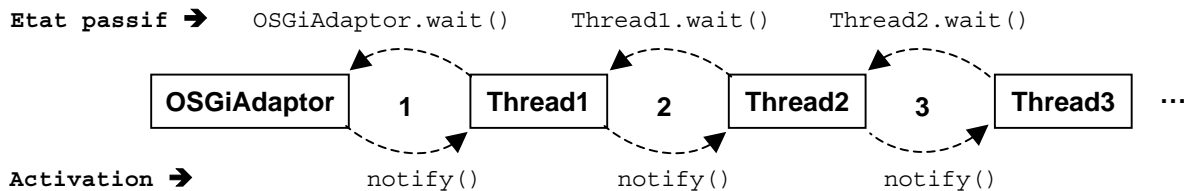


Figure 44. Blocage et libération des appels interceptés

Lorsqu'un appel est intercepté et que le proxy est à l'état passif, l'invocation de la méthode `waitUntilActivation()` permet de bloquer l'appel jusqu'à ce que l'appel précédent soit effectué. La Figure 44 illustre la chaîne de blocage et de libération des threads qui comportent les appels interceptés.

4 ADAPTABILITE DYNAMIQUE DES INTERFACES

Dans la section précédente, nous avons expliqué que le service OSGiAdaptor est fortement lié à un autre service appelé OSL (OSGiServiceLookup).

4.1 Problème à l'origine d'OSL

Lorsqu'une instance de composant, appartenant à une application OSGi, a besoin d'un service particulier, elle s'adresse au service de localisation d'OSGi pour rechercher le service requis. Si le service recherché est disponible, sa référence est retournée. Dans le cas contraire, l'instance qui a demandé le service risque de ne pas pouvoir continuer à s'exécuter normalement.

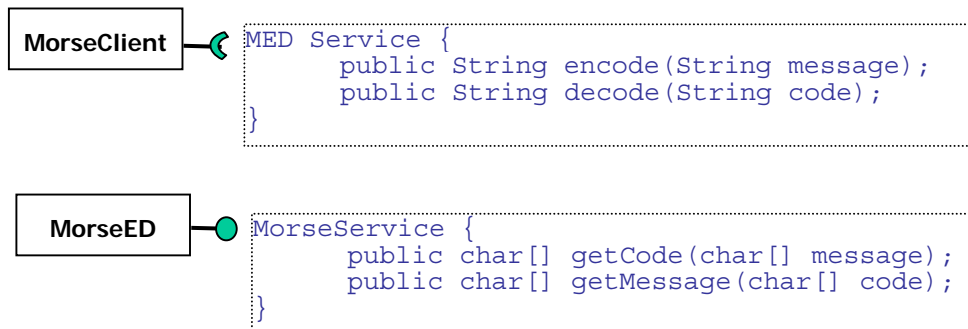
La plupart des systèmes orientés service [BC01-a] (tels que Jini [AO+99], Openwings [BC01-b],...) sont basés, en général, sur l'utilisation de services standardisés. La standardisation permet surtout l'interopérabilité entre des services mis en œuvre indépendamment, par différents développeurs. Malheureusement il est difficile d'imaginer la possibilité de standardiser tous les services qui peuvent être nécessaires à un moment donné. Parfois, des services équivalents à ceux recherchés peuvent être disponibles. Cependant, leur utilisation peut ne pas être possible s'ils ne répondent pas exactement à la spécification de l'interface du service requis.

C'est justement pour tenter de résoudre ce problème (dans le cadre d'OSGi) que nous avons conçu le service OSL [KB03].

4.2 OSL (OSGiServiceLookup)

OSL est un mécanisme élaboré de mapping et de recherche de services au dessus du mécanisme de localisation de services standard d'OSGi. Pour illustrer le principe de fonctionnement d'OSL, nous prenons un exemple d'application. Cette application est constituée de deux composants : *MorseED* et *MorseClient*. Le premier est un encodeur/décodeur de codes Morse. Il fournit un service appelé *MorseService*. Ce service définit deux opérations : *getCode()* et *getMessage()*. L'opération *getCode()* permet de calculer le code Morse correspondant à un message donné et *getMessage()* détermine le message correspondant à un code Morse. Le second composant (*MorseClient*) est le client qui permet de tester le fonctionnement de *MorseED*.

Supposons que ces deux composants ont été développés séparément, selon les deux spécifications suivantes :



Dans la recherche standard, le client *MorseClient* ne peut pas obtenir la référence du service *MorseService* car celui-ci n'est pas compatible avec le service demandé. Le rôle d'OSL est d'intervenir dans de telles situations pour proposer des alternatives aux services indisponibles. Concrètement, OSL propose de faire dynamiquement le mapping entre le service requis et le ou les services disponibles. Le mapping est sémantique, et ne peut pas être complètement automatisé. Un acteur humain doit participer à sa réalisation.

Le processus du mapping passe par plusieurs étapes. Ces étapes sont illustrées à travers notre exemple d'application.

4.2.1 Sélection du service de substitution

Lorsque OSL reçoit la requête d'un client, il tente d'abord d'obtenir le service demandé. Si le service est disponible, il crée un proxy représentant ce service et retourne sa référence au client. Ceci correspond à ce que nous avons expliqué dans la section précédente. Dans le cas contraire, OSL indique à l'utilisateur l'absence du service requis et lui propose de sélectionner un ou plusieurs services parmi ceux disponibles pour simuler le service manquant. Ceci est illustré par la Figure 45.

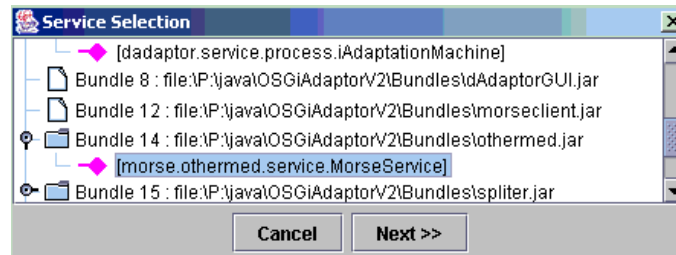


Figure 45. Sélection des services de substitution

4.2.2 Mapping des opérations

Après la sélection du ou des services de substitution, l'utilisateur doit spécifier le mapping entre les opérations. Pour chaque opération du service originalement requis, il est possible d'associer une ou plusieurs opérations de substitution. Il est possible de valider le mapping à ce stade, si à chaque opération correspond exactement une seule opération avec la même signature (même si les noms des opérations sont différents). Ceci n'est pas possible dans notre exemple d'application car les types de paramètres et les types de retour ne sont pas similaires. Cette étape est illustrée par la Figure 46.

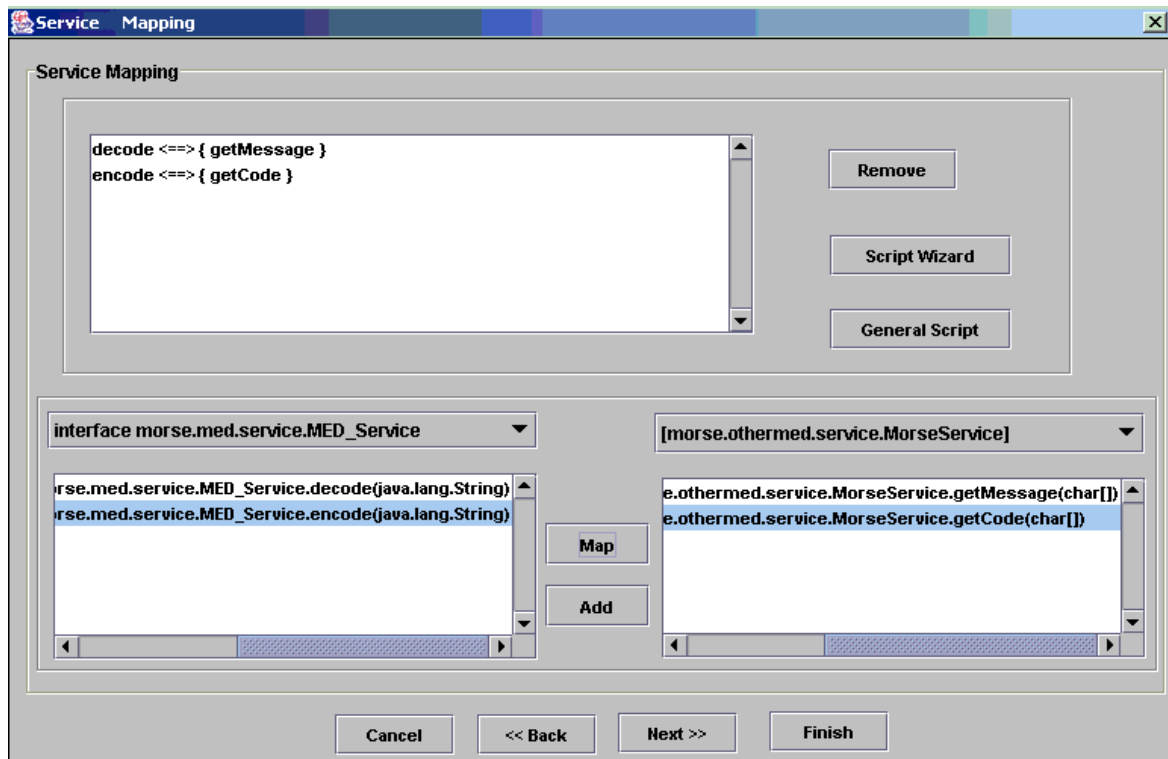


Figure 46. Mapping des opérations

4.2.3 Mapping des paramètres et des types de retour

Si les paramètres de l'opération originale et de celle qui est destinée à la remplacer ne sont pas identiques, il est nécessaire de définir la relation exacte entre ces informations. La Figure 47 présente l'interface qui permet de définir une telle relation. Dans le haut de l'interface, l'opération originale est décrite. L'utilisateur doit spécifier pour chaque paramètre de l'opération de substitution, une expression qui permet de le calculer. Par exemple, la première ligne de l'interface signifie que le premier paramètre de l'opération `getCode()` est équivalent à l'expression `"oldParam_0.toCharArray()"`, où `oldParam_0` est le premier paramètre de la méthode originale.

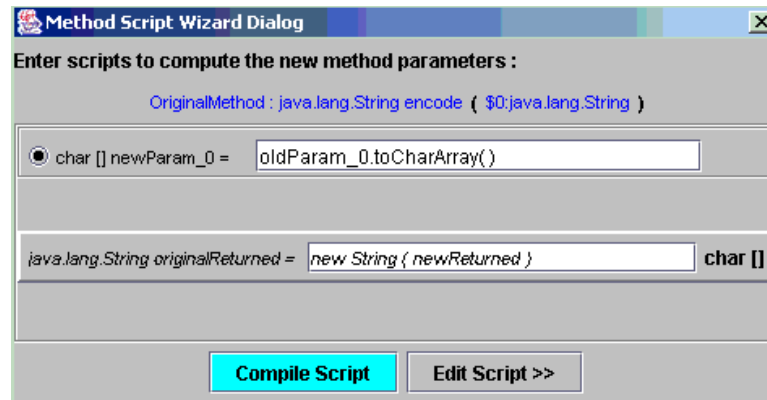


Figure 47. Mapping des paramètres et des types de retour

La dernière ligne de l'interface du mapping concerne les types de retour. Si le type de retour de l'opération attendue est différent de celui de l'opération de substitution, il est nécessaire de préciser comment calculer la valeur de retour attendue à partir de la valeur retournée par l'opération de substitution.

4.2.4 Utilisation des informations de mapping

Ce paragraphe explique comment les informations de mapping sont utilisées lorsque l'opération originale est appelée par le client.

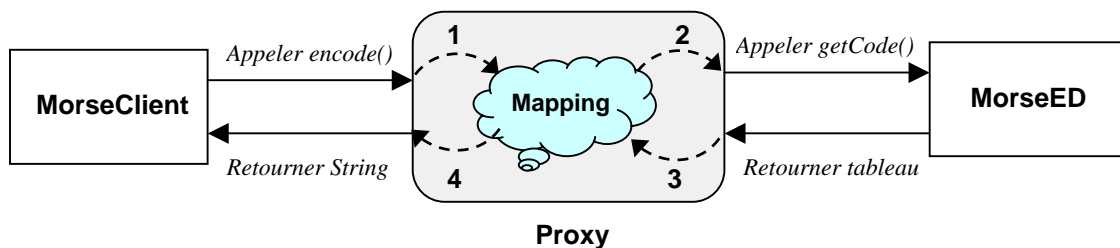


Figure 48. Utilisation des informations de mapping dans un appel

La Figure 48 montre les différentes étapes de l'appel :

1. A l'interception de l'appel d'une opération, le proxy détermine l'instance et l'opération de substitution et calcule les nouveaux arguments, éventuellement à partir de ceux passés par le client.
2. Le proxy appelle l'opération déterminée avec les arguments calculés.
3. Le proxy reçoit éventuellement le résultat de l'appel. Il calcule ensuite le résultat que le client attend en utilisant les informations du mapping.
4. Le proxy retourne enfin le résultat au client.

4.2.5 Composition de services

L'exemple de mapping que nous avons discuté est relativement simple. La situation devient plus complexe si le service requis par le client ne peut pas être entièrement couvert par un seul autre service. Par rapport à notre exemple d'application, supposons qu'il n'existe pas un autre service capable de fournir les mêmes fonctionnalités que *MED_Service*. Supposons d'un autre côté la présence de quatre services capables tous ensemble de simuler ces fonctionnalités. Ces services sont :

- Service d'encodage de messages (MES) : fourni par l'instance *MessageEncoder*, capable de remplacer la fonction d'encodage des messages (opération *encode()*).
- Service de découpage de codes (CSS) : fourni par l'instance *CodeSplitter*. Il découpe un code Morse en plusieurs items où chaque item correspond à une lettre.
- Service de décodage d'items (IDS) : fourni par l'instance *LetterDecoder*. Il détermine la lettre qui correspond à un item Morse.
- Service d'inversion de messages (MIS) : fourni par l'instance *MessageInverter*. Il permet d'inverser un message. Ce service est nécessaire si les codes Morse traités sont inversés.

Comme illustré par la Figure 49, l'opération originale *decode()* est simulée par trois autres opérations de substitution.

A screenshot of a software interface showing a mapping definition. The text is: `decode <==> { splitCode, decodeLetter, invertMessage }`. The text is highlighted in blue.

Figure 49. Mapping multiple d'opérations

L'utilisateur doit ensuite définir un script pour exprimer comment agir lorsque l'opération originale *decode()* est appelée, et comment calculer la valeur de retour le cas échéant. Le squelette du script est automatiquement généré par OSL. L'utilisateur doit compléter ce squelette avec la partie sémantique du mapping (le code Java). La Figure 50 montre un script potentiel pour notre exemple.

A screenshot of a code editor with a black background and green text. The code is as follows:

```
// --- User Manual Code ---
// 1 - Split the code
String array[] = splitter.splitCode(params);
// 2 - Decode Items
for(int i=0; i<array.length; i++)
{
    char c = decoder.decodeLetter(array[i]);
    finalReturnedResult+=c;
}
// 3- Invert the message
finalReturnedResult = inverter.invertMessage(finalReturnedResult);
```

Figure 50. Code de mapping

L'architecture de l'application après le mapping est illustrée par la Figure 51.

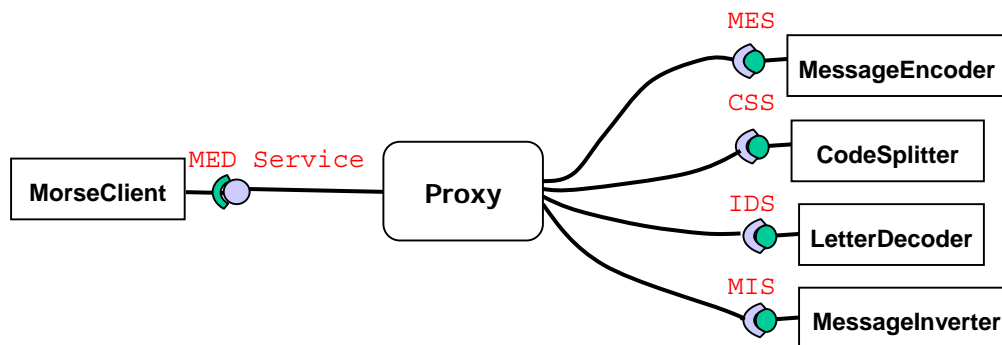


Figure 51. Architecture après le mapping

5 SYNTHÈSES ET CONCLUSION

5.1 Synthèse de l'expérimentation sur le modèle JavaBeans

Nous avons présenté dans ce chapitre notre première contribution pratique dans le cadre de cette thèse. Nous avons commencé par illustrer le support de reconfiguration que nous avons mis en œuvre pour le modèle JavaBeans : la DBeanBox. Ce support est une extension de la BeanBox, un environnement d'assemblage graphique des applications à base de JavaBeans. L'objectif de l'extension est de supporter la reconfiguration dynamique des applications assemblées avec cet environnement. Même si ce premier travail est basé sur la BeanBox, il est très facile de l'adapter à d'autres environnements d'assemblage graphique d'applications à base de JavaBeans. C'est ce que nous avons d'ailleurs fait, avec un minimum d'effort, pour le BeanBuilder [BBUI], un autre environnement développé par Sun Microsystems (le BeanBuilder peut être considéré comme une version plus élaborée de la BeanBox).

Dans la DBeanBox, nous avons pris en compte uniquement les événements échangés par les instances de Beans. Comme nous l'avons expliqué précédemment, la communication entre les instances peut également être réalisée à travers l'appel des méthodes de consultation et de modification des propriétés. Cet aspect de communication n'a pas été traité avec la DBeanBox.

5.2 Synthèse de l'expérimentation sur le modèle OSGi

Pour valider et consolider nos idées, nous avons mené une deuxième expérimentation sur le modèle OSGi. Nous avons le choix entre deux approches pour la mise en œuvre d'une solution de reconfiguration pour ce modèle :

- *La modification du framework OSGi* : augmenter le framework avec les mécanismes et les structures nécessaires pour supporter la reconfiguration dynamique. L'avantage de cette solution réside dans sa transparence par rapport aux applications. Le développement peut se faire sans considérer la reconfiguration car cette dernière est implantée à un niveau plus bas. En contre partie, l'inconvénient est la dépendance de la solution d'une implémentation particulière du framework OSGi.
- *La mise en œuvre d'un service dédié à la reconfiguration dynamique* : les applications doivent être développées en considérant ce service. On impose alors une manière de programmation particulière (par exemple, la demande des services ne doit pas être faite auprès du mécanisme de recherche standard d'OSGi). L'avantage de cette solution est sa portabilité car elle est indépendante d'une implémentation particulière d'OSGi.

Pour des raisons de portabilité et de simplicité, nous avons adopté la deuxième approche. Nous avons mis en œuvre principalement deux services OSGi :

- *OSGiAdaptor* : il implémente les différentes structures et algorithmes de reconfiguration. Il fournit une interface permettant aux utilisateurs de reconfigurer dynamiquement les applications. Il s'appuie sur un autre service appelé OSL.
- *OSL (OSGiServiceLookup)* : il fournit un mécanisme de recherche plus élaboré au dessus du mécanisme de recherche standard d'OSGi. Il incorpore des capacités de mapping qui permettent, si un service requis n'est pas disponible, de le simuler avec un ou plusieurs services de substitution. Les applications doivent utiliser explicitement le service OSL pour demander les services dont elles ont besoin.

Ces deux services peuvent être déployés et utilisés comme tout autre service OSGi.

5.3 Synthèse globale

Les opérations et les algorithmes de reconfiguration que nous avons développés sont pratiquement identiques dans les deux expérimentations que nous avons menées. Deux différences méritent d'être soulignées. La première réside dans la forme de solution mise en œuvre : dans le cas du modèle JavaBeans, nous avons modifié l'environnement BeanBox, cependant dans le cas d'OSGi, nous avons développé des services indépendants, dédiés à la reconfiguration. La deuxième différence concerne la gestion des messages envoyés pendant la

reconfiguration dynamique. Dans le cas du modèle JavaBeans, nous avons considéré la communication événementielle. Nous avons mis en place un mécanisme de gestion de files d'attente de messages. Dans le cas d'OSGi, la communication entre les instances de composants est basée sur des appels de méthodes. Sa nature synchrone était nettement plus compliquée à gérer.

Les opérations de reconfiguration implantées dans les deux expérimentations sont résumées dans le tableau suivant :

Propriété	JavaBeans	OSGi
Déconnexion	Déconnexion de deux ports, chaque port correspond à une méthode à appeler.	Déconnexion de deux ports, chaque port correspond à un ensemble de méthodes à appeler (interface).
Reconnexion	Reconnexion sélective (une seule méthode) ou totale (toutes les méthodes). La reconnexion peut affecter la source ou la destination des canaux de communication.	Reconnexion sélective ou totale. Cependant, la reconnexion ne peut affecter que la destination des canaux de communication.
Mapping	Mapping des propriétés qui représentent l'état des instances. Mapping des opérations publiques. Ceci est nécessaire lorsque les deux instances ne sont pas du même composant.	Idem mais le mapping peut concerner plusieurs instances remplaçantes au lieu d'une seule.
Remplacement des instances	Une instance peut être remplacée par une instance d'un même composant, ou d'un composant compatible. Le mapping est nécessaire dans ce dernier cas.	Une instance peut être remplacée par une instance d'un même composant ou par une ou plusieurs instances qui simulent l'instance originale. Le mapping est plus complexe que dans le cas du modèle JavaBeans.

Propriété	JavaBeans	OSGi
Changement d'interfaces	Une instance remplaçante peut avoir une interface différente de celle de l'instance remplacée. Le mapping des opérations est exploité à ce stade.	Idem. De plus, l'interface de l'instance remplacée peut être simulée par l'ensemble des interfaces des instances remplaçantes.
Type de communication	Événements	Appels de méthodes
Passivation/Activation	Pendant la reconfiguration, les événements envoyés sont estampillés et stockés dans des files d'attente. Ces événements sont envoyés à leur destination dans l'ordre de leur arrivée après la reconfiguration.	Les appels de méthodes interceptés pendant la reconfiguration sont bloqués sur des threads. Ils sont libérés après la reconfiguration.
Transfert d'état	L'état est représenté par les propriétés des Beans. Il est transféré de l'instance remplacée vers l'instance remplaçante. Les informations du mapping d'état sont exploitées à ce stade.	Les attributs qui représentent l'état d'une instance doivent être accessibles (publiques ou ayant des opérations de consultation (pour l'instance à remplacer), et des opérations de modification (pour l'instance remplaçante)).
Modification du framework	La BeanBox a été étendue avec des algorithmes et des structures de données pour supporter la reconfiguration.	Le framework OSGi n'a pas été modifié. La solution de reconfiguration se présente sous forme de deux services OSGi dédiés à la reconfiguration : OSGiAdaptor et OSL.

Tableau 5. Résumé des opérations de reconfiguration

5.4 Conclusion sur les expérimentations préliminaires

Du point de vue de la mise en œuvre, il est important de noter que la solution de reconfiguration pour le modèle OSGi a été développée à partir de rien. Ceci malgré le fait que,

comme nous l'avons expliqué, le principe de la solution, les algorithmes et les opérations de reconfiguration sont pratiquement identiques dans les deux expérimentations. Nous n'avons exploité de l'expérimentation sur le modèle JavaBeans que les idées. Nous pensons que ceci n'est pas suffisant, et qu'il est nécessaire de trouver une approche qui permet plus de réutilisation afin d'éviter de réimplanter à chaque fois la même chose.

Nous pensons que l'abstraction peut apporter une solution acceptable à ce problème de réutilisation. En arrivant à abstraire les éléments techniques et les concepts liés à chaque modèle, il devrait être possible de définir une approche qui favorise la réutilisation et qui peut être exploitée dans différents contextes.

Il est aussi nécessaire, pour atteindre un degré acceptable de généricité, que la solution à proposer soit séparée :

- ❑ des applications à développer : le programmeur ne doit s'occuper que de la logique de son application. Il doit être dégagé de la responsabilité de reconfiguration.
- ❑ des modèles visés : pour pouvoir appliquer la solution à plusieurs modèles cibles, il faut qu'elle soit indépendante des particularités de ces modèles.

Il est à noter que les deux solutions décrites dans ce chapitre ne fournissent aucun support d'automatisation. Un acteur humain doit explicitement décider et lancer les opérations de reconfiguration. La solution proposée doit supporter, d'un côté, des formalismes pour décrire les stratégies qui permettent d'automatiser la reconfiguration, et d'un autre côté, les mécanismes nécessaires pour raisonner sur ces stratégies et prendre des décisions cohérentes.

Le chapitre suivant est justement dédié à présenter la solution que nous proposons pour supporter la reconfiguration dynamique, et qui tente de répondre à ces besoins.

CHAPITRE 5

DYVA : UN NOYAU GENERIQUE POUR LA RECONFIGURATION DYNAMIQUE PRINCIPES DE CONCEPTION

PRÉAMBULE.

Ce chapitre décrit la solution de reconfiguration dynamique que nous avons développée dans le cadre de cette thèse. Cette solution constitue un aboutissement et une généralisation des différentes expérimentations que nous avons menées. Notre solution générale se présente sous forme d'une machine virtuelle, baptisée DYVA, qui prend en charge la reconfiguration dynamique. Les motivations derrière notre approche, l'architecture logique de notre machine et les concepts sur lesquels elle est basée, sont expliqués dans ce chapitre.

1 INTRODUCTION

Nous avons expliqué dans le chapitre précédent, qu'après l'ensemble des expérimentations que nous avons menées, notre objectif était de développer une solution de reconfiguration dynamique plus générale. Ce chapitre explique les détails de cette solution. Pour pouvoir développer une solution suffisamment réutilisable et séparée des applications à reconfigurer, nous proposons le concept de *machine virtuelle de reconfiguration dynamique*. Son rôle est de prendre en charge la responsabilité de reconfiguration, et de permettre aux développeurs de se concentrer sur la logique applicative.

Dans la suite, nous expliquons d'abord les motivations de notre approche. Nous présentons ensuite un aperçu de DYVA [KB04], notre machine de reconfiguration, et nous décrivons les éléments qui constituent son architecture logique. Nous illustrons à travers un exemple d'application le modèle de composants abstrait, sur lequel DYVA est basé, et qui permet d'augmenter sa généralité. Nous discutons dans la section suivante la capacité d'auto-reconfiguration de DYVA et nous présentons les éléments qui permettent de la supporter. Ce chapitre présente aussi la vue externe de DYVA. Cette vue est utilisée pour personnaliser DYVA pour un modèle de composants particulier.

Avant de conclure ce chapitre, nous discutons la solution que nous avons développée pour le problème de transfert d'état, et nous mettons l'accent sur le processus à suivre pour personnaliser et exploiter DYVA.

2 MOTIVATIONS

Avant de présenter DYVA, nous discutons d'abord dans cette section quelques solutions pour créer des applications à base de composants dynamiquement reconfigurables. Ceci permet de situer notre approche et de montrer son intérêt.

En général, pour développer une application pouvant être reconfigurée dynamiquement, plusieurs visions peuvent être considérées :

- La reconfiguration dynamique est prise en charge directement par l'application : le code qui assure la reconfiguration dynamique est intégré et confondu avec le code de l'application. Ceci correspond au principe des systèmes fermés que nous avons présentés auparavant. Pour développer une nouvelle application supportant la reconfiguration dynamique, il faut qu'elle intègre de la même manière le code de reconfiguration. Cette solution est très lourde et difficilement utilisable. Aussi, elle ne garantit pas la séparation et l'évolution du code applicatif, et n'assure pas non-plus la réutilisation du système de reconfiguration.
- La reconfiguration dynamique est prise en charge par le modèle sous-jacent. Ceci peut être obtenu en intégrant les fonctions de reconfiguration à la plate-forme d'exécution associée au modèle. Toutes les applications basées sur ce modèle, et s'exécutant au-dessus de sa plate-forme bénéficient automatiquement de la reconfiguration dynamique. Dans ce cas, la reconfiguration peut être vue comme une propriété non fonctionnelle au même titre que les transactions, la sécurité et la persistance. En considérant, par exemple, un modèle de composants basé sur Java, on peut éventuellement intégrer les fonctions de reconfiguration dans la machine virtuelle associée au modèle de composant, si elle existe, dans la machine virtuelle de Java ou même dans le système d'exploitation.
- L'inconvénient de la solution précédente est qu'il est nécessaire d'implanter la solution de reconfiguration pour chaque plate-forme d'exécution. Nous pensons qu'il est possible d'améliorer les capacités de réutilisation en considérant la reconfiguration dynamique à un niveau supérieur. En d'autres termes, les fonctions de reconfiguration doivent être supportées par un système dédié à cet effet, et intégrées aux applications développées ou aux plate-formes d'exécution associées aux modèles ciblés.

DYVA, notre machine de reconfiguration dynamique est justement basée sur cette troisième vision. Plusieurs questions peuvent être posées par rapport à la conception et à l'utilisation d'une telle machine :

- ❑ Quelles sont les opérations de reconfiguration qu'il faut mettre en œuvre, et surtout comment les séparer des applications à reconfigurer ?
- ❑ Quelles sont les exigences que doivent satisfaire les applications pour bénéficier de la reconfiguration dynamique ?
- ❑ Quelles sont les mécanismes nécessaires pour pouvoir lier les opérations de reconfiguration à l'application ou aux éléments à reconfigurer ?

Les sections suivantes sont destinées, entre autres, à illustrer l'architecture de DYVA et à répondre à ces questions.

3 DYVA : UNE CONCEPTION REFLEXIVE

Nous avons expliqué dans les chapitres précédents les avantages de la réflexion pour la construction de systèmes adaptables. Pour bénéficier de ces avantages, et proposer une solution simple, nous avons basé DYVA sur une conception réflexive.

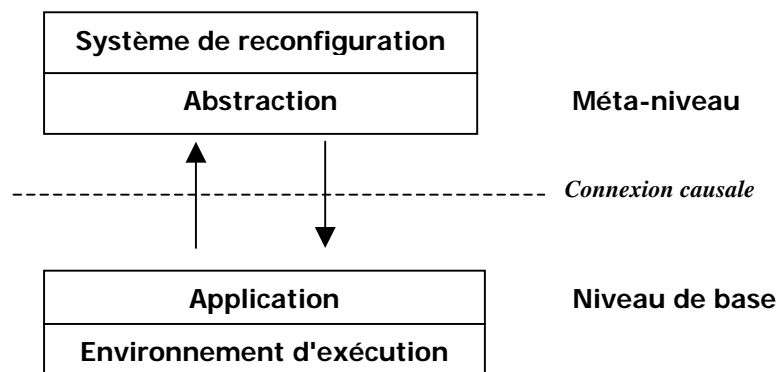


Figure 52. Conception réflexive de DYVA

Comme l'indique la Figure 52, deux niveaux peuvent être identifiés :

- ❑ *Niveau de base* : correspond aux applications à reconfigurer. Certains paramètres appartenant à l'environnement d'exécution peuvent être pertinents pour la reconfiguration (espace disque, mémoire, bande passante, charge du processeur, etc.). Ces paramètres doivent également être considérés par le niveau de base.

- *Méta-niveau* : il comporte d'un côté l'image abstraite de l'application, et d'un autre côté le système qui est responsable de réaliser la reconfiguration dynamique.

Selon le principe des systèmes réflexifs, ces deux niveaux sont causalement liés. Les modifications au niveau de l'application sont projetées sur son abstraction et vice-versa. Les sections suivantes expliquent en détails les éléments du méta-niveau, et montrent comment la connexion causale avec le niveau de base est matérialisée.

L'objectif de notre travail est de concevoir une machine virtuelle de reconfiguration dynamique. Cette machine, qualifiée de virtuelle, ne peut pas être exploitée en tant que telle. Elle doit être personnalisée pour un modèle de composants particulier pour qu'elle devienne effectivement opérationnelle. En prenant notre machine virtuelle DYVA comme point de départ, nous pouvons identifier plusieurs rôles d'exploitation :

- *Personnalisation de DYVA* : conception d'une couche pour envelopper DYVA et le spécialiser pour opérer sur un modèle de composants particulier.
- *Installation de DYVA* : réalisation du lien entre les applications à reconfigurer et la version personnalisée de DYVA. Ceci peut être fait explicitement par programmation, ou en utilisant des outils d'instrumentation.
- *Utilisation de DYVA* : reconfiguration dynamique des applications à travers les services fournis par DYVA.

Nous rappelons que dans le contexte de cette thèse, nous sommes partis des considérations suivantes :

- Une configuration est un ensemble d'instances de composants, potentiellement interconnectées.
- Une reconfiguration est une opération qui a pour but de changer la configuration courante.
- Une reconfiguration dynamique est une reconfiguration réalisée en exécution, et devant garantir l'intégrité de l'application à reconfigurer.

En analysant ces définitions, nous pouvons constater que notre système de reconfiguration adresse plus particulièrement la modification de l'aspect architectural des applications. Il gère les modifications à un méta-niveau. Ceci correspond à la logique employée dans les systèmes architecturalement réflexifs.

4 ARCHITECTURE INTERNE DE DYVA

Cette section présente l'architecture logique de DYVA. La Figure 53 illustre les différents éléments formant cette architecture.

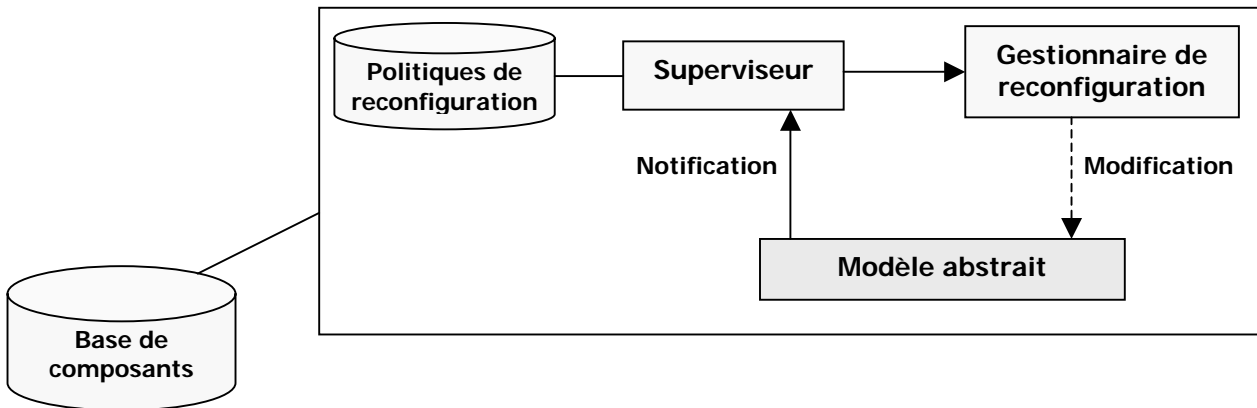


Figure 53. Architecture de référence de DYVA

Comme nous l'avons expliqué précédemment, un *modèle de composants abstrait* est utilisé pour augmenter l'indépendance du système de reconfiguration vis-à-vis des modèles de composants ciblés. L'élément principal de l'architecture est le *gestionnaire de reconfiguration*. Il fournit les opérations de reconfiguration de base et opère sur le modèle abstrait. Pour supporter l'auto-reconfiguration, une capacité de supervision et de raisonnement est nécessaire. Le *superviseur* est l'élément qui joue ce rôle. Il s'appuie sur un ensemble de *politiques* pour décider et lancer des opérations de reconfiguration. Ces différents éléments de l'architecture sont expliqués en détail dans les sous-sections suivantes.

4.1 Modèle abstrait d'application utilisé par DYVA

Dans les expérimentations présentées dans le chapitre précédent, les fonctions de reconfiguration opèrent sur les éléments d'un modèle particulier. Le modèle abstrait permet de concevoir une solution de reconfiguration générale, et d'éviter que les fonctions de reconfiguration soient figées sur un modèle particulier. La Figure 54 présente une vue simplifiée en UML [UML97] du modèle abstrait. Ce modèle se découpe en deux parties :

- ❑ Partie statique : correspond à la définition statique des composants en termes de ports fournis et requis. A ce niveau, les composants primitifs et composites sont définis de la même manière.
- ❑ Partie dynamique : reflète l'architecture dynamique de l'application. C'est à ce niveau que la différence est faite entre les composants primitifs et composites : seules les instances des composants composites sont autorisées à contenir d'autres instances.

Les éléments constituant le modèle abstrait sont présentés dans le paragraphe 4.1.2. L'application du paragraphe 4.1.1 sera utilisée pour les illustrer.

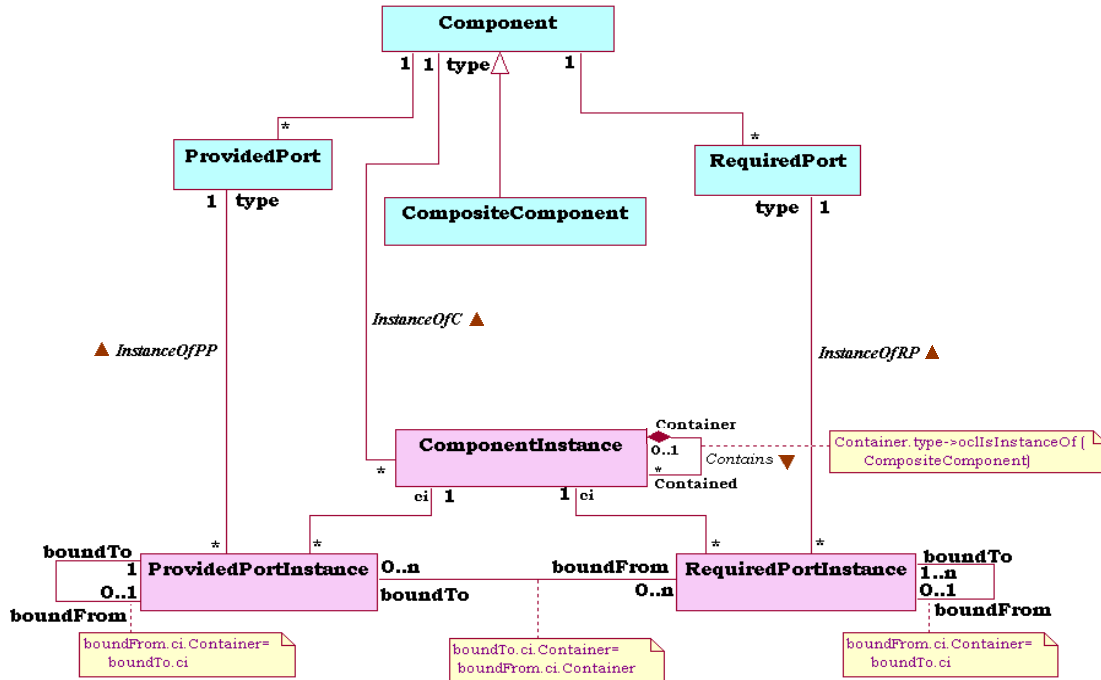


Figure 54. Modèle abstrait d'application

Il est important de souligner que le modèle abstrait a été introduit pour augmenter la réutilisabilité de DYVA. Notre objectif n'était pas de développer un modèle de composants exécutable (doté d'une plate-forme d'exécution). Nous ne prétendons pas non plus avoir mis au point un modèle de composants générique pour remplacer d'autres modèles.

4.1.1 Exemple d'application

La Figure 55 montre un exemple d'application représentée selon le modèle abstrait. Cet exemple sera utilisé pour illustrer les éléments du modèle.

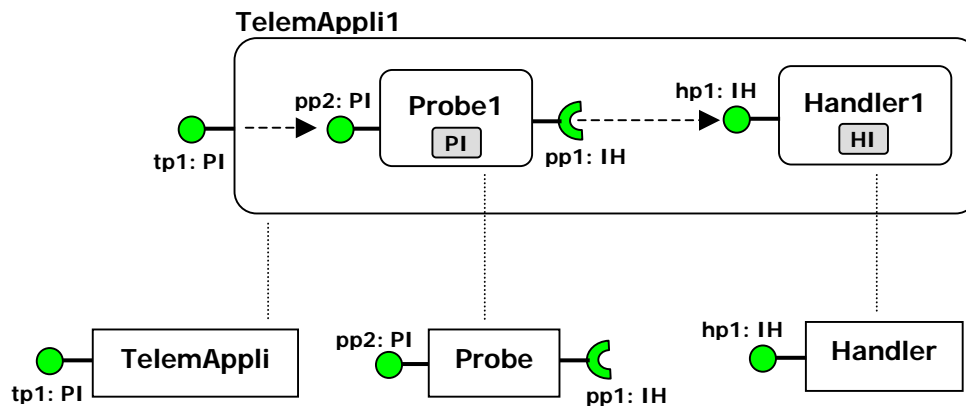


Figure 55. Exemple d'application

L'exemple montre une instance d'un composant composite (*TelemAppli1*) qui représente une application de télémessure. Cet instance est constituée de deux instances internes (*Probe1* et *Handler1*), interconnectées via leurs ports "*pp1*" et "*hp1*" ayant les deux comme type l'interface "*IH*". L'instance "*TelemAppli1*" fournit un port nommé "*tp1*". La fonctionnalité exposée à travers ce port est déléguée à l'instance interne "*Probe1*", grâce à la liaison entre les deux ports "*tp1*" et "*pp2*".

4.1.2 Éléments du modèle abstrait

Les points suivants illustrent les éléments les plus importants du modèle abstrait à travers cet exemple d'application :

- *Composant* : représente un type qui peut être instancié. Il est caractérisé par un nom et un ensemble de ports fournis et/ou requis. Par exemple, le composant "*Handler*" fournit un port nommé "*hp1*" de type "*IH*". Un composant peut être primitif ou composite.
- *Composant primitif* : il est caractérisé par une classe qui représente son implémentation. "*Probe*" est un composant primitif, implémenté par la classe "*PI*".
- *Composant composite* : défini statiquement en fonction de ses ports fournis et requis. Le contenu d'un composant composite est décidé en exécution.
- *Port fourni* : correspond à une interface implémentée par le composant. Chaque port (fourni ou requis) du composant possède un nom unique. Deux composants différents peuvent avoir deux ports avec le même nom. Ceci ne cause pas une confusion car un port est identifié par son nom et le nom de son composant.
- *Port requis* : caractérise une fonctionnalité requise par le composant. Dans le cas d'un composant primitif, un port requis correspond à un attribut dans la classe d'implémentation du composant, qui référence une autre instance de composant (par exemple, le port "*pp1*" du composant "*Probe*"). Dans le cas d'un composant composite, le port requis peut représenter (être lié à) un ou plusieurs ports requis de ses instances internes.
- *Instance de composant* : un composant peut avoir un nombre quelconque d'instances. L'instance du composant racine représente le point d'entrée de l'application. A l'image de son composant, une instance est caractérisée par un ensemble d'instances de ports fournis et/ou requis. Une instance d'un port requis (par une instance de composant) peut être connectée à une instance d'un port fourni (par une autre instance de composant de même niveau : appartenant au même composite). Par exemple, l'instance du port requis "*pp1*" est connectée à l'instance du port fourni "*hp1*". Une instance d'un port requis peut être aussi *liée* à une instance d'un port requis par le composant englobant. Une instance d'un port fourni par un composant composite peut être liée à une instance d'un port fourni par l'une de ses instances internes. Ceci peut être vu comme une délégation de responsabilité.

Il est important de noter que le terme *instance* ne veut pas forcément qualifier une entité à l'exécution. L'instanciation signifie simplement l'utilisation d'une entité de base dans un contexte particulier. L'exemple présenté par la Figure 55 peut être une application en exécution et peut être aussi une description statique (sous forme d'un ADL par exemple) qui pourra être exécutée.

4.1.3 Annotations

Il est possible d'attacher à chaque élément du modèle un ensemble d'annotations. Une annotation matérialise une propriété de l'élément auquel elle est attachée. Les propriétés associées aux éléments du modèle permettent d'augmenter la capacité d'auto-reconfiguration. Par exemple, on peut attacher à un composant "A" une annotation qui décrit la valeur de la bande passante nécessaire pour son exécution. Si à un moment donné, après la mise en service de l'application, le système constate la chute de la bande passante au dessous de la barre minimale, il déclenche automatiquement le remplacement dynamique de "A" par une autre instance qui nécessite moins de bande passante.

4.2 Gestionnaire de reconfiguration

Le gestionnaire de reconfiguration est la partie centrale de l'architecture. Il fournit les fonctions de base du système de reconfiguration. Ces fonctions opèrent sur le modèle abstrait (et non pas directement sur les applications) ce qui garantit l'indépendance du système vis-à-vis d'une application particulière ou d'un modèle de composants particulier. Dans ce qui suit, nous décrivons brièvement ces fonctions.

- ❑ *Passivation d'une instance de composant* : nécessaire avant toute opération sur l'instance. Elle permet de verrouiller les canaux de communication concernant l'instance, et de s'assurer qu'elle est dans un état stable. L'instance doit répondre affirmativement quand elle est dans un état stable. Si sa réponse est négative, la passivation ne peut pas avoir lieu.
- ❑ *Activation d'une instance de composant* : permet d'ouvrir les canaux de communication concernant l'instance, et de traiter les messages envoyés après sa passivation.
- ❑ *Supprimer une instance de composant* : l'instance est d'abord mise à l'état passif avant sa suppression. Si la passivation ne réussit pas, la suppression ne peut avoir lieu. La suppression d'une instance entraîne systématiquement la suppression de ses connexions avec les autres instances.
- ❑ *Supprimer un composant* : supprimer une instance n'affecte pas les autres instances du même composant. Parfois, au lieu de supprimer une seule instance, il est souhaitable de supprimer plutôt un composant. La suppression d'un composant entraîne la suppression de toutes ses instances.

- *Déconnexion dynamique* : deux types de déconnexion sont assurés par le gestionnaire de reconfiguration :
 - Déconnexion au niveau port : il faut dans ce cas spécifier le port source et le port cible (de l'instance de composant) concernés par la déconnexion.
 - Déconnexion au niveau instance : tous les ports de l'instance en question sont déconnectés.

La déconnexion est forcément précédée par la passivation de l'instance cible. Elle ne peut avoir lieu qu'après la réussite de cette opération.

- *Connexion dynamique* : connecter deux ports compatibles de deux instances de composants.
- *Reconnexion dynamique* : peut être vue comme une déconnexion suivie d'une connexion. Le gestionnaire supporte les reconnexions au niveau port ou au niveau instance.
- *Ajouter un composant* : cette opération est nécessaire pour pouvoir créer des instances de nouveaux composants non encore chargés par l'application. Le nom du composant à ajouter doit être spécifié.
- *Création d'une instance de composant* : nécessaire pour étendre ou adapter les fonctionnalités de l'application. L'utilisateur doit spécifier le nom du composant qu'il souhaite instancier.
- *Remplacer une instance de composant* : pour des raisons de performance, d'adaptabilité ou de correction d'erreurs, il est parfois nécessaire de remplacer une instance par une autre instance qui répond mieux aux besoins de l'application. L'utilisateur doit spécifier le nom de l'ancienne et de la nouvelle instance. L'utilisateur peut également, au lieu de spécifier le nom de la nouvelle instance, préciser le nom d'un composant. Dans ce cas, le système crée automatiquement une nouvelle instance de ce composant et l'utilise pour le remplacement. Le remplacement d'instances est lui-même un processus complexe composé de plusieurs activités. Ce processus peut être résumé comme suit :
 - S'assurer qu'une correspondance a été définie explicitement par l'utilisateur, si l'ancienne et la nouvelle instance ne sont pas de même type.
 - Créer l'instance remplaçante si elle n'existe pas.
 - Mettre à l'état passif l'instance à remplacer.
 - Transférer l'état de l'instance à remplacer vers l'instance remplaçante, en suivant les règles de correspondance. Cette étape est ignorée si l'une des instances n'est pas déclarée à état.
 - Reconnecter la nouvelle instance à la place de l'ancienne.

- Activer la nouvelle instance, ce qui permet de traiter les requêtes en attente, et supprimer éventuellement l'ancienne instance.

4.3 Capacité d'auto-reconfiguration

L'auto-reconfiguration signifie la capacité du système de reconfiguration à prendre des décisions de reconfiguration cohérentes, et à matérialiser ces décisions sans l'intervention d'un acteur humain. L'automatisation des décisions et de leur application facilite considérablement l'utilisation du système, et permet de décharger les administrateurs des tâches d'observation, d'analyse et de décision. L'importance de l'automatisation augmente lorsque ces tâches deviennent fréquentes. La Figure 56 montre le cycle d'automatisation. Ce cycle est constitué des trois étapes : *observation*, *décision* et *action*.

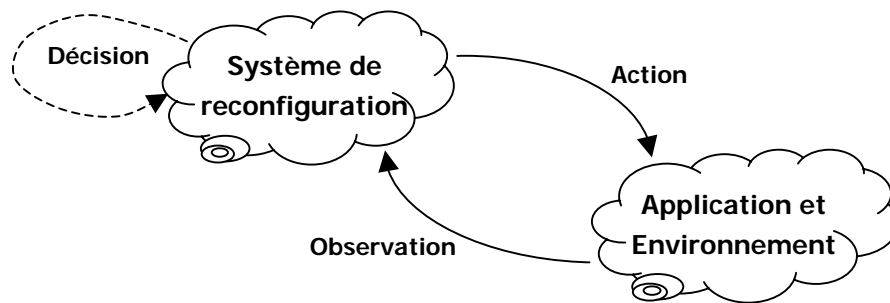


Figure 56. Cycle d'automatisation

4.3.1 Stratégies de reconfiguration

Lorsque la reconfiguration est lancée manuellement, l'utilisateur emploie un ensemble de connaissances implicites (pourquoi reconfigurer, qu'est-ce qu'il faut reconfigurer, sous quelles conditions, etc ?). Pour automatiser la reconfiguration, il est nécessaire de trouver un moyen pour exprimer ces connaissances et les rendre explicites. Dans DYVA, nous avons utilisé le concept de *règles de reconfiguration* pour ces fins.

La structure que nous avons développée pour représenter les règles est très simple. Elle est inspirée du formalisme ECA (Événement/Condition/Action) et des déclencheurs (triggers) utilisés dans les bases de données actives [Eri97]. La forme générale d'une règle de reconfiguration est la suivante :

```
ON <événement> IF <condition> THEN <action>
```

- *Événement* : requête ou information provenant de l'application à reconfigurer ou des capteurs/sondes surveillant son environnement d'exécution.

- ❑ *Condition* : prédicat évalué sur les propriétés des éléments du modèle abstrait (donc sur les éléments de l'application), et/ou sur les valeurs remontées par l'événement.
- ❑ *Action* : opération ou bloc d'opérations de reconfiguration pouvant porter sur les éléments du modèle. Une action peut aussi être la délégation de la décision à l'administrateur, en affichant simplement un message expliquant l'événement reçu, les conditions à vérifier, et si possible, le résultat d'évaluation des conditions. L'administrateur peut prendre ensuite la décision adéquate.

L'exemple suivant montre une règle de reconfiguration relative à l'application de télémétrie.

```
[RULE]
ON: OVERLOAD_EVENT
IF: ((LOAD > 80) AND (Handler.version = 1))
THEN: replace Handler1 NewHandler1
```

Cette règle sera traduite de la manière suivante : lorsqu'un événement nommé "OVERLOAD_EVENT" est reçu, et si la condition est vérifiée, il faut remplacer l'instance "Handler1" par la nouvelle instance "NewHandler1" (instance d'un autre composant). La condition porte sur la valeur transmise par l'événement ("LOAD") et sur une propriété du composant "Handler". Pour que cette condition soit vérifiée, il faut que la valeur transmise par l'événement (correspondant à la charge du processeur) soit supérieure à la valeur "80", et que le composant "Handler" soit dans sa première version.

Pour plus de souplesse, la base des règles a été conçue pour évoluer dynamiquement. Les règles peuvent ainsi être ajoutées, modifiées ou retirées en exécution.

4.3.2 Moteur d'analyse des règles (superviseur)

Les règles que nous avons présentées dans la section précédente incarnent la logique de reconfiguration. L'élément de l'architecture qui est responsable d'interpréter et d'exploiter cette logique est le *superviseur*. Il reçoit les événements envoyés par l'application et/ou son environnement. Il incarne un moteur de raisonnement qui analyse les règles et prend des décisions de reconfiguration. Une décision peut avoir l'une des formes suivantes :

- ❑ Appel des opérations du gestionnaire de reconfiguration spécifiées dans le bloc de l'action associée à la règle.
- ❑ Délégation de la décision à un acteur externe. Ceci peut avoir lieu dans deux cas : (1) le superviseur rencontre des problèmes pour interpréter la règle, ou, (2) la règle indique explicitement qu'il faut déléguer la décision (action "DELEGATE").

4.3.3 Base de composants

Dans certaines situations, et pour déclencher des opérations de reconfiguration complexes, la base des règles n'est pas suffisante. Par exemple, on souhaite créer une règle qui exprime qu'il faut remplacer une instance "A" par n'importe quelle autre instance compatible, sans vouloir préciser le nom de cette nouvelle instance. Dans ce genre de situations, le système doit être capable de trouver ou de créer une instance compatible pour appliquer la règle.

Le système nécessite donc une base qui permet de structurer les différents composants de l'application, et les composants qui peuvent potentiellement être ajoutés (à cette application). Cette base doit répondre aux requêtes suivantes :

- Retourner la liste des composants compatibles avec un composant donné, et vérifiant un prédicat. Le prédicat peut être éventuellement vide.
- Vérifier si deux composants sont compatibles.
- Retourner les règles de correspondance entre deux composants compatibles. Les règles de correspondance sont éventuellement nécessaires même entre deux versions du même composant. En effet, deux versions du même composant partagent les mêmes interfaces, cependant, leur structure interne peut être différente. Une correspondance est alors nécessaire pour pouvoir transférer l'état des instances.

Tout système répondant à ces besoins peut être éventuellement utilisé. Dans le cadre de DYVA, nous avons conçu une *base de composants* ayant une structure très simple. La Figure 57 illustre l'organisation de cette base.

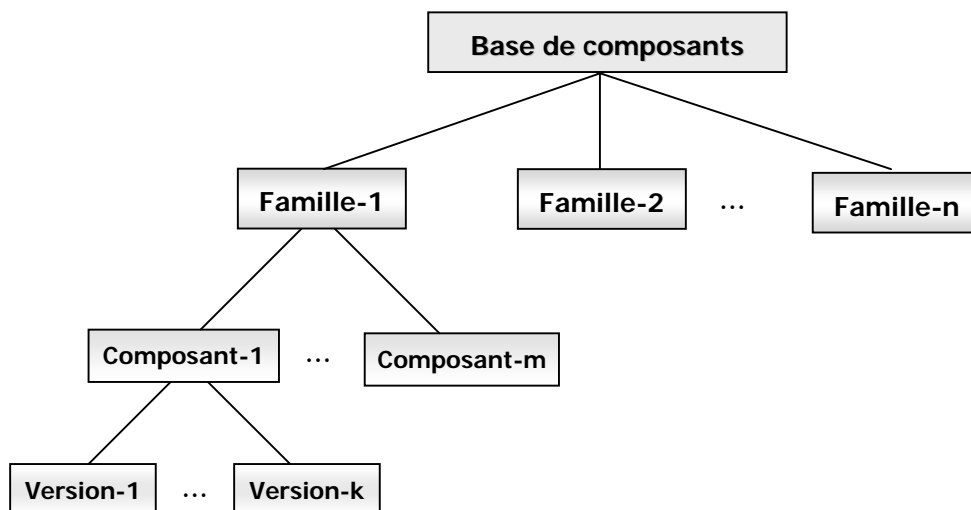


Figure 57. Organisation de la base de composants

Les composants sont organisés en plusieurs familles. Une famille regroupe des composants sémantiquement compatibles (assure la même fonction, et peuvent donc être interchangeables), mais ayant des interfaces différentes. Des règles de correspondance (entre les interfaces) sont nécessaires pour l'inter-remplacement des composants d'une même famille.

Chaque composant peut avoir plusieurs versions. Toutes les versions d'un même composant sont sémantiquement et syntaxiquement compatibles (assurent la même fonction et ont les mêmes interfaces). Elles peuvent être différentes uniquement du point de vue de l'implémentation. Des règles de correspondance (entre les structures de données) restent tout de même nécessaires pour le transfert d'état entre les instances de deux versions différentes (l'implémentation peut être différente, donc la structure interne peut être aussi différente).

5 VUE EXTERNE DE DYVA

Nous avons présenté dans les sections précédentes les éléments architecturaux qui composent notre système de reconfiguration. Nous avons expliqué que ces éléments opèrent sur un modèle abstrait. Cette section se concentre sur la facette externe de DYVA.

Pour personnaliser DYVA, on ne doit normalement pas se préoccuper de sa composition interne. On peut l'exploiter en utilisant un ensemble d'interfaces et de classes utilitaires fournies. La Figure 58 présente la vue externe de DYVA.

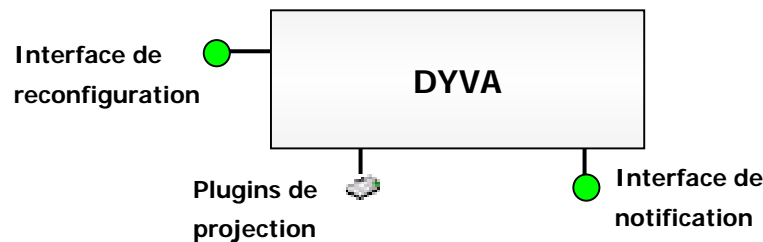


Figure 58. Vue externe de DYVA

La section 7 ci-dessous explique le processus de personnalisation et d'utilisation de DYVA dans le contexte d'un modèle de composants particulier. Dans cette section nous présentons les interfaces qui peuvent être utilisées dans ce processus. Comme nous l'avons expliqué, le modèle abstrait représente l'image de l'application à administrer. Les modifications au niveau de l'application doivent être reflétées sur sa représentation abstraite et vice-versa. L'interface de notification fournie par DYVA permet aux applications de notifier certaines modifications au niveau abstrait. Puisque DYVA n'est pas destinée à un modèle particulier, la notification inverse (la projection des modifications sur l'application) ne peut pas être définie d'une manière définitive. Nous avons conçu une structure de plugins pour permettre d'adapter les opérations de projection en fonction du modèle de composant ciblé.

5.1 Interface de notification

L'interface de notification définit plusieurs méthodes qui permettent à l'application de notifier quelques événements. Un événement est envoyé dans l'une des situations suivantes :

- ❑ Une instance de composant est créée : un événement spécifiant la référence de l'instance créée est envoyé.
- ❑ Une instance de composant est supprimée : un événement spécifiant la référence de l'instance supprimée est envoyé.
- ❑ Une connexion entre deux ports de deux instances est créée : un événement est levé. Cet événement doit préciser le nom du port source, le nom de l'instance source, le nom du port cible et le nom de l'instance cible.
- ❑ Une connexion entre deux ports de deux instances est supprimée : un événement est levé. Cet événement doit préciser le nom du port source, le nom de l'instance source, le nom du port cible et le nom de l'instance cible.

Ces quatre types d'événement concernent l'architecture de l'application. Ils permettent de mettre en œuvre un système architecturalement réflexif, comme celui implémenté par [CS+99]. Deux autres types d'événement peuvent être également notifiés. Le premier concerne les interactions entre les instances. Il permet ainsi de mettre en place le principe de réflexion comportementale. Le dernier type d'événement est réservé aux changements qui affectent l'environnement d'exécution de l'application.

- ❑ *Une interaction entre deux instances* : lorsqu'une instance appelle une opération définie dans une autre instance, l'appel est intercepté et redirigé vers le méta-niveau. Des traitements peuvent d'abord être réalisés, l'opération originale peut ensuite être appelée, avant de faire d'éventuels traitements pour finir. Pour plus de flexibilité, il est possible de configurer DYVA pour envoyer un événement lors de l'appel de chaque opération ou d'envoyer un événement de façon sélective.
- ❑ *Un changement significatif dans l'environnement d'exécution* : la configuration de l'application dépend parfois de certains paramètres de l'environnement. Il est nécessaire, d'un côté, de disposer d'outils pour surveiller l'évolution de ces paramètres, et d'un autre côté, d'un moyen qui permet à ces outils de communiquer les changements au système de reconfiguration pour les analyser et réagir. Nous avons mis en œuvre une structure simple qui représente un événement de l'environnement. Cette structure décrit l'événement (nom, date, explication) et comporte éventuellement un tableau de valeurs associées à l'événement.

5.2 Interface de reconfiguration

L'interface de reconfiguration permet à l'administrateur d'invoquer les opérations de reconfiguration fournies par DYVA. Les opérations définies dans cette interface correspondent à celles implémentées par le gestionnaire de reconfiguration (cf. section 4.2).

5.3 Plugins de projection

Nous avons expliqué que les fonctions du gestionnaire de reconfiguration opèrent sur la représentation abstraite de l'application. Il est évidemment nécessaire de projeter les modifications établies au niveau abstrait sur l'application concrète. Dans un esprit réflexif, ceci correspond à la mise en œuvre de la connexion causale dans le sens descendant.

Dans le cas de l'interface de notification et de celle de reconfiguration, les opérations définies travaillent sur les éléments du modèle abstrait. Ces opérations sont ainsi indépendantes d'un modèle particulier, ce qui nous a permis de les définir définitivement.

La projection des modifications, d'un autre côté, dépend du modèle ciblé et de l'application concrète. Partant du principe que notre objectif est d'adresser des modèles différents, les opérations qui projettent les modifications sur les applications concrètes ne peuvent pas être définies définitivement, et pour tous les modèles. Considérons par exemple l'opération de déconnexion. Lorsque cette dernière est réalisée au méta-niveau, elle doit être matérialisée sur l'application. Si on prend une application en JavaBeans, la déconnexion se fait en dés-enregistrant l'instance cible de la liste des auditeurs de l'instance source. Si on prend une application (instrumentée) en OSGi, la déconnexion se fait en appelant une méthode spécifique de l'instance source. Enfin, si on prend une application basée sur le modèle Fractal [FRACTAL], la déconnexion se fait en appelant la méthode du *contrôleur de binding* associé à l'instance source.

Pour garder notre système de reconfiguration suffisamment générique, nous l'avons conçu d'une manière à ce que les opérations de projection puissent être intégrées sous forme de plugins. Ceci permet d'adapter les fonctions de projection en fonction du modèle adressé. La liste des plugins requis pour exploiter DYVA pour un modèle particulier est présentée dans la section 7.

6 PROBLEME DE TRANSFERT D'ETAT

Le transfert d'état est l'un des problèmes les plus délicats auquel il faut faire face pendant la reconfiguration dynamique. La difficulté réside dans le fait que l'état est spécifique et interne à chaque instance, et qu'il est par conséquent difficile de trouver une solution générale sans la participation de cette dernière.

Dans le contexte de cette thèse, nous avons développé une solution basée sur la représentation abstraite de l'état. Chaque instance de composant au niveau abstrait contient l'état de l'instance concrète. La Figure 59 présente le modèle abstrait de l'état.

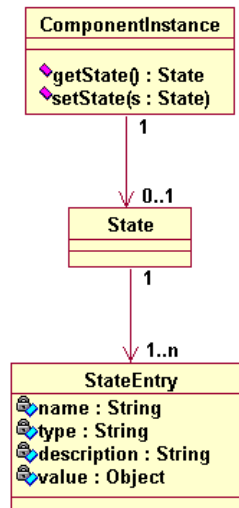


Figure 59. Modèle abstrait de l'état

Si une instance est à état, elle doit fournir deux opérations pour consulter et modifier son état. L'état en lui-même est vu simplement comme une liste d'enregistrements. Chacun représente un attribut, et, est décrit par un nom, un type, une description et une valeur. Pour la synchronisation de l'état de l'instance avec sa représentation abstraite, nous avons le choix entre deux solutions :

- ❑ Synchronisation immédiate : chaque fois qu'un élément faisant partie de l'état change au niveau de l'application, l'information est remontée au niveau abstrait. Cette solution assure une représentation fidèle de l'état à tout moment. Elle peut causer en contre partie une surcharge de l'application et peut dégrader considérablement ses performances.
- ❑ Synchronisation différée : il est possible de considérer qu'au niveau abstrait, on a besoin de l'état uniquement au moment de la reconfiguration. Dans ce cas, la synchronisation est établie uniquement à ce moment-là. C'est cette deuxième approche que nous avons adoptée dans DYVA.

Nous sommes partis du principe que l'état est sémantique, et que sa spécification dépend de l'application, du contexte d'utilisation et même du développeur des composants. Nous avons décidé de déléguer la responsabilité de spécification de l'état au développeur lui-même. Nous avons mis au point un outil pour l'assister et l'aider à réaliser cette spécification.

L'outil est spécifique aux composants développés en Java. Son rôle est d'introspecter la classe d'implémentation du composant, et d'afficher la liste des attributs qui font

potentiellement partie de l'état. Le développeur peut sélectionner ensuite les attributs qui constituent effectivement l'état. L'outil se charge finalement de générer et d'intégrer automatiquement les deux opérations `getState()` et `setState()` dans la classe d'implémentation du composant.

7 PROCESSUS DE PERSONNALISATION ET D'UTILISATION

Nous expliquons brièvement dans cette section comment personnaliser et utiliser DYVA pour supporter la reconfiguration dynamique dans le contexte d'un modèle de composants particulier. Cette personnalisation doit être faite par un spécialiste de ce modèle.

Comme le montre la

Figure 60, la personnalisation consiste à définir des interfaces spécifiques au dessus des interfaces fournies par DYVA. Elle consiste aussi à implanter les plugins de projection associés au modèle.

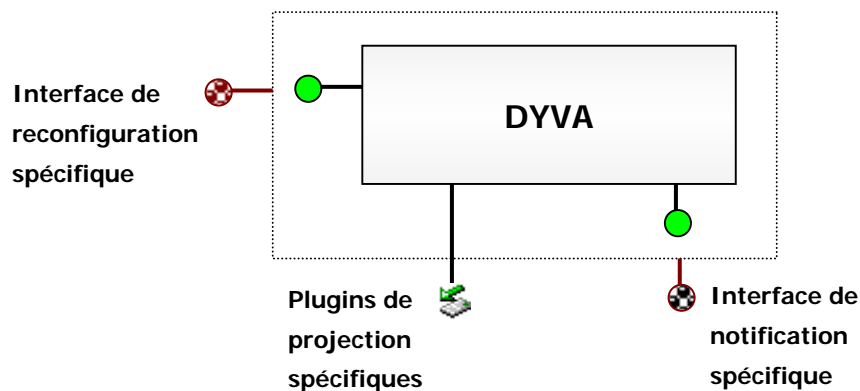


Figure 60. Personnalisation de DYVA

- **Interface de notification spécifique** : définit des opérations qui permettent aux applications d'envoyer des événements au système de reconfiguration. Ces opérations peuvent être basées sur les opérations définies dans l'interface de notification de DYVA. Les nouvelles opérations peuvent être un simple renommage des opérations de notification de DYVA. Elles peuvent être aussi plus complexe qu'un simple renommage. Par exemple, on décide de réaliser un filtrage sur les notifications envoyés par l'application ou par son environnement.
- **Interface de reconfiguration spécifique** : définit les opérations de reconfiguration associées au modèle adressé. Ces opérations spécialisent celles définies dans l'interface de reconfiguration de DYVA. L'interface de reconfiguration spécifique définit aussi de nouvelles opérations pour traiter les concepts particuliers du

modèle adressé. Par exemple, supposons que dans un modèle donné, on a le concept de sous-application, définie comme un ensemble d'instances interconnectées. On souhaite ainsi définir une opération, associée à ce concept, et permettant à titre d'exemple de supprimer une sous-application. Cette opération se charge de mettre à l'état passif les instances d'une sous-application, de les déconnecter, plus de demander leur suppression effective.

- **Plugins de projection** : constituent la partie la plus complexe et la plus importante du processus de personnalisation. Ils permettent de projeter les modifications introduites au modèle abstrait sur l'application concrète. L'ensemble de plugins à concevoir, qui reflètent chacun une opération de projection, peut être résumé par les points suivants :
 - *Plugin de binding* : permet d'établir et de supprimer les connexions entre deux ports de deux instances.
 - *Plugin d'instanciation* : responsable de la création et de la suppression d'instances de composants.
 - *Plugin de déploiement* : responsable de déployer de nouveaux composants, et de supprimer de l'application les composants inutiles.
 - *Plugin de gestion d'état* : responsable de retourner l'état d'exécution d'une instance donnée. Il est aussi responsable d'initialiser l'état d'une instance de composant.
 - *Plugin de gestion de communication* : responsable de la passivation et de l'activation des instances de composants. Il assure la sauvegarde des messages envoyés pendant la reconfiguration, et le traitement de ces messages dans le bon ordre après la reconfiguration.

Le chapitre 7 est consacré à discuter un cas réel de personnalisation de DYVA pour le modèle OSGi. Il présente en détail la spécification de l'interface de notification, de celle de reconfiguration et explique le fonctionnement des plugins de projection associés à OSGi.

8 CONCLUSION

Nous avons présenté dans ce chapitre DYVA, notre système de reconfiguration dynamique. Nous avons surtout insisté sur son architecture interne, sur sa capacité d'auto-reconfiguration et sur sa vue externe.

Pour assurer l'ouverture et la réutilisabilité de notre système, nous l'avons basé sur un modèle de composants abstrait. Ce modèle incarne l'image de l'application à reconfigurer et permet de traiter d'une manière homogène, des applications issues de modèles de composants différents.

Nous avons favorisé dans notre approche l'automatisation. Cette automatisation a deux facettes :

- *Instrumentation des applications* : cela consiste à introduire des modifications dans une application déjà développée pour qu'elle puisse répondre aux besoins du système de reconfiguration. Ceci concerne l'introduction du code qui permet de réaliser les notifications, et celui qui est responsable du transfert d'état.
- *Auto-reconfiguration* : nous avons mis en place un moteur de raisonnement qui permet de reconfigurer automatiquement les applications. Ce moteur reçoit des notifications, et prend des décisions en fonction de stratégies exprimées sous forme de règles spécifiées à l'avance.

Ce chapitre a expliqué aussi la solution que nous avons développée pour le problème de transfert d'état. Cette solution est basée sur un modèle abstrait de l'état. Elle permet, grâce à ce modèle, d'avoir une vision commune de la notion d'état.

La dernière partie de ce chapitre a brièvement expliqué le processus de personnalisation de notre système de reconfiguration pour un modèle de composants particulier. Cette personnalisation se fait par la définition d'une interface de notification, d'une interface de reconfiguration et d'un ensemble de plugins de projection spécifiques au modèle ciblé.

Le chapitre suivant est consacré à décrire l'implémentation de notre système de reconfiguration. Il explique les interfaces, les structures de données et les outils qui ont été mis en œuvre.

CHAPITRE 6

DYVA : IMPLEMENTATION

PRÉAMBULE.

Ce chapitre décrit l'implémentation de notre système de reconfiguration. Il explique d'abord les détails d'implémentation de la base de composants. Puis il illustre l'implémentation des différents éléments de l'architecture. Nous présentons également, dans ce chapitre, les différents outils que nous avons développés comme support à notre approche.

1 INTRODUCTION

Les chapitres précédents ont expliqué le principe de notre approche. Nous avons réalisé, dans cette thèse, une implémentation en Java pour démontrer la faisabilité et la validité de cette approche. Ce chapitre présente les éléments les plus importants de cette implémentation.

Nous nous concentrons principalement, dans la suite, sur les interfaces, les structures et les outils que nous avons développés comme support à notre approche. Nous commençons par une description détaillée de la base de composants, sur laquelle DYVA est basé. Puis nous décrivons les différents éléments de l'architecture, comme le modèle abstrait, les interfaces et les classes principales formant l'architecture. Pour simplifier la spécification de l'état, et la génération des opérations qui permettent de consulter et de modifier l'état d'une instance, nous avons implémenté un outil qui sera présenté après l'illustration de l'architecture. Avant de conclure ce chapitre, nous décrivons brièvement l'interface graphique que nous avons implémentée, pour simplifier le contrôle et la reconfiguration dynamique des applications.

2 IMPLEMENTATION DE L'ARCHITECTURE

2.1 Modèle abstrait

L'implémentation du modèle abstrait a été automatiquement générée à partir du modèle présenté dans la section 4.1 (chapitre 5). Nous avons créé le schéma XML [XMLS] correspondant au modèle, puis nous avons généré les classes Java à partir du schéma en utilisant l'outil Castor [CASTOR]. Comme nous l'avons expliqué dans le chapitre précédent, chaque élément du modèle peut être décrit par un ensemble de méta-données. La classe "Annotation" permet de gérer les méta-données associées à un élément donné. Tous les éléments du modèle héritent de cette classe. La classe "Annotation" implémente l'interface "IAnnotation" définie ci-dessous :

```
public interface IAnnotation {
    public void addAnnotation(String name, Object value);
    public Object getAnnotation(String name);
    public void updateAnnotation(String name, Object newValue);
    public void removeAnnotation(String name);
}
```

Les annotations permettent d'augmenter la souplesse du modèle et de formuler des conditions plus riches au niveau des règles de reconfiguration. Elles peuvent être attachées, consultées, modifiées et supprimées à tout moment. Il est possible d'associer, statiquement, des annotations ayant des valeurs primitives (entiers, chaînes de caractères, etc.) aux éléments d'une application directement dans la base de composants. Ceci est illustré par l'exemple suivant :

```
- <version>
  <name>VideoServer_HighQ</name>
  <description>High quality video streaming</description>
  - <annotations>
    <annotation name="min_bandwidth" value="512" />
  </annotations>
  ...
</version>
```

Dans cet exemple, on indique que la version "VideoServer_HighQ" d'un composant qui sert les vidéos en mode *streaming*, nécessite une bande passante minimale de 512Kb/s pour fonctionner correctement. Cette information est exploitée si une chute de la bande passante est observée.

2.2 Organisation du méta-niveau

Cette section présente l'organisation des classes et des interfaces les plus importantes constituant le méta-niveau. La Figure 61 illustre cette organisation.

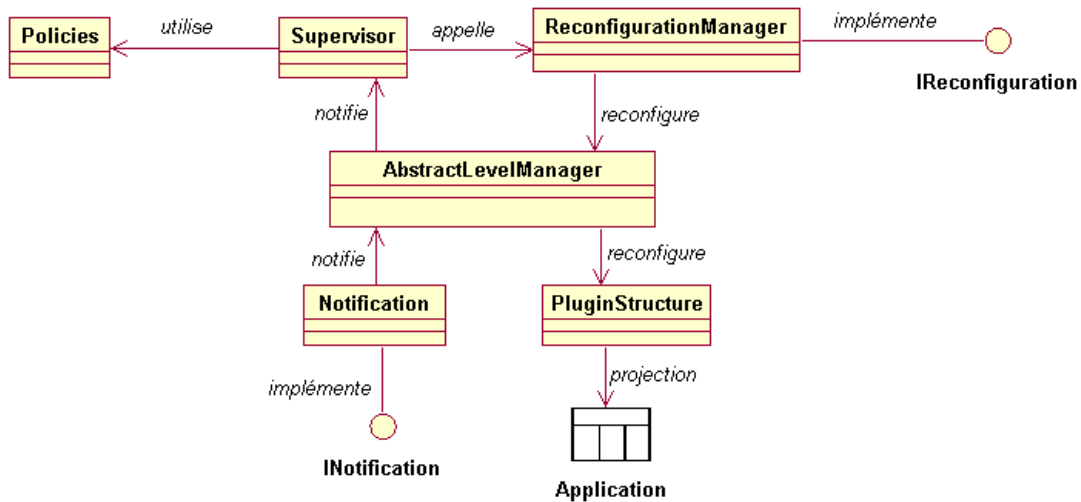


Figure 61. Organisation du méta-niveau

- Classe "Notification" : implémente les méthodes de base qui permettent d'envoyer des notifications. Ces méthodes sont définies dans l'interface "INotification" :

```

public interface INotification
{
    public void notifyComponentInstanciation(Object instance);
    public void notifyConnection(Object sourceInst, String sourcePort,
        Object targetInst, String targetPort);
    public void notifyComponentInstanceRemoval(Object instance);
    public void notifyDisconnection(Object sourceInst, String sourcePort,
        Object targetInst, String targetPort);

    public void notifyEvent(NotifEvent event);
}
    
```

- Classe "AbstractLevelManager" : permet de gérer la représentation abstraite de l'application. Elle manipule ainsi les différentes classes du modèle abstrait.

- Classes "Policies" : organise les règles de reconfiguration, chargées à partir d'un fichier. Les opérations qui permettent au superviseur de consulter les règles sont définies dans l'interface "IPolicies" :

```
public interface IPolicies {  
    public Rule[] getRule(String eventName);  
    public Rule[] getAllRules();  
}
```

Il est à noter que cette interface concerne la consultation des règles. Une autre interface implémentée par la classe "Policies" permet de remplacer et de retirer même dynamiquement les règles.

- Classe "Supervisor" : implémente le moteur qui analyse et applique les règles de reconfiguration. A la réception d'un événement nommé "E", le superviseur demande au gestionnaire de règles (instance de la classe "Policies") de lui retourner toutes les règles dont l'événement porte le nom "E". Il évalue ensuite la condition associée à chaque règle. Supposons que le superviseur reçoit un événement de nom "OVERLOAD_EVENT" et qu'il trouve la règle suivante :

```
[RULE]  
ON: OVERLOAD_EVENT  
IF: ((LOAD > 80) AND (Handler.version = 1))  
THEN: replace Handler1 NewHandler1
```

Pour évaluer la condition, le superviseur a besoin des valeurs des deux propriétés "LOAD" et "version". Il appelle la méthode `getValue("LOAD")` sur l'événement reçu pour obtenir la première valeur. Il appelle ensuite la méthode `getAnnotation("version")` sur le composant nommé "Handler" pour obtenir la deuxième valeur. Enfin il évalue la condition. La version actuelle du superviseur est très élémentaire. Elle ne supporte que les conditions simples.

- Classe "ReconfigurationManager" : implémente les différentes fonctions de reconfiguration fournies par le système. Ces fonctions opèrent sur les éléments du modèle abstrait. Leur effet est ensuite projeté sur les applications concrètes. Les fonctions de reconfiguration sont définies dans l'interface "IReconfiguration" :

```
public interface IReconfiguration
{
    public boolean disconnectPortInstance(
        ComponentInstance sourceCI, String sourcePort,
        ComponentInstance targetCI, String targetPort);

    public boolean disconnectRequiredPortInstance(
        ComponentInstance ci, String requiredPort);

    public boolean disconnectProvidedPortInstance(
        ComponentInstance ci, String providedPort);

    public boolean connectPortInstance(
        ComponentInstance sourceCI, String sourcePort,
        ComponentInstance targetCI, String targetPort);

    public boolean reconnectFromPortInstance(
        ComponentInstance oldSourceCI, String oldSourcePort,
        ComponentInstance targetCI, String targetPort,
        ComponentInstance newSourceCI, String newSourcePort);

    public boolean reconnectToPortInstance(
        ComponentInstance sourceCI, String sourcePort,
        ComponentInstance oldTargetCI, String oldTargetPort,
        ComponentInstance newTargetCI, String newTargetPort);

    public ComponentInstance createComponentInstance(Component component);

    public boolean removeComponentInstance(ComponentInstance removedCI);

    public Component[] deployComponent(String componentName);

    public boolean replaceComponentInstance(
        ComponentInstance oldCI, ComponentInstance newCI);

    public boolean passivateComponentInstance(
        ComponentInstance passivatedCI);

    public boolean activateComponentInstance(ComponentInstance activatedCI);
}
```

- Structure de plugins : les opérations de reconfiguration sont appliquées à la représentation abstraite de l'application à reconfigurer. Il est nécessaire de propager ces modifications à l'application concrète. La propagation dépend du modèle ciblé et c'est à ce niveau que la généralité du système de reconfiguration trouve ses limites. Pour répondre aux besoins spécifiques de chaque modèle ciblé, nous avons mis en

place une structure qui intègre un ensemble de plugins, spécifiques à ce modèle. Ces plugins sont responsables de la propagation effective des modifications appliquées au niveau abstrait. La liste des plugins à développer pour chaque modèle a été présentée dans la section 7 du chapitre précédent. Les points suivants illustrent les interfaces qui doivent être implémentées par chaque plugin :

- *Plugin de binding* : doit implémenter l'interface "IBinding".

```
public interface IBinding
{
    public boolean disconnectPortInstance(
        ComponentInstance sourceCI, RequiredPortInstance sourcePI,
        ComponentInstance targetCI, ProvidedPortInstance targetPI);
    public boolean connectPortInstance(
        ComponentInstance sourceCI, RequiredPortInstance sourcePI,
        ComponentInstance targetCI, ProvidedPortInstance targetPI);
}
```

- *Plugin d'instanciation* : doit implémenter l'interface "IInstanciation".

```
public interface IInstanciation
{
    public String createComponentInstance(Component component);
    public boolean removeComponentInstance(ComponentInstance removedCI);
}
```

- *Plugin de déploiement* : doit implémenter l'interface "IDeployment".

```
public interface IDeployment
{
    public Component[] deployComponent(String componentName);
}
```

- *Plugin de gestion d'état* : doit implémenter l'interface "IStateTransfer".

```
public interface IStateTransfer
{
    public State getComponentInstanceState(ComponentInstance sourceCI);
    public boolean setComponentInstanceState(
        ComponentInstance targetCI, State state);
}
```

- *Plugin de gestion de communication*: doit implémenter l'interface "ICommunication".

```
public interface ICommunication
{
    public boolean passivateComponentInstance(
        ComponentInstance passivatedCI);
    public boolean activateComponentInstance(
        ComponentInstance activatedCI);
}
```

Les opérations définies dans les interfaces présentées ci-dessus, et implémentées par les différents plugins, ont pour rôle de projeter les modifications appliquées au niveau abstrait sur l'application concrète. Le chapitre suivant illustre des cas réels de développement de plugins de projection.

3 OUTIL D'ASSISTANCE AU TRANSFERT D'ETAT

Cette section illustre l'outil que nous avons mis en œuvre pour aider le développeur à spécifier la description de l'état. La Figure 62 montre la vue graphique de cet outil.

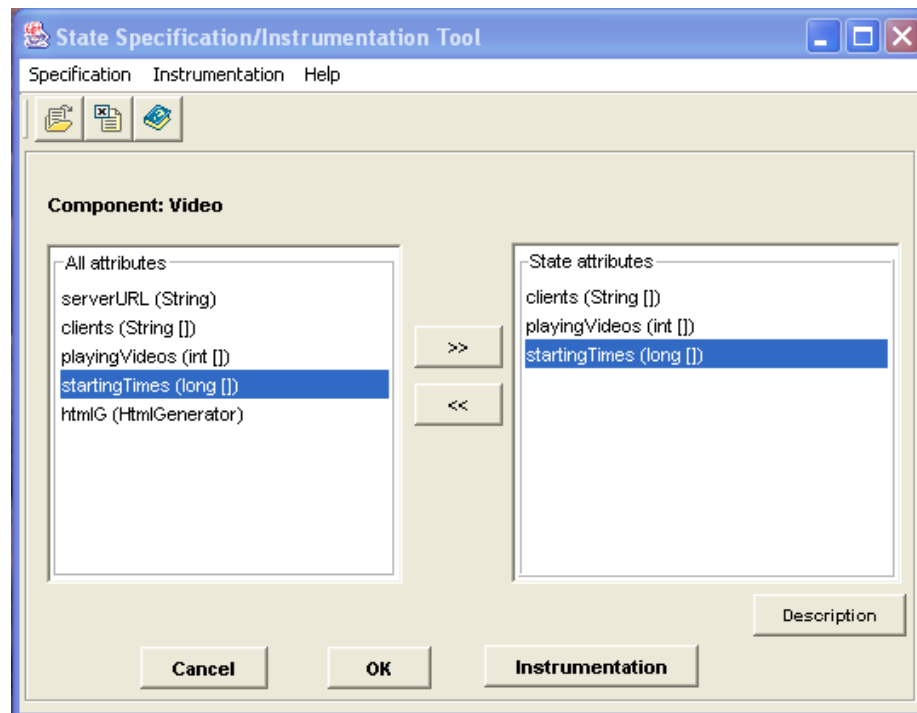


Figure 62. Outil de spécification et d'instrumentation d'état

Comme illustré par la figure précédente, l'outil affiche la liste des attributs appartenant à la classe d'implémentation du composant sélectionné. L'utilisateur doit sélectionner dans cette liste les attributs qui forment l'état. Il peut attacher à chaque attribut une description. Le bouton "Ok" permet de générer la spécification XML des attributs formant l'état. La spécification correspondant à l'exemple précédent est la suivante :

```
<state>
  <attribute name="clients" type="String []" description="clients playing videos" />
  <attribute name="playingVideos" type="int []" description="video number played by each client" />
  <attribute name="startingTimes" type="long []" description="time at which the video has been requested" />
</state>
```

En cliquant sur le bouton "Instrumentation", l'outil génère d'abord la spécification XML comme dans le cas précédent, puis procède à l'instrumentation de la classe d'implémentation. Comme nous l'avons expliqué, l'instrumentation consiste à générer les deux opérations "getState" et "setState" qui permettent respectivement, de consulter et d'initialiser l'état d'une instance. Le code suivant illustre les deux opérations générées :

```
public State getState()
{
    try {
        State state=new State();
        state.addEntry(new StateEntry(
            "clients", "String []",
            "clients playing videos", clients));
        state.addEntry(new StateEntry(
            "playingVideos", "int []",
            "video number played by each client", playingVideos));
        state.addEntry(new StateEntry(
            "startingTimes", "long []",
            "time at which the video has been requested",
            startingTimes));
        return state;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

public void setState(State state)
{
    try {
        clients = (String [])state.getEntry(0).getValue();
        playingVideos = (int [])state.getEntry(1).getValue();
        startingTimes = (long [])state.getEntry(2).getValue();
    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
}
```

Pour l'instrumentation, nous avons utilisé Javassist [Chi98, JAVASS], un outil de manipulation de bytecode Java. Ceci permet de travailler sur le code binaire sans avoir besoin du code source des composants. Le choix de Javassist a été motivé par sa facilité d'utilisation. En particulier, pour étendre un code binaire existant, l'utilisateur peut décrire le code à ajouter directement en source Java. Il n'a pas besoin, comme dans le cas d'autres outils, de spécifier le code d'extension en bytecode.

4 INTERFACE GRAPHIQUE D'ADMINISTRATION

Nous avons développé une interface graphique, pour simplifier l'administration des applications. Cette interface permet de visualiser l'architecture de l'application à administrer.

Elle permet aussi à l'administrateur d'agir sur cette application en utilisant un ensemble de menus et de boutons. La Figure 63 illustre cette interface.

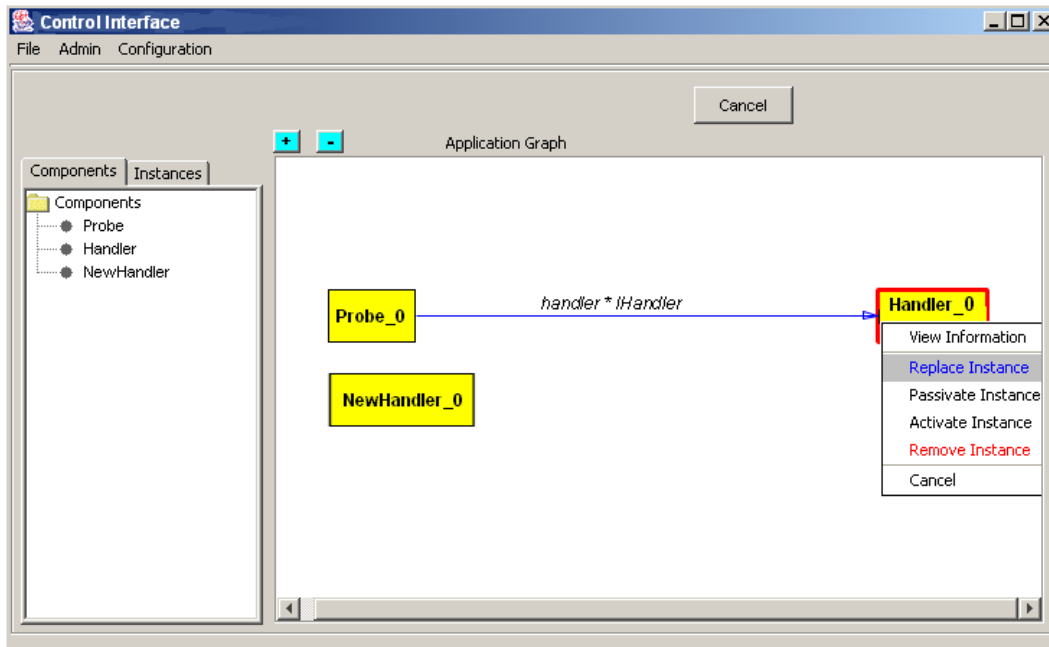


Figure 63. Interface graphique d'administration

Pour l'affichage du graphe qui représente l'architecture de l'application, nous avons utilisé un outil qui s'appelle "*Grappa*" (*A Java Graph Package*) [GRAPPA], développé au laboratoire AT&T (USA). Grappa est une partie du projet Graphviz [GRAPHVIZ]. Il permet la création et la gestion de graphes dans les applications Java. Il se charge de répartir automatiquement les nœuds d'un graphe selon des algorithmes dédiés (DOT, NEATO...); cependant, une petite modification de l'un des éléments du graphe (nœud ou arc), conduit, en général, à un repositionnement complet de tout le graphe. Ceci complique le suivi des modifications par l'utilisateur.

Les différentes fonctionnalités de l'interface graphique seront illustrées dans le chapitre suivant, à travers un exemple d'application développée en OSGi.

5 BASE DE COMPOSANTS

Nous avons expliqué dans le chapitre précédent que notre système de reconfiguration est fondé sur une base de composants. C'est dans cette base que la description des composants est stockée. Cette description est nécessaire pour pouvoir prendre en charge la

reconfiguration dynamique des applications. Les informations de la base sont organisées sous forme de fichiers XML. Les opérations implémentées par la base pour répondre aux requêtes du système de reconfiguration sont définies dans l'interface *"IComponentBase"* :

```
public interface IComponentBase {
    //Retourner un composant ayant un nom donné
    public Component getComponent(String name);
    //Retourner les composants compatibles avec un composant donné
    public Component [] getCompatible(String name, String filter);
    //Dire si deux composants sont compatibles
    public boolean isCompatible(String c1Name, String c2Name);
    //Retourner les informations de mapping de l'état entre deux composants
    public StateMapping getStateMapping(String c1Name, String c2Name);
    //Retourner les informations de mapping des opérations entre deux composants
    public OperationMapping getOperationMapping(String c1Name, String c2Name);
}
```

Il est à noter que les opérations citées permettent au système de reconfiguration de consulter les informations stockées dans la base. Une autre interface est aussi définie pour faire la mise à jour. La description stockée dans la base peut être :

- soit donnée de façon explicite,
- soit partiellement calculée à partir du code de l'application, et ensuite complétée.

Nous décrivons ci-après chacune de ces méthodes.

5.1 Description explicite

Le développeur spécifie et ajoute la description des différents composants de son application dans la base de composants. Le fichier suivant montre un exemple de base contenant la description de quelques composants.

```

<?xml version="1.0" ?>
- <ComponentBase>
  <name>DYVA Demo Repository</name>
  - <families>
    - <family>
      <name>Shopping</name>
      <description>Components for e-shopping</description>
      - <components>
        - <component>
          <name>ElecShopping</name>
          <description>Shopping of electronic components</description>
          - <requiredPorts>
            - <requiredPort>
              <name>database</name>
              <type>portal.db.IDB</type>
            </requiredPort>
          </requiredPorts>
          - <providedPorts>
            - <providedPort>
              <name>shoppingPort</name>
              <type>portal.shopping.IShopping</type>
            </providedPort>
          </providedPorts>
          - <versions>
            - <version>
              <name>ShoppingV1</name>
              <description>Simple shopping component</description>
              <implementation class="portal.shopping.ShoppingImplV1" />
            </version>
            - <version>
              <name>ShoppingV2</name>
              <description>Shopping component that shows the total price</description>
              <implementation class="portal.shopping.ShoppingImplV2" />
            </version>
            - <version>
              <name>ShoppingV3</name>
              <description>Shopping component that shows the selected products</description>
              <implementation class="portal.shopping.ShoppingImplV3" />
            </version>
          </versions>
        </component>
        ...
      </components>
    </family>
    ...
  </families>
</ComponentBase>

```

La base ci-dessus contient une famille qui s'appelle *Shopping* formée d'un ensemble de composants de commerce électronique. *ElecShopping* est l'un des composants de cette famille.

Il est utilisé dans la vente de pièces électroniques. Il a un port requis (*database*) et un port fourni (*shoppingPort*). Trois versions de ce composant existent dans la base. *ShoppingV1* permet d'afficher les produits disponibles dans une base de données, et de réaliser une commande. *ShoppingV2* permet d'afficher, en plus des produits disponibles, le prix total des produits commandés par un client. Enfin, *ShoppingV3* est une amélioration des deux premières versions. Il permet, en plus, d'afficher la liste et le nombre des produits commandés. Chaque version a une implémentation différente.

5.2 Description partiellement calculée

Dans certains cas, au lieu de fournir explicitement la description des composants, le développeur peut suivre quelques conventions de programmation. Un outil peut ensuite être utilisé pour analyser le code de l'application et extraire la description des composants. Cette approche a été utilisée dans un modèle de composants expérimental, que nous avons employé dans la phase de développement de DYVA. Ce modèle, que nous avons baptisé Java-C, est basé sur Java et se réduit simplement à un ensemble de règles lexicales, ressemblant ainsi au modèle JavaBeans. Deux règles simples ont été adoptées :

- Un composant est identifié par une classe Java, dont le nom est toujours suffixé par le mot "*Component*".
- Les ports fournis correspondent aux interfaces implémentées par cette classe.
- Les ports requis sont les attributs de la classe qui référencent d'autres composants. Leur noms doivent être suffixés par le mot "*Receptacle*".

Le code suivant montre un exemple d'application de ces règles :

```

public class HandlerComponent implements IHandler {

    public HandlerComponent() {
    }

    public void handle(int param) {
        // ...
    }
}

public class ProbeComponent {
    IHandler handlerReceptacle;

    public ProbeComponent() {
    }
}

```

Nous avons développé un outil permettant d'analyser le code binaire d'une application écrite selon ce modèle. Le résultat d'analyse du code précédent par exemple, est illustré par la Figure 64.

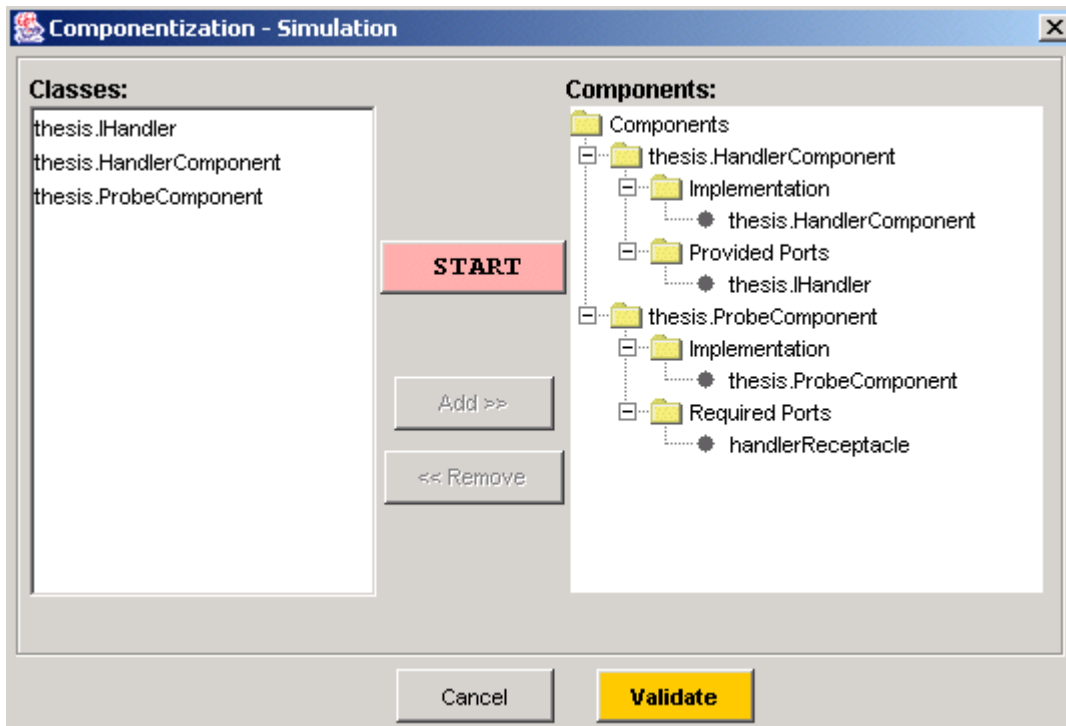


Figure 64. Analyse du code binaire d'une application

La partie gauche de l'interface montre les différentes classes de l'application, écrites selon les règles lexicales mentionnées précédemment. La vision "composant" obtenue après l'analyse des classes est affichée dans la partie droite.

Après la validation, la description obtenue est ajoutée à la base de composants. Elle peut être complétée ultérieurement à la main ou à l'aide d'un outil spécifique. Cet outil est présenté dans la sous-section suivante.

5.3 Outil d'édition et de manipulation de la base de composants

Nous avons développé un outil graphique pour simplifier la création, l'édition et la mise à jour de la base de composants. La Figure 65 donne un aperçu de cet outil.

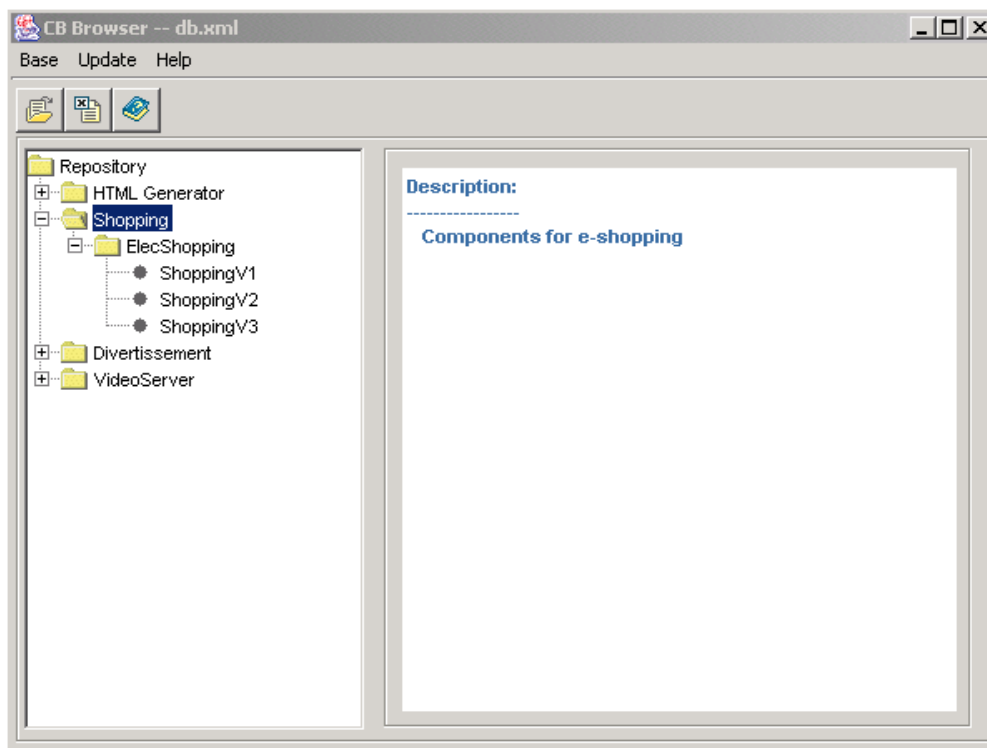


Figure 65. Outil d'édition et de manipulation

La partie gauche de l'interface de l'outil montre l'organisation hiérarchique de la base de composants. Comme nous l'avons expliqué dans le chapitre précédent, les composants sont

organisés en familles, et chacun peut avoir plusieurs versions. La partie droite de l'interface fournit la description de l'élément sélectionné de la hiérarchie. Comme le montre la Figure 66, l'outil permet aussi de modifier cette description.

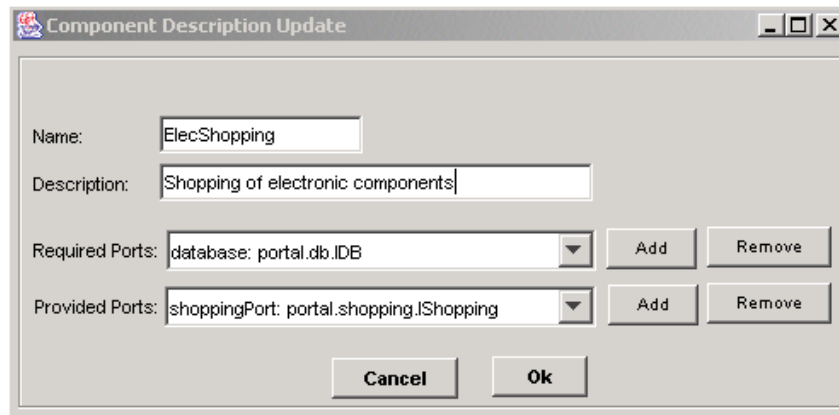


Figure 66. Mise à jour de la base de composants

6 CONCLUSION

Nous avons décrit dans ce chapitre les éléments d'implémentation de notre système de reconfiguration DYVA. Nous avons présenté les interfaces et les structures les plus importantes du système. Nous avons également illustré les différents outils que nous avons mis en œuvre pour assister l'utilisateur, et simplifier l'utilisation de notre système de reconfiguration.

Dans le cadre de cette thèse, nous n'avons pas travaillé sur les outils de surveillance de l'environnement (bande passante, mémoire, disque, etc.). Pour les besoins de nos tests, nous avons mis en œuvre de simples outils, qui permettent de simuler les paramètres de l'environnement, et qui se chargent d'envoyer des notifications au système de reconfiguration. Actuellement, nous sommes en train d'étudier la possibilité d'utiliser quelques outils de surveillance existants, et en particulier un ensemble d'outils développés dans le cadre du projet LeWYS [LEWYS]. Ce travail a été confié à un étudiant de Master [Gue04].

Le chapitre suivant explique comment nous avons personnalisé DYVA pour l'appliquer au modèle OSGi. Il présente aussi une application de portail Web, que nous avons développée en OSGi, pour illustrer les différentes fonctions de reconfiguration supportées par notre système.

CHAPITRE 7

DYVA : PERSONNALISATION

PRÉAMBULE.

La personnalisation de notre système de reconfiguration pour le modèle OSGi fait l'objet de ce chapitre. Nous illustrons, dans un premier temps, la création des interfaces de notification et de reconfiguration, spécifiques au modèle OSGi, en fonction des interfaces exposées par DYVA. Nous donnons ensuite des exemples de mise en œuvre des plug-ins de projection. Avant de conclure ce chapitre, nous présentons une application, illustrant un portail Web, que nous avons développée en OSGi, dans le but de valider la personnalisation.

1 INTRODUCTION

Nous avons expliqué, dans les chapitres précédents, que l'un des objectifs de cette thèse était de concevoir un système de reconfiguration qui favorise la modularité et la réutilisation. Pour satisfaire ces objectifs, nous avons fait en sorte que notre système soit le plus possible indépendant des modèles de composants ciblés. Ceci nous a conduit à mettre au point une partie générique (un "framework") qui matérialise la logique générale de reconfiguration. Cette partie ne peut pas être exploitée en tant que telle. Elle doit être spécialisée en fonction des caractéristiques spécifiques d'un modèle particulier, pour pouvoir supporter la reconfiguration dynamique des applications issues de ce modèle.

Ce chapitre explique comment nous avons personnalisé DYVA pour le modèle OSGi. Il discute le processus de personnalisation, et présente les interfaces, les classes et les outils que nous avons mis en œuvre pour les besoins du modèle OSGi.

La personnalisation du système de reconfiguration consiste à spécifier et implémenter une interface de notification et une interface de reconfiguration. Elle consiste aussi à développer l'ensemble des plug-ins qui permettent de projeter les modifications sur les applications OSGi, et éventuellement quelques outils.

2 INTERFACE DE NOTIFICATION D'OSGi

Nous avons défini et implémenté une interface pour permettre aux applications OSGi d'envoyer des notifications au système de reconfiguration. Cette interface est basée sur l'interface de notification de base de DYVA. Dans l'implémentation actuelle, l'interface de notification d'OSGi permet simplement de voir les opérations de notification de base, avec une terminologie différente. Cette interface est définie comme suit :

```
public interface IOSGiNotification {
    public void notifyServiceCreation(Object instance);
    public void notifyServiceConnection(
        Object sourceObject, String sourcePort,
        Object targetObject, String targetPort);
    public void notifyServiceRemoval(Object instance);
    public void notifyServiceDisconnection(
        Object sourceObject, String sourcePort,
        Object targetObject, String targetPort);
    public void notifyEvent(NotifEvent event);
}
```

Par exemple, lorsqu'un objet service est créé, la méthode "*notifyServiceCreation()*" doit être appelée avec la référence de l'objet comme argument.

Nous insistons sur le fait que, d'après la spécification OSGi, les services sont fournis par les *objets services*. Un objet service implémente une ou plusieurs *interfaces de service*, sous lesquelles, les services sont enregistrés dans le framework OSGi. Ceci est illustré par la Figure 67.

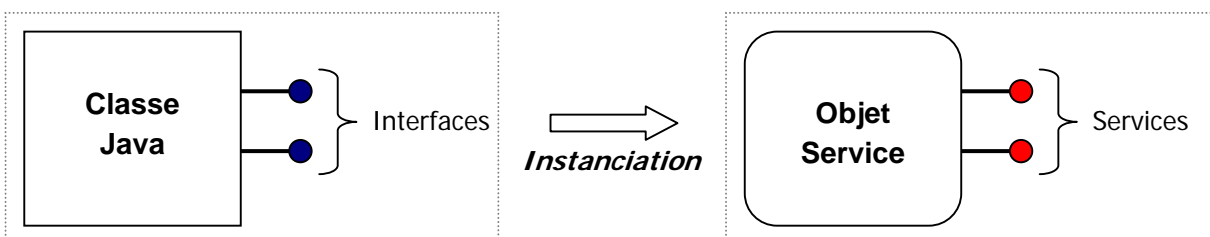


Figure 67. Services OSGi

Le codage effectif de l'appel des méthodes de notification peut être fait de deux manières différentes :

- Soit explicitement par le développeur de l'application. Par exemple, pour notifier la création d'un objet service, il suffit d'insérer, dans le constructeur

de la classe d'implémentation du composant, un appel à la méthode `notifyServiceCreation()`.

- Soit implicitement, à l'aide d'un outil d'instrumentation. D'une façon générale, le rôle de cet outil est de matérialiser la connexion entre une application et le système de reconfiguration. L'outil que nous avons implémenté pour cet effet sera présenté dans la section 5.

3 INTERFACE DE RECONFIGURATION D'OSGi

L'interface de reconfiguration d'OSGi définit les opérations de reconfiguration spécifiques aux applications OSGi. Elle est basée sur l'interface de reconfiguration de base de DYVA. Les opérations définies dans cette interface, et qui permettent à l'administrateur de reconfigurer une application OSGi, sont les suivantes :

```
public interface IOSGiReconfiguration {

    // Libérer une interface de service
    public boolean freeServiceInterface(
        String sourceSerName, String receptacle,
        String targetSerName, String interf);

    // Libérer un service
    public boolean freeService(
        String sourceSerName, String targetSerName);

    // Libérer un service (par tout le monde)
    public boolean freeService(String sourceSerName);

    // Libérer un bundle
    public boolean freeBundle(long bundleID);

    // Affecter un service
    public boolean setService(
        String sourceSer, String receptacle,
        String targetSer, String interf);

    // Reconnecter vers un nouveau service
    public boolean reconnectToService(
        String sourceSer, String receptacle,
        String oldTargetSer, String oldInterf,
        String newTargetSer, String newInterf);

    // Supprimer un service/bundle
    public boolean removeService(String removedSer);
    public boolean removeBundle(long bundleID);

    // Déployer un bundle
    public long[] deployBundle(long bundleID);

    // Créer un service
    public Object createService(String serviceType);

    // Remplacer un service/bundle
    public boolean replaceService(Object oldSer, Object newSer);
    public boolean replaceBundle(long oldBundleID, long newBundleID);

}
```

L'implémentation des opérations présentées ci-dessus, est fortement basée sur l'implémentation des opérations de reconfiguration de base.

En relation avec le concept de *bundle* qui représente l'unité de déploiement, on souhaite, par exemple, mettre au point une nouvelle opération "*freeBundle()*". Le rôle de cette opération est de déconnecter un bundle de l'application, en vue de le libérer (décharger ses ressources de la mémoire et le désinstaller). Si nous partons du principe qu'un bundle peut contenir plusieurs composants, et que chacun de ces composants peut avoir plusieurs

instances, l'opération "*freeBundle()*" peut être interprétée comme la libération de toutes les instances de tous les composants contenus dans le bundle.

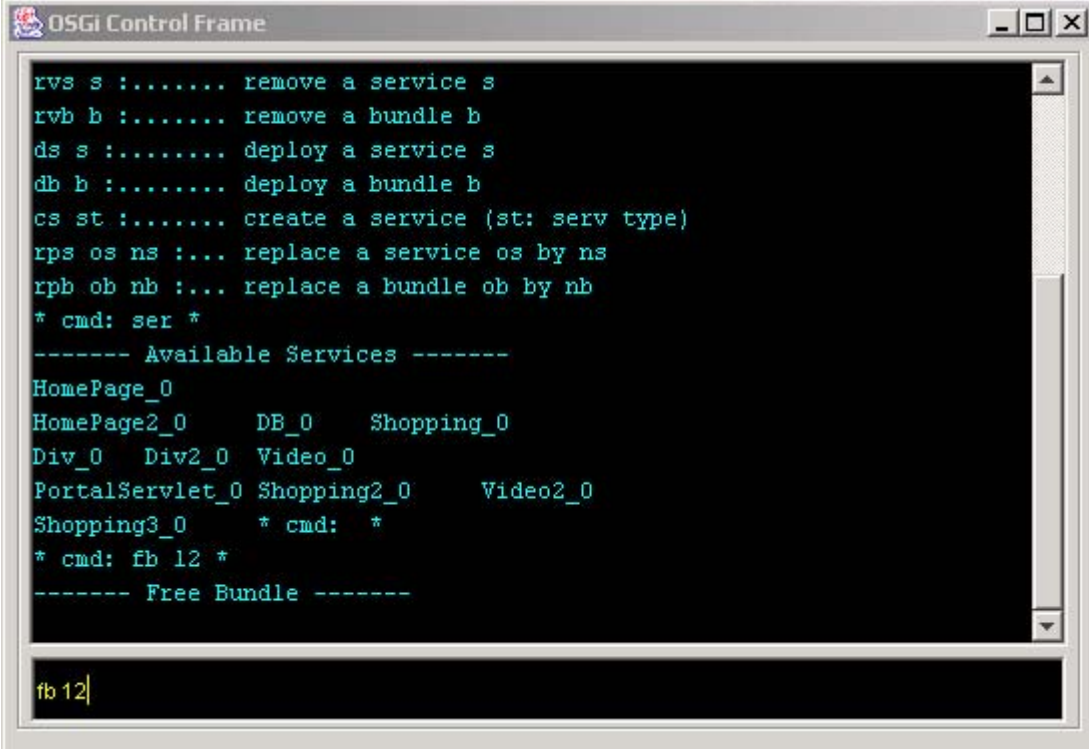
L'opération "*freeBundle()*" peut être implémentée concrètement de différentes manières. Le code suivant illustre l'une des implémentations possibles :

```
public boolean freeBundle(long bundleID)
{
    // retrouver le bundle qui correspond à l'identificateur
    Bundle bundle=bundleContext.getBundle(bundleID);
    if(bundle==null)
        return false;
    // retrouver les composants contenus dans le bundles
    String location=bundle.getLocation();
    ComponentInstance cis[]=reconfManagerRef.getAllComponentInstances();

    // libérer les services fournis par tous les composants
    for(int i=0; i<cis.length; i++) {
        if(("file:"+cis[i].getType().getLocation()).equals(location))
            freeService(cis[i].getName());
    }

    // arrêter le bundle
    try {
        bundle.stop();
        return true;
    }
    catch(Exception ex) {
        System.out.println("
            [!]: Error when stopping Bundle ["+bundleID+"] !");
        return false;
    }
}
```

Nous avons développé une interface textuelle pour permettre à l'administrateur d'interagir avec le système, et d'invoquer les opérations de reconfiguration d'OSGi. Cette interface est illustrée par la Figure 68. Elle comporte deux parties, l'une (en bas) pour saisir les commandes de reconfiguration, et l'autre (en haut) pour suivre l'exécution de ces commandes. Dans le haut de la fenêtre de la Figure 68, on peut voir une partie des commandes supportées (suppression de service, suppression de bundle...). Puis on voit le résultat d'exécution de deux commandes ("ser" pour afficher les services disponibles et "fb 12" pour libérer l'un des bundles de l'application.



```

OSGi Control Frame
----- Available Services -----
HomePage_0
HomePage2_0  DB_0  Shopping_0
Div_0  Div2_0  Video_0
PortalServlet_0  Shopping2_0  Video2_0
Shopping3_0  * cmd: *
* cmd: fb 12 *
----- Free Bundle -----
fb 12

```

Figure 68. Interface textuelle de contrôle OSGi

Il est possible de contrôler une application OSGi en utilisant l'interface graphique de DYVA. Cependant, avec cette interface, il n'est possible d'invoquer que les opérations de reconfiguration de base fournies par DYVA.

4 PLUGINS DE PROJECTION

Les plugins de projection représentent la partie la plus importante du processus de personnalisation. Ces plugins sont responsables, comme leur nom l'indique, de projeter les modifications appliquées sur la représentation abstraite de l'application vers l'application concrète. Le code suivant illustre le *plugin de binding* que nous avons mis en œuvre pour le modèle OSGi, et qui permet d'établir et de supprimer la connexion entre deux instances :

```

public class OSGiBindingPlugin implements IBinding {

    public boolean disconnectPortInstance(
        ComponentInstance sourceCI, RequiredPortInstance sourcePI,
        ComponentInstance targetCI, ProvidedPortInstance targetPI) {

        // retrouver l'objet d'implémentation
        Object sourceObject=AbstractLevel.getCIOBJECT(sourceCI);
        if(sourceObject==null)
            return false;

        // retrouver la méthode de déconnexion
        String methodName=new String("unbind"+ sourcePI.getName());
        Method unbindMethod=dyva.common.utils.Utilities.findMethod(
            methodName, sourceObject.getClass());
        if(unbindMethod==null)
            return false;

        // réaliser la déconnexion en appelant la méthode obtenue
        try {
            unbindMethod.invoke(sourceObject, new Object[]{null});
            return true;
        }
        catch(Exception ex) {
            ex.printStackTrace();
            return false;
        }
    }
}

```

```

public boolean connectPortInstance(
    ComponentInstance sourceCI, RequiredPortInstance sourcePI,
    ComponentInstance targetCI, ProvidedPortInstance targetPI) {

    // retrouver l'objet d'implémentation "source"
    Object targetObject=AbstractLevel.getCIOBJECT(targetCI);
    if(targetObject==null)
        return false;

    // retrouver l'objet d'implémentation "cible"
    Object sourceObject=AbstractLevel.getCIOBJECT(sourceCI);
    if(sourceObject==null)
        return false;

    // retrouver la méthode de connexion
    String methodName=new String("bind"+ sourcePI.getName());
    Method bindMethod=dyva.common.utils.Utilities.findMethod(
        methodName, sourceObject.getClass());
    if(bindMthod==null)
        return false;

    // réaliser la connexion en appelant la méthode obtenue
    try {
        bindMethod.invoke(sourceObject, new Object[]{targetObject});
        return true;
    }
}

```

```
        ex.printStackTrace();
        return false;
    }
}
```

Dans le cas du modèle OSGi, les composants n'implémentent pas obligatoirement des opérations de binding. En effet, chaque instance est responsable de demander explicitement les références des instances dont elle a besoin. Pour pouvoir agir sur les connexions de l'extérieur des composants, au moins deux solutions sont possibles :

- ❑ La première solution consiste à utiliser des adaptateurs à interposer entre les instances connectées. Ceci est identique à la solution que nous avons mise en œuvre dans le cadre de l'expérimentation présentée dans la section 3 du chapitre 4.
- ❑ La solution la plus simple consiste à exiger que chaque composant implémente des opérations de binding. Ces opérations peuvent être soit codées explicitement par le développeur, soit ajoutées à l'implémentation dans une phase d'instrumentation après le développement. Ceci est expliqué dans la section suivante.

Le plugin de binding que nous avons développé pour OSGi, adopte cette deuxième solution. Son implémentation est basée sur les deux considérations suivantes :

- ❑ Pour réaliser la déconnexion, la méthode "disconnectPortInstance()" appelle une méthode spécifique de l'instance source. Le nom de cette méthode est constitué du mot "unbind", suivi du nom du port source. Dans OSGi, le nom du port source correspond simplement au nom de l'attribut, qui aura comme valeur, la référence de l'instance cible.
- ❑ Pour réaliser la connexion, la méthode "connectPortInstance()" appelle une méthode spécifique de l'instance source. Le nom de cette méthode est constitué du mot "bind", suivi du nom du port source.

L'implémentation des plugins de projection dépend fortement des capacités du modèle ciblé. A titre d'exemple, le modèle Fractal prévoit des méthodes pour réaliser le binding. Ces méthodes, définies dans le *contrôleur de binding*, doivent être implémentées par les composants. Le code suivant montre une façon d'implanter la méthode de déconnexion, associée au plugin de binding, que nous avons réalisé au cours de nos tests sur le modèle Fractal :

```

public boolean disconnectPortInstance(
    ComponentInstance sourceCI, RequiredPortInstance sourcePI,
    ComponentInstance targetCI, ProvidedPortInstance targetPI)
{
    // obtenir la référence du composant Fractal "source"
    org.objectweb.fractal.api.Component sourceInst=
    (org.objectweb.fractal.api.Component)
    AbstractLevel.getCIObject(sourceCI);

    // obtenir la référence du composant Fractal "cible"
    org.objectweb.fractal.api.Component targetInst=
    (org.objectweb.fractal.api.Component)
    AbstractLevel.getCIObject(targetCI);

    if(sourceInst==null || targetInst==null) // erreur
        return false;

    // réaliser la déconnexion grâce au contrôleur de binding
    try
    {
        Fractal.getBindingController(sourceInst).unbindFc(sourcePI.getName());
        return true;
    }
    catch(Exception ex)
    {
        ex.printStackTrace();
        return false;
    }
}

```

Les points suivants donnent un aperçu des autres plugins OSGi que nous avons développés :

- ❖ *Plugin d'instanciation* : il permet de concrétiser la création et la suppression d'instances de composants OSGi. En effet, chaque bundle OSGi possède son propre chargeur de classes. Par conséquent, la simple connaissance de la classe d'implémentation d'un composant, ne suffit pas pour l'instancier. Nous avons implémenté une classe fabrique ("Factory") générique, pour résoudre le problème de création d'instances. Cette classe implémente l'interface "IGenericFactory" définie comme suit :

```

public interface IGenericFactory
{
    public Object createInstance(String className);
    public void registerGenericFactory(BundleContext ctxt);
}

```

La première méthode, définie dans l'interface, permet de créer une instance d'un composant en fonction de sa classe d'implémentation. La deuxième, quant-à-elle, permet d'enregistrer une instance de la fabrique comme un service OSGi. Une

seule instance de la fabrique (et par conséquent, un seul service enregistré) suffit pour chaque bundle. Elle est responsable de l'instanciation de n'importe quel composant contenu dans le bundle.

Pour la suppression d'instances, chaque composant doit implémenter une opération spécifique nommée "removeInstance()". Cette opération doit permettre de libérer les ressources utilisées par l'instance (fichiers, fenêtres...). Elle est appelée par le plugin d'instanciation lorsque l'instance qui la reflète est supprimée au niveau abstrait.

- ❖ *Plugin de déploiement* : il permet de déployer un composant ou un ensemble de composants contenus dans le même bundle OSGi. Nous nous sommes appuyés dans le développement de ce plugin, sur les facilités offertes par le framework OSGi lui-même, en matière de déploiement. En effet, le système de reconfiguration obtient au démarrage une référence de contexte (instance de la classe "BundleContext"). Cette référence permet d'interagir avec le framework OSGi et d'utiliser ses services. L'un de ces services est justement le déploiement dynamique de bundles (opération "installBundle()").
- ❖ *Plugin de transfert d'état* : nous avons expliqué dans les chapitres précédents, que le transfert d'état est géré par deux méthodes ("getState()" et "setState()"), qui permettent respectivement de consulter et de modifier l'état d'une instance. C'est sur ces deux méthodes, intégrées potentiellement dans la phase d'instrumentation, que le plugin de transfert d'état est basé. Ce plugin se contente simplement d'exploiter ces deux méthodes pour répondre aux requêtes du système de reconfiguration.
- ❖ *Plugin de communication* : son rôle est de gérer le cycle de vie d'une instance, en la passant de l'état actif à l'état passif ou inversement. Il est aussi responsable de stocker les requêtes envoyées à une instance en état passif, et de traiter ces requêtes après l'activation de cette instance. Le code des méthodes constituant ce plugin n'est pas encore développé à l'état actuel. Nous envisageons, par contre, d'utiliser la même solution que celle que nous avons mise en œuvre dans l'expérimentation OSGi (cf. section 3 du chapitre 4).

5 OUTIL D'INSTRUMENTATION OSGI

Nous avons à plusieurs reprises, dans les chapitres précédents, mentionné un outil utilisé pour instrumenter les applications après les avoir développées. Cette section donne un aperçu de cet outil.

Par instrumentation nous entendons la modification d'un composant ou d'une application pour répondre à des conditions spécifiques. Ces conditions permettent de manipuler les entités instrumentées d'une manière uniforme par le système de reconfiguration. L'outil

d'instrumentation que nous avons développé pour OSGi permet d'analyser une application formée de plusieurs bundles OSGi. Il s'appuie sur la description stockée dans la base de composant pour analyser les différents bundles.

Physiquement, chaque bundle est un Jar (une archive Java). L'outil décompresse le contenu des différents Jars dans un répertoire temporaire. Il réalise les modifications nécessaires sur les éléments décompressés (Figure 69), et enfin il remplace le contenu des Jars analysés, avec les éléments modifiés.

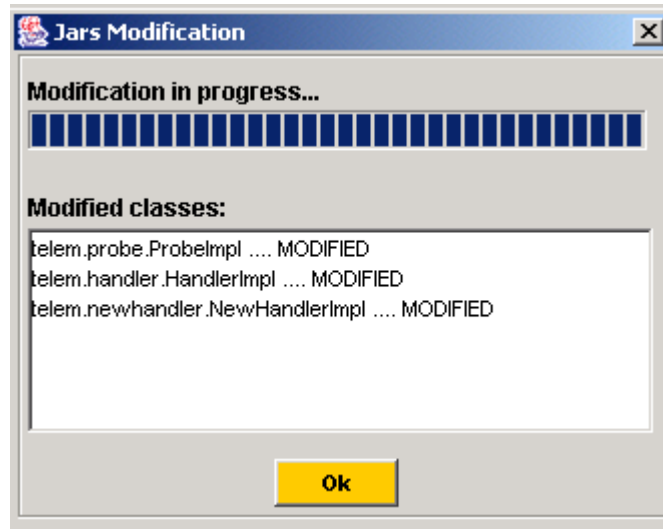


Figure 69. Instrumentation des bundles OSGi

Comme nous l'avons expliqué dans le chapitre précédent, pour la modification du code, nous avons utilisé l'outil "Javassist". Les points suivants résument les modifications introduites lors de la phase d'instrumentation :

- Pour notifier la création d'instances, une instruction de notification est ajoutée dans le constructeur de chaque classe représentant l'implémentation d'un composant. L'instruction est ajoutée à la fin du constructeur. Le code suivant montre comment ajouter l'instruction de notification en utilisant l'API de Javassist. Dans ce code, "dyvaNotifRef" représente la référence du système de reconfiguration, ce qui permet d'envoyer des notifications. Cette référence est aussi ajoutée dans la phase d'instrumentation. Ceci est omis dans le code suivant pour des raisons de simplification.

```

final CtClass implClass; // implementation class
CtConstructor constructors[]=implClass.getConstructors();
String statement=
    "{ dyvaNotifRef.notifyComponentInstanciacion(this); }";

for(int k=0; k<constructors.length; k++)
{
    // Add code to alert that an instance is created
    constructors[k].insertAfter(statement);
}

```

L'instruction de notification est ajoutée dans tous les constructeurs, comme le montre le code précédent. Il se peut alors que la création d'une instance soit notifiée plus d'une fois (plusieurs constructeurs invoqués à la création de la *même* instance). C'est pourquoi, un contrôle est effectué par le système de reconfiguration pour vérifier que la notification concerne effectivement une nouvelle instance.

- Si le développeur ne crée pas les méthodes de binding, ces dernières sont ajoutées automatiquement lors de l'instrumentation. Pour chaque port requis ayant pour nom "pr", deux méthodes "bindPr(TypeDePr target)" et "unbindPr(TypeDePr target)" sont ajoutées. Ces deux méthodes peuvent être interprétées différemment selon les deux cas suivants :
 - *Port simple* : la première méthode permet d'affecter la référence de l'instance cible (target) à l'attribut qui représente le port. La deuxième méthode, quant à elle, permet simplement d'affecter la valeur "null" à l'attribut. Ceci provoque la déconnexion.
 - *Port multiple* : dans ce cas l'attribut qui représente le port est multivalué. La méthode "bind" a pour rôle d'ajouter l'instance cible dans la liste des valeurs de l'attribut. La méthode "unbind" supprime la référence de l'instance cible de la liste des valeurs. Ceci explique la présence de l'argument "target" dans la méthode de déconnexion.
- Que les méthodes de liaison ("binding") soient codées par le développeur, ou qu'elles soient introduites par l'outil d'instrumentation, ces deux méthodes sont responsables d'envoyer des notifications de connexion et de déconnexion au système de reconfiguration. Des instructions réalisant ces notifications sont insérées à la fin de chaque méthode.
- La phase d'instrumentation comporte aussi l'intégration des méthodes de gestion de l'état (getState() et setState()). Comme nous l'avons expliqué dans le chapitre précédent, ceci relève de la responsabilité d'un autre outil, dédié à la spécification et l'instrumentation de l'état.

- Indépendamment de la modification de bytecode, la phase d'instrumentation consiste aussi à faire quelques modifications sur le Jar qui matérialise le bundle analysé. Concernant l'instanciation des composants par exemple, la classe "GenericFactory" qui est responsable de cette tâche, doit être intégrée dans chaque bundle qui comporte des composants à instancier dynamiquement. Ceci est nécessaire car la classe "GenericFactory" doit être chargée par le même chargeur que les classes d'implémentation des composants à instancier. Ceci est l'une des difficultés provoquées par le fait qu'à chaque bundle OSGi est affecté à un chargeur différent des autres.
- L'instrumentation concerne aussi les méta-données, définies dans chaque manifeste attaché à chaque bundle OSGi. Pour que les différents composants puissent interagir avec le système de reconfiguration, quelques *packages*, constituant ce dernier, doivent être importés explicitement. L'exemple suivant montre un manifeste avant et après sa transformation :

```
// --- Avant la transformation ---  
Bundle-Name: Telemetry Handler - V1  
Bundle-Activator: telem.handler.HandlerV1Activator  
Export-Package: telem.handler.service  
  
↓  
  
// --- Après la transformation ---  
Bundle-Name: Telemetry Handler - V1  
Bundle-Activator: telem.handler.HandlerV1Activator  
Export-Package: telem.handler.service  
Import-Package: dyva.osgi.reconf.service,  
                dyva.common.state
```

6 APPLICATION DE PORTAIL WEB

Pour tester la personnalisation de notre système de reconfiguration, nous avons développé une application en OSGi. Cette application représente un portail simple qui expose des services à travers le Web.

6.1 Description de l'application

L'application que nous avons développée est formée de plusieurs composants OSGi, chacun est conditionné dans un bundle, responsable de son déploiement. La Figure 70 montre un aperçu de la base de composants associée à cette application.

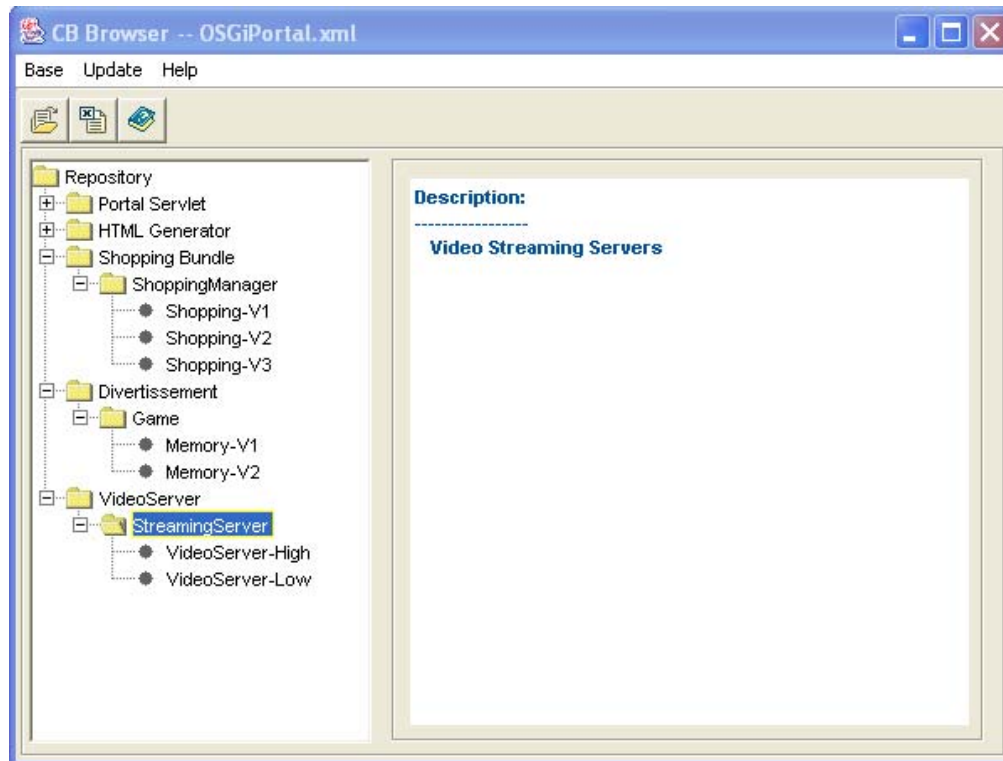


Figure 70. Base de composants de l'application portail Web

L'accès aux services fournis par le portail est assuré par le composant *PortalServlet*. Ce composant reçoit toutes les requêtes envoyées par les clients, et les redirige vers le composant approprié de l'application afin qu'elles soient traitées.

6.2 Lancement de l'application

Dans le cadre de cette thèse, nous avons réalisé nos expérimentations et nos tests, en utilisant OSCAR [OSCAR]. C'est une implémentation source libre de la spécification OSGi, développée dans l'équipe Adèle du laboratoire LSR. Le lancement de l'application se fait simplement par le lancement du framework OSGi, et par le choix du profil correspondant à l'application (Figure 71).

```
welcome to Oscar.  
=====  
Enter profile name: OSGiPortal
```

Figure 71. Lancement de l'application

La Figure 72 montre l'interface graphique de contrôle, associée à OSCAR. Les différents bundles liés à l'application sont entourés par des pointillés.

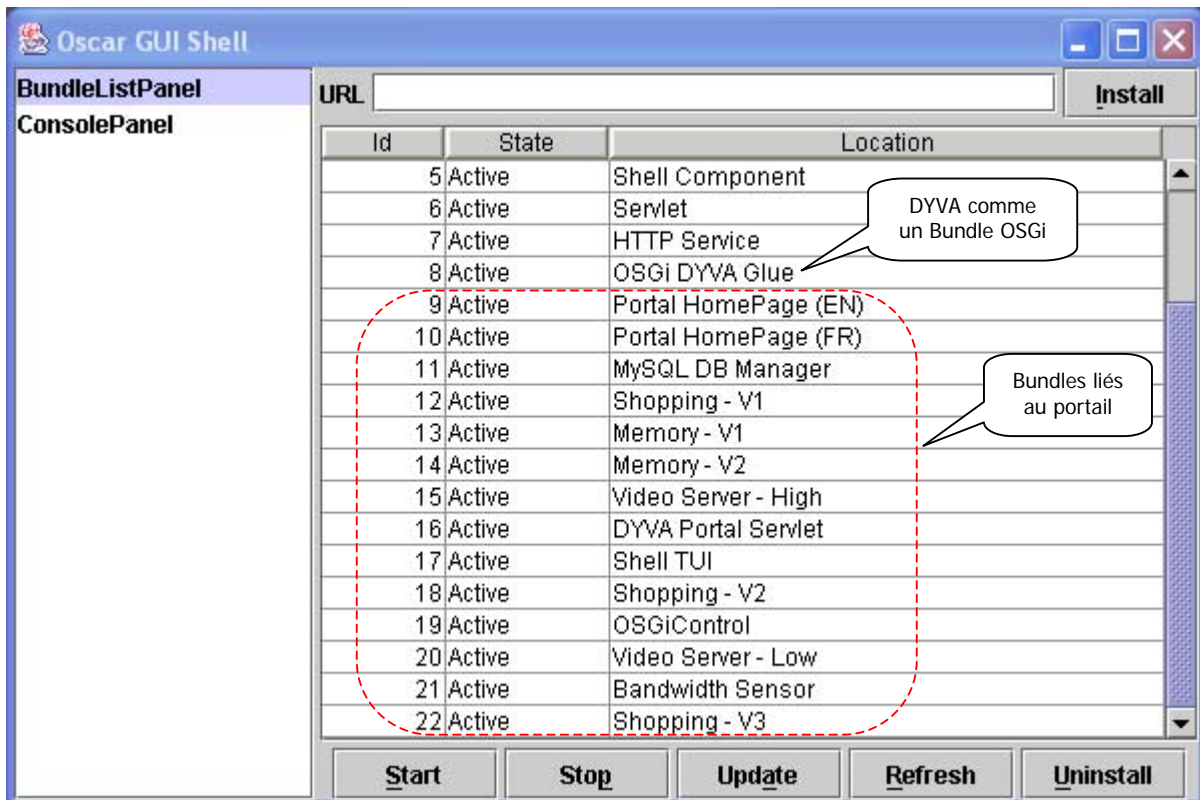


Figure 72. Interface graphique de contrôle d'OSCAR

La section suivante illustre l'interface graphique de DYVA. Cette interface simplifie la compréhension de l'architecture des applications, et permet de contrôler et de reconfigurer dynamiquement ces applications.

6.3 Utilisation et reconfiguration dynamique de l'application

Un utilisateur peut exploiter les services exposés par l'application, en utilisant simplement un navigateur Web. A la première requête interceptée (connexion), le servlet retourne la page principale de l'application. Cette page permet de sélectionner les services voulus (Figure 73).



Figure 73. Page principale du portail

L'utilisateur peut ensuite visualiser des vidéos, faire des achats ou faire des jeux. Sa requête est envoyée au servlet. Ce dernier la redirige vers le composant concerné, récupère la réponse (sous forme de page HTML), et la retourne, enfin, à l'utilisateur.

L'architecture de l'application en termes d'instances et de connexions, peut être suivie à tout moment grâce à l'interface graphique de contrôle de DYVA. La Figure 74 illustre l'architecture de l'application portail. La convention que nous avons adoptée pour nommer les instances de composants est très simple. Chaque instance porte le nom de son composant, suivi de l'ordre de création de cette instance. A titre d'exemple, la première instance du composant "MySQLDBManager" (gestionnaire de la base de données MySQL [MySQL] du portail) est nommée "MySQLDBManager_0". Si on crée une autre instance, elle sera nommée "MySQLDBManager_1" et ainsi de suite.

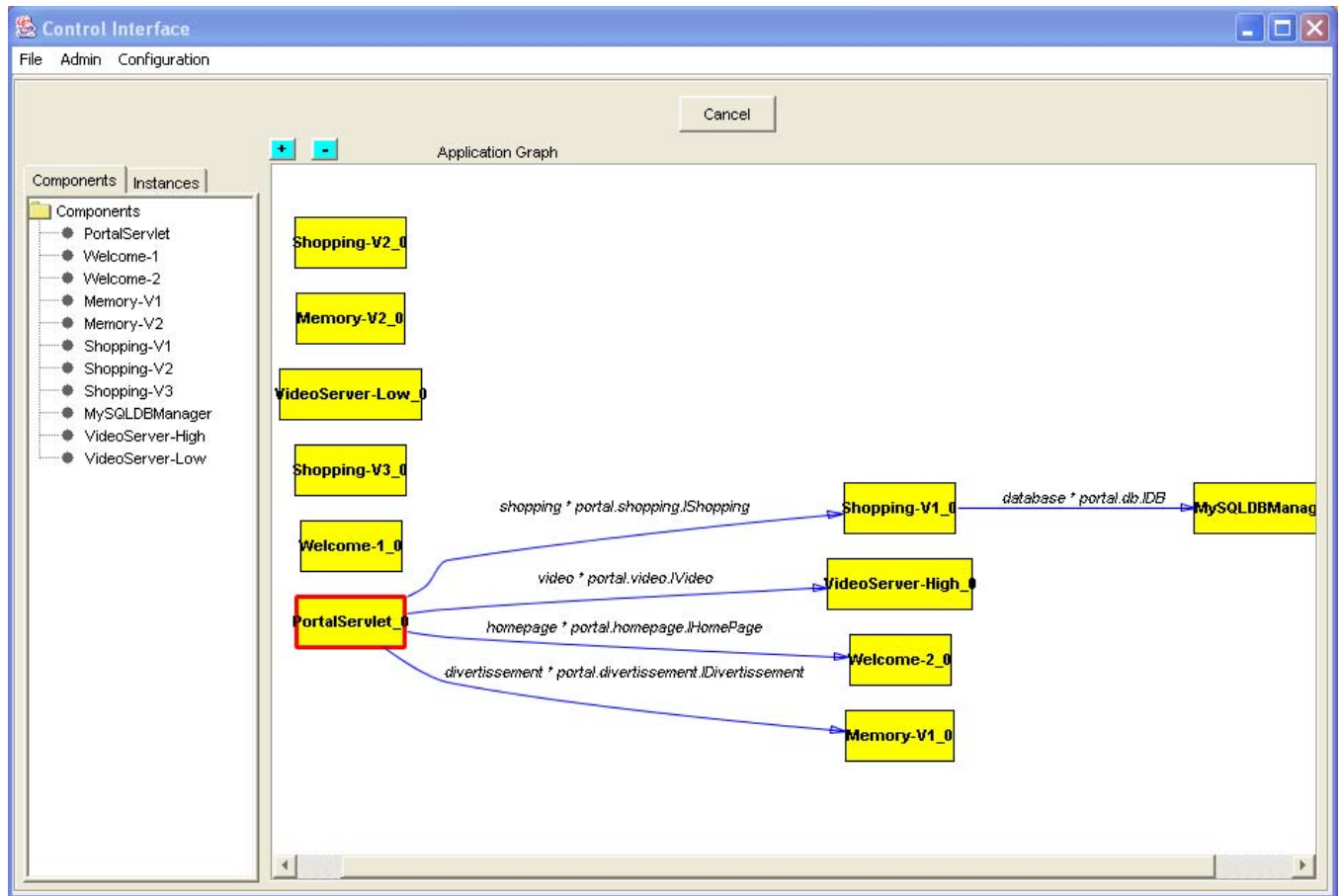


Figure 74. Architecture de l'application portail

La partie gauche de l'interface comporte deux onglets. Le premier (*Components*) présente la liste des composants, le deuxième (*Instances*) montre la liste des instances formant l'application. La partie droite de l'interface décrit l'architecture de l'application, en termes d'instances et de connexions entre elles. Chaque connexion entre deux ports de deux instances est matérialisée par un arc étiqueté. L'étiquette décrit le nom du port source et son type (le nom du port cible, dans le cadre d'OSGi, coïncide simplement avec le type de ce port).

Il est possible d'utiliser les menus pour reconfigurer l'application. Il est aussi possible d'utiliser des menus contextuels, attachés aux différents éléments de l'interface. Dans la partie droite par exemple, il est possible de créer de nouvelles instances, et il est possible d'afficher des informations d'un composant particulier (Figure 75).

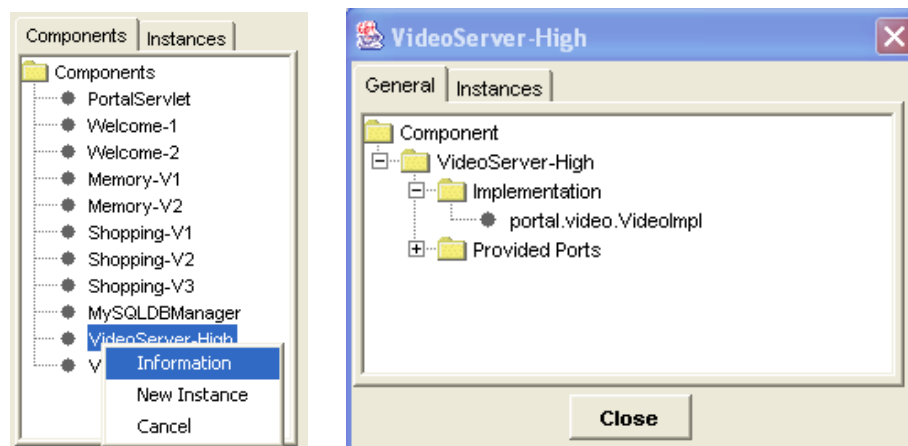


Figure 75. Affichage de la description d'un composant

Dans la suite, nous discutons l'un des scénarios de reconfiguration dynamique de l'application portail.

6.3.1 Scénario de reconfiguration

Le scénario de reconfiguration, que nous illustrons dans cette section, concerne la visualisation des vidéos exposées à travers le portail. Les vidéos qui peuvent être visualisées existent en double version : haute et basse qualité. L'application comporte deux composants pour diffuser les vidéos. Le premier diffuse des vidéos de haute qualité. Il nécessite une bonne bande passante pour fonctionner normalement (bande passante requise > 600 KB/S). Le deuxième, quant-à-lui, permet de diffuser les vidéos de basse qualité. En revanche, il n'est pas gourmand en bande passante (fonctionne avec une bande passante même inférieur à 400 KB/S).

On souhaite passer d'un composant à l'autre, en fonction de la bande passante du réseau. Ce passage doit évidemment se faire dynamiquement, car on ne peut pas s'amuser à arrêter et à redémarrer tout le portail à chaque fois que le passage d'une version à l'autre est nécessaire. La dynamique est encore plus justifiée si la bande passante change de façon fréquente.

Nous avons développé un outil qui simule les variations de la bande passante, et qui envoie périodiquement des événements au système de reconfiguration. Chaque événement décrit la dernière valeur notifiée (si elle existe), et la nouvelle valeur de la bande passante. La Figure 76 illustre cet outil de simulation.

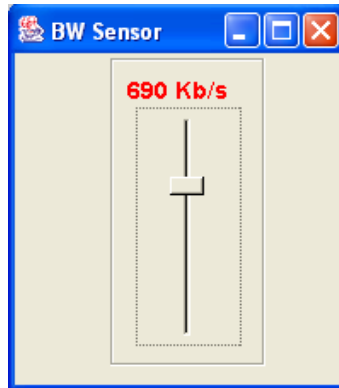


Figure 76. Simulation des variations de la bande passante

Pour automatiser le passage d'une version à l'autre, des règles de reconfiguration sont nécessaires. Les deux règles suivantes correspondent aux hypothèses que nous avons citées auparavant :

```
[RULE]
ON: BANDWIDTH_EVENT
IF: ( (NEWBW = 400) AND (NEWBW < OLDBW) )
THEN: replace VideoServer-High_0 VideoServer-Low_0

[RULE]
ON: BANDWIDTH_EVENT
IF: ( (NEWBW = 600) AND (NEWBW > OLDBW) )
THEN: replace VideoServer-Low_0 VideoServer-High_0
```

La première règle exprime que si la nouvelle valeur de la bande passante vaut 400KB/S, et que cette valeur est inférieure à l'ancienne (la bande passante est en train de chuter), il faut passer de la version haute qualité vers la version basse qualité du serveur de vidéos. La deuxième règle exprime que si la nouvelle valeur de la bande passante vaut 600KB/S, et que cette valeur est supérieure à l'ancienne (la bande passante est en train d'augmenter), il faut passer de la version basse qualité vers la version haute qualité du serveur de vidéos. Entre les deux seuils 400KB/S et 600KB/S, la configuration de l'application doit rester la même pour éviter des reconfigurations très fréquentes et probablement inutiles.

Lorsque la reconfiguration est lancée, l'une des versions est connectée à la place de l'autre. Le problème le plus délicat reste autour du transfert d'état. En effet, dans notre exemple, un simple transfert d'état n'est pas suffisant. Au moment de la reconfiguration, des clients peuvent être connectés et être en train de jouer des vidéos. Il faut que ces clients prennent en compte instantanément la reconfiguration, et se synchronisent avec la nouvelle version du serveur vidéo (Figure 77).

La difficulté réside dans le fait que la connexion entre le serveur et les clients (navigateurs) n'est pas permanente. Dès qu'un client envoie une requête HTML au serveur, et reçoit une réponse, la connexion est directement rompue. Cependant, dans notre application, il faut trouver une solution qui permet au serveur de connaître les clients connectés à un moment donné, pour pouvoir leur demander de se synchroniser en cas de reconfiguration. Plusieurs solutions sont possibles pour maintenir la connexion entre le serveur et les clients :

- Mettre au point une communication "*Serveur Push*" [HC98] : peut être réalisé en incorporant dans la page HTML, envoyée au client, la directive "*multipart/x-mixed-replace*". Cette directive permet de garder une connexion permanente avec les clients. Elle indique que tout nouveau bloc de données, envoyé au client, doit remplacer l'ancien. Dans un premier lieu, nous avons adopté cette solution; cependant, les résultats que nous avons obtenus étaient aléatoires, surtout si la page envoyée au client est complexe. Ceci vient probablement du fait que le protocole est juste expérimental (la lettre "x" dans la directive).
- Intégrer une applet Java dans la page envoyée au client : le rôle de l'applet se limite à maintenir la connexion avec le serveur. Elle se charge de vérifier périodiquement auprès du serveur s'il faut se synchroniser (si une reconfiguration a eu lieu). C'est cette deuxième solution que nous avons implantée.

Dans tous les cas de figure, le serveur est responsable de gérer l'ensemble des clients connectés. Il doit être capable de reconnaître, par exemple, la vidéo (ou les vidéos) jouée (s) par un client donné, et l'heure à laquelle cette vidéo a commencé à être jouée. Cette dernière information permet de continuer la vidéo au bon point en cas de reconfiguration (Figure 77).

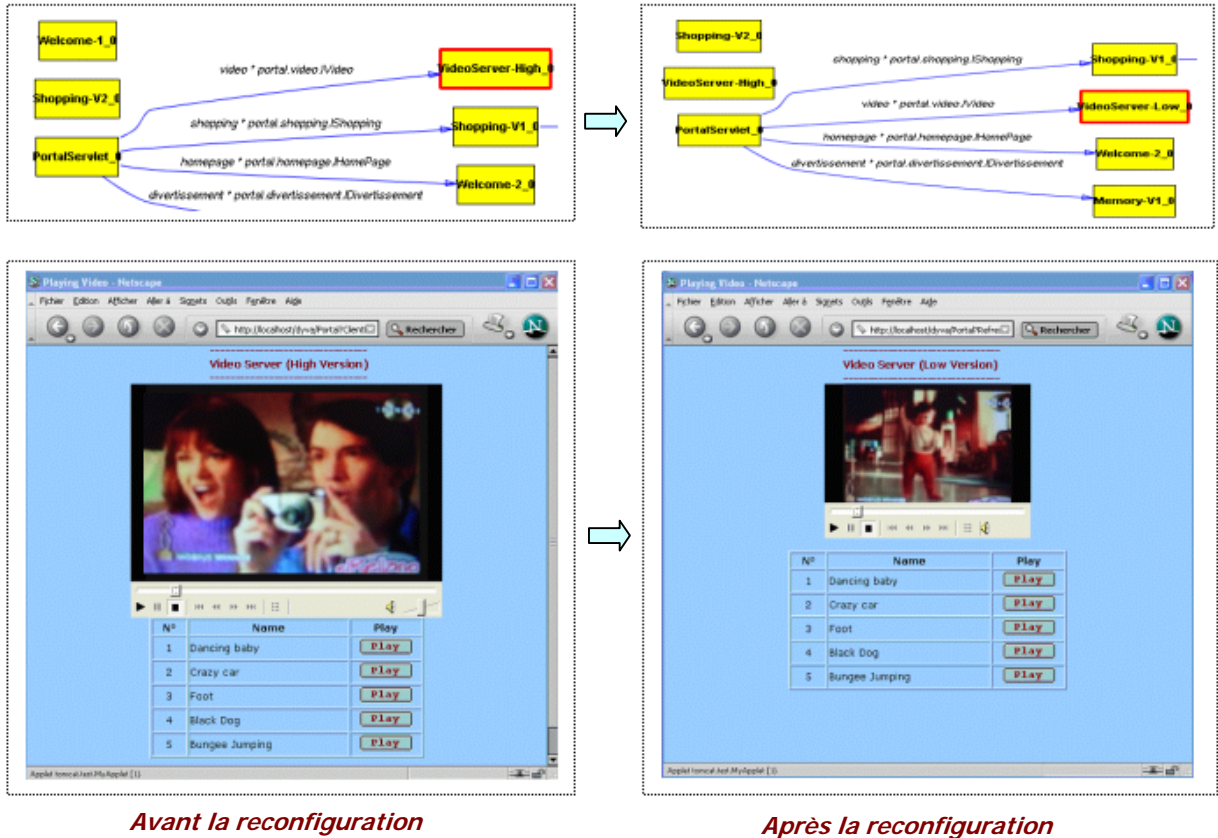


Figure 77. Remplacement du composant de diffusion vidéo

Il est également possible d'agir manuellement sur l'application pour la reconfigurer. Ceci peut être fait à l'aide de l'interface textuelle de contrôle d'OSGi, présentée par la Figure 68 (section 3). L'interface de contrôle graphique de DYVA peut être aussi utilisée pour lancer les opérations de reconfiguration. Ceci est illustré par la figure suivante :

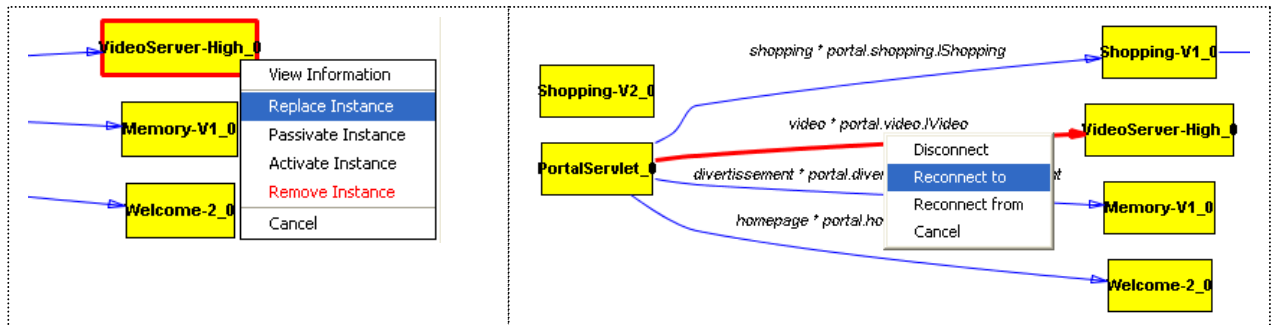


Figure 78. Reconfiguration à l'aide de l'interface graphique de DYVA

D'autres scénarios de reconfiguration peuvent également être réalisés dans notre application de portail. Ces scénarios concernent les composants qui gèrent les achats ("Shopping-Vi"), et ceux qui implantent le jeu de mémoire ("Memory-Vi").

7 CONCLUSION

Le système de reconfiguration que nous avons spécifié et développé dans cette thèse, comporte une partie générale, développée autour d'un modèle d'applications abstrait. Pour l'exploitation de ce système dans le cadre d'un modèle de composants spécifique, une phase de personnalisation est nécessaire. Cette phase permet de compléter le système de reconfiguration et de prendre en compte les spécificités du modèle en question. Nous avons montré, à travers ce chapitre, un exemple de processus de personnalisation. Cet exemple concerne le modèle OSGi. Il illustre comment créer les interfaces de notification et de reconfiguration. Il donne aussi une idée sur la mise en œuvre des plugins de projection et des outils d'instrumentation.

La dernière partie de ce chapitre a présenté une application développée pour valider la personnalisation que nous avons réalisée. Nous avons également montré, à travers un scénario, le fonctionnement de l'auto-reconfiguration et mis en évidence quelques difficultés techniques, rencontrées durant la réalisation.

CHAPITRE 8

CONCLUSION ET PERSPECTIVES

8 CONCLUSION

8.1 Problématique de reconfiguration dynamique

Les différents travaux autour de la reconfiguration dynamique, étudiés dans cette thèse, montrent la complexité et la délicatesse de cette problématique. Les expérimentations et la solution de reconfiguration que nous avons réalisées confirment également ces constatations. La complexité est liée à l'absence de mécanismes de haut niveau, permettant d'intervenir dynamiquement sur une application. La délicatesse est liée au fait qu'on agisse sur l'application quand elle s'exécute, et change donc d'état constamment. Une simple anomalie dans une opération de reconfiguration peut par conséquent conduire l'application à s'arrêter, et ainsi remettre en cause la raison d'être de la dynamicité.

La reconfiguration dynamique est un sujet qui a été abordé depuis quelques années par une grande communauté de chercheurs. Les travaux liés à ce sujet relèvent de domaines différents, d'intérêts différents et de technologies différentes. Les questions clés, souvent abordées par ces travaux, peuvent être résumées en quatre points :

- ❑ Identifier les éléments de l'application sur lesquels les opérations de reconfiguration doivent opérer (composant, objet, procédure,...).
- ❑ Exprimer les conditions que doit satisfaire l'application pour pouvoir être reconfigurée dynamiquement (composant instrumenté sous un certain format, classe non instanciée, procédure non active,...).
- ❑ Réaliser les mécanismes nécessaires à la reconfiguration (instrumentation du code, utilisation des techniques de l'AOP [DES02], utilisation des proxys, modification de l'infrastructure d'exécution,...).
- ❑ Définir les interfaces et les mécanismes nécessaires pour superviser le déroulement des opérations de reconfiguration, et intervenir en cas de problème.

Cette liste n'est pas exhaustive. Plusieurs autres questions liées à la problématique de reconfiguration dynamique restent ouvertes et sujettes à des travaux de recherche.

8.2 Principales contributions et résultats obtenus

Le travail que nous avons réalisé dans cette thèse se présente principalement en deux parties : conception et réalisation de solutions de reconfiguration pour des modèles de composants spécifiques, puis l'extension des résultats obtenus pour définir une solution plus générale.

8.2.1 Solutions spécifiques

Au lieu de définir un nouveau modèle de composants et de le doter de capacités de reconfiguration dynamique, nous sommes plutôt partis de modèles existants.

- Nous avons dans un premier temps étudié le modèle JavaBeans, et nous avons analysé la BeanBox qui est un outil de composition graphique d'applications en JavaBeans. Nous avons proposé et implémenté ensuite une extension de la BeanBox pour supporter la reconfiguration dynamique des applications composées. Les différentes opérations de reconfiguration mises en œuvre concernent principalement l'architecture des applications. Parmi ces opérations, nous pouvons citer la connexion, la déconnexion, la reconnexion et le remplacement dynamique des composants. Toute opération de reconfiguration doit nécessairement garantir la cohérence de l'application reconfigurée. Dans la solution que nous avons développée, nous avons travaillé en particulier sur la gestion de la communication entre les composants lors de la reconfiguration, pour éviter de perdre des messages ou des données échangés. Nous avons également développé des outils pour aider au transfert de l'état lors du remplacement des composants. Les mécanismes nécessaires pour maîtriser la cohérence des applications reconfigurées sont primordiales dans un système de reconfiguration. La mise en œuvre de ces mécanismes représente une partie considérable de l'effort de développement d'un tel système.
- Nous avons ensuite réalisé une solution de reconfiguration similaire pour le modèle OSGi. Ce deuxième travail sur un autre modèle nous a aidés à approfondir et à valider les idées acquises lors de la première expérimentation sur le modèle JavaBeans. La solution développée se présente sous la forme d'un composant OSGi spécifique qui incarne le système de reconfiguration. Elle est donc indépendante de toute implémentation du modèle OSGi, ce qui n'était pas le cas dans la première expérimentation qui est liée à l'environnement BeanBox.

- Le troisième travail que nous avons mené dans cette thèse traite le problème d'incompatibilité des interfaces. En effet, les composants formant une même application peuvent être réalisés indépendamment par des développeurs différents. Des composants équivalents peuvent ainsi avoir des interfaces différentes, ce qui complique leur substituabilité. Nous avons réalisé dans le contexte du modèle OSGi une solution pour résoudre le problème d'incompatibilité d'interfaces. Cette solution permet, en particulier, de remplacer un composant par un autre composant sémantiquement équivalent, même s'il a une interface différente.

8.2.2 Solution générale

Notre objectif principal dans cette thèse était la spécification et la réalisation d'une solution de reconfiguration générale. Un tel objectif constitue un vrai défi du fait que chaque modèle de composants possède ses propres concepts et sa propre plate-forme d'exécution. Cette différence entre les modèles nous a amenés à définir une solution de reconfiguration partielle mais extensible pour pouvoir considérer les spécificités de chaque modèle. Le système de reconfiguration que nous avons mis en œuvre est fondé sur un modèle de composants abstrait. Il est cependant important de souligner que nous n'avons pas introduit le modèle abstrait dans l'esprit de concevoir un nouveau modèle de composants, et nous ne prétendons pas définir un modèle de composants générique pour remplacer les autres modèles. L'intérêt du modèle abstrait est d'augmenter l'indépendance (idéalement totale) de notre système de reconfiguration vis-à-vis des modèles de composants ciblés. La solution que nous avons développée présente plusieurs avantages qui peuvent être résumés dans les points suivants :

- La logique de reconfiguration qui est en général commune à différents modèles de composants est incarnée au sein d'un même système réutilisable. Ceci évite de réimplanter la même chose à chaque fois qu'on souhaite développer un système de reconfiguration pour un nouveau modèle de composants.
- Le modèle abstrait permet d'ajouter de la sémantique aux composants et aux connexions qui les relient grâce aux annotations qui peuvent y être attachées. Ces annotations permettent aussi d'augmenter les capacités d'auto-reconfiguration de notre système comme expliqué dans le point suivant.
- Reconfigurer dynamiquement une application nécessite un cycle d'observation, d'analyse et de décision. Ces différentes étapes constituent une lourde tâche surtout si la reconfiguration doit être réalisée d'une façon répétitive. Pour assister l'utilisateur de notre système, nous avons doté ce dernier d'un moteur de raisonnement qui permet de reconfigurer automatiquement les applications. Ce moteur reçoit des notifications, et

prend des décisions en fonction d'un ensemble de règles spécifiées à l'avance. Ces règles matérialisent la logique de reconfiguration

- La prise en compte de la responsabilité de reconfiguration par un système dédié, permet au développeur de se concentrer sur la logique métier de ses applications. Cette séparation facilite la maintenance et l'évolution aussi bien des applications que du système de reconfiguration.

8.2.3 Synthèse générale

Lors du travail que nous avons réalisé sur les deux modèles JavaBeans et OSGi, nous avons rencontré plusieurs difficultés. Ces difficultés étaient en général techniques et résolubles avec plus ou moins d'efforts de développement. Les problèmes auxquels nous avons dû faire face concernent le transfert d'état, la gestion des communications et la question de cohérence au moment de la reconfiguration. Ces différents points seront abordés plus en détails dans la section présentant les perspectives.

Nous avons eu un vrai défi dans cette thèse lors de la conception et de la réalisation de notre système général. Les différents modèles de composants sont naturellement basés sur des concepts différents et possèdent des plate-formes d'exécution différentes. Il n'existe pas un modèle pivot autour duquel on aurait pu définir un système générique. Il était même difficile d'identifier une notion commune de ce qu'est un composant ou de ce qu'est une application à base de composants dans le sens général et indépendamment d'une technologie particulière. Ces différences vont à l'encontre d'une solution de reconfiguration générique. Nous sommes arrivés ainsi à la conclusion qu'il faut concevoir une solution en deux parties, l'une générique et indépendante des modèles ciblés, et l'autre complémentaire et spécifique à un modèle particulier. La question clé à ce niveau, et qui nous a demandé un effort considérable, était de décider qu'est-ce qui est commun, applicable à tous les modèles et ainsi intégrable dans la partie générique, et qu'est-ce qui est spécifique à chaque modèle.

Au vu de nos objectifs, nous avons identifié une architecture de référence d'un système de reconfiguration. Nous avons également instancié cette architecture pour le modèle OSGi. L'architecture conceptuelle et le noyau de reconfiguration ont été complètement repris. Nous nous sommes donc concentrés, dans l'instanciation, uniquement sur les spécificités du modèle OSGi (interfaces spécifiques, plugins de projection,...). Le travail d'instanciation continue et a besoin d'être consolidé sur d'autres modèles (Fractal, EJB,...) pour en tirer une évaluation complète.

9 PERSPECTIVES DE LA THESE

Le travail que nous avons réalisé dans cette thèse a permis, d'un côté, d'explorer quelques approches de reconfiguration dynamique, et d'un autre côté, de réaliser quelques expérimentations et de développer une solution de reconfiguration. Même si cette solution a été réalisée dans un esprit de réutilisabilité, et qu'elle favorise l'automatisation, beaucoup de

travail reste à faire et le chemin vers un système de reconfiguration complet et satisfaisant demeure très long. Plusieurs facettes constituent les perspectives de notre travail de thèse :

- Formalisation du problème de transfert d'état,
- Cohérence de la reconfiguration,
- Auto-reconfiguration,
- Approche par raffinement.

9.1 Formalisation du problème de transfert d'état

Le transfert d'état est l'une des problématiques les plus difficiles auxquelles nous avons dû faire face lors du développement de notre système de reconfiguration. Il est considéré comme une condition fondamentale, pour assurer, en partie, la cohérence des applications reconfigurées. Les entités en exécution (objets, instances de composants, etc.) ont un état qui évolue depuis la mise en service de l'application. Cet état correspond à une certaine sémantique englobée par les entités de l'application (solde d'un compte, historique, valeur d'un paramètre, etc.). Si en remplaçant une entité, son état n'est pas transféré vers l'entité remplaçante, la sémantique représentée par l'application peut être perdue. Dans le cadre de cette thèse, nous avons développé plusieurs outils pour assister l'utilisateur dans la tâche de spécification de l'état, et dans la génération des opérations effectuant son transfert. La notion d'état mérite d'être plus approfondie et mieux formalisée. Cette formalisation est indispensable pour pouvoir traiter l'état d'une façon uniforme et permettre plus d'automatisation dans son transfert.

Dans le travail que nous avons réalisé, nous n'avons traité que le transfert de l'état faible, ce qui correspond aux valeurs des structures de données appartenant aux entités remplacées (variables, tableaux, etc). Dans certains cas, ceci ne suffit pas, et il est nécessaire de prendre en considération même le flot d'exécution (état fort). En effet, lorsque le remplacement implique du code, les processus en exécution peuvent être affectés. Dans ce cas, il faut garantir que les processus affectés par le changement, reprennent leur exécution, dans le nouveau code, à partir d'un point cohérent. Ce type de transfert d'état est plus difficile à traiter. Dans le cas général, il requiert, par exemple, la présence de points de correspondance entre l'ancien et le nouveau code. Des travaux dans ce sens ont été faits sur le langage Java dans [BH02, BHD03]. Il est important d'étudier comment fusionner et exploiter ces travaux pour compléter et étendre notre travail de thèse.

9.2 Cohérence de la reconfiguration

Nous pouvons qualifier une reconfiguration de cohérente si elle garantit au moins le passage de l'application en reconfiguration d'un état stable vers un autre. Dans le même sens, nous pouvons qualifier un état de stable si l'application est capable de continuer normalement son exécution à partir de celui-ci, et de fournir des résultats corrects. Dans cette

thèse, nous nous sommes intéressés à la mécanique de reconfiguration et nous avons concentré nos efforts pour définir une solution réutilisable. Nous avons effleuré la problématique de cohérence à travers notre travail sur le transfert d'état et la gestion de la communication pendant la reconfiguration.

Il est très intéressant d'étudier d'une façon approfondie le problème de cohérence, de le formaliser et de développer des mécanismes pour le résoudre. Plusieurs questions méritent d'être analysées à ce stade :

- Comment savoir si une reconfiguration s'est bien déroulée ?
- Comment agir en cas de problème, et quels sont les mécanismes nécessaires pour remettre l'application dans un état stable ?
- Comment identifier les états stables d'une application ?
- Est-il possible d'automatiser la définition et l'identification des états stables ?
- Est-il possible d'automatiser le diagnostic d'une application reconfigurée et les actions qu'il faut entreprendre en cas de problème ?

Il est important de souligner que la cohérence est une condition nécessaire pour la validité de tout système de reconfiguration, et que la dynamique de la reconfiguration perd son sens sans la garantie de cette condition. Ainsi, la problématique de cohérence est aussi importante que la problématique autour des mécanismes de reconfiguration, même si dans les approches que nous avons étudiées, la première est largement négligée au profit de la deuxième.

9.3 Auto-reconfiguration

Nous avons essayé dans notre approche de favoriser l'automatisation de la reconfiguration. Nous avons développé plusieurs outils dans ce sens, et nous avons bâti une architecture basée sur des règles, incarnant la logique de reconfiguration, et destinées à être interprétées par un moteur de raisonnement. L'architecture que nous avons conçue et le moteur que nous avons mis en œuvre sont très élémentaires. Ils ont été définis pour tester les concepts. L'une des perspectives de cette thèse consiste à repenser et à étendre ce que nous avons réalisé. Pour cela il est nécessaire de se pencher sur d'autres domaines pour exploiter les compétences déjà acquises autour des techniques de formalisation et de raisonnement, et notamment dans le domaine de l'intelligence artificielle, des systèmes multi-agents et des bases de données actives.

9.4 Approche par raffinement

En regardant de plus près l'approche que nous avons développée, nous pouvons constater que l'idée de base était de définir un modèle de composants "canonique", autour duquel opère un système de reconfiguration. Pour reconfigurer une application, on crée d'abord son image

selon le modèle canonique, et le système de reconfiguration agit sur cette image avant de projeter le résultat sur l'application concrète.

Une autre approche mérite d'être étudiée. Elle consiste à couvrir le cycle de vie d'une application depuis la phase de développement, et à assister l'utilisateur dans les différentes étapes de ce cycle. Dans la programmation orientée composants, l'idée est de créer d'abord les composants, et ensuite de les assembler pour former une application. L'approche que nous proposons d'étudier emprunte une voie complémentaire. Elle est inspirée de l'approche MDA [Poo01, Bez01] proposée par l'OMG [OMG].

Le développeur doit d'abord dessiner son application d'une manière abstraite. Il peut à ce stade décider des propriétés de reconfiguration à affecter aux composants formant son application (Quel composant doit être adaptable? Quelle connexion doit être obligatoire? Quel composant dépend de quel autre? Quel composant possède un état? Quel groupe de composants supporte la mobilité?...). L'application abstraite peut être ensuite soumise à une série de raffinements, au cours desquels les composants peuvent être concrétisés dans le respect des propriétés initialement spécifiées. Cette approche permet surtout de spécifier les applications indépendamment d'un modèle de composants particulier, de spécifier leurs propriétés de reconfiguration et enfin, de générer les applications concrètes dans un modèle particulier, en utilisant des outils dédiés.

9.5 Perspectives à court terme

Le travail que nous avons réalisé dans cette thèse a pris de l'ampleur et est devenu une thématique importante dans notre équipe. Cette thématique est centrée autour des applications nécessitant l'adaptation dynamique. Notre travail sera exploité dans plusieurs projets de l'équipe comme le projet PISE [PISE]. Ces projets traitent principalement les applications à base de capteurs comme les applications domotiques et les systèmes de contrôle industriel.

A court terme, nous comptons travailler sur le problème de mobilité que nous n'avons pas traité dans cette thèse. Un tel travail sera exploité dans le projet autour du modèle "Edge-Computing" [SM02] dans lequel notre équipe est engagée avec la société Bull [BULL]. Les sociétés d'hébergement de contenus ou de services, par exemple, utilisent une architecture dans laquelle les requêtes sont prises en charge par des serveurs applicatifs. Le modèle "Edge-Computing" définit une nouvelle architecture pour permettre de rapprocher les ressources de calcul des utilisateurs finaux. Cette architecture permet d'optimiser l'utilisation des ressources réseau et de décharger les serveurs applicatifs. Concrètement, le projet est centré autour de JOnAS [JONAS], un serveur EJB développé par Bull. Pour répartir la charge sur différents serveurs (serveurs applicatifs et serveurs frontaux), on a besoin de faire la migration des données et des applications. La migration en tant que telle est fortement liée au problème de transfert d'état, avec comme cible une machine distante. Elle constitue donc le premier axe de notre futur travail à court terme.

Le projet PISE, d'un autre côté, vise le développement d'une infrastructure logicielle pour passerelles Internet sécurisées, capables d'accueillir des services d'une façon dynamique, et d'un outil permettant d'administrer à distance ces passerelles. L'infrastructure à développer sera compatible avec la norme OSGi et reposera sur un modèle de composants. L'un des verrous technologiques majeurs liés au projet PISE, et sur lequel nous pensons pouvoir contribuer, concerne la manipulation des services non-fonctionnels (sécurité, journalisation,...). Pour définir un environnement flexible et adaptable, il faut pouvoir attacher et détacher ces services aux composants métiers, en fonction des besoins, et d'une façon dynamique. L'une des techniques qui permet de manipuler les services non-fonctionnels à l'exécution, et qui a été expérimentée dans [Jar01] sur le serveur JOnAS [JONAS], consiste à gérer l'adaptation au niveau du conteneur. Dans le cadre du projet PISE, l'idée est de considérer les services non-fonctionnels au même titre que les services métiers (Figure 79). A première vue, attacher/détacher un service non-fonctionnel, par exemple, peut être traité comme la connexion/déconnexion entre services métiers. Des tests sont nécessaires pour évaluer cette vision et examiner de plus près les différences entre la manipulation dynamique des services métiers et des services non-fonctionnels.

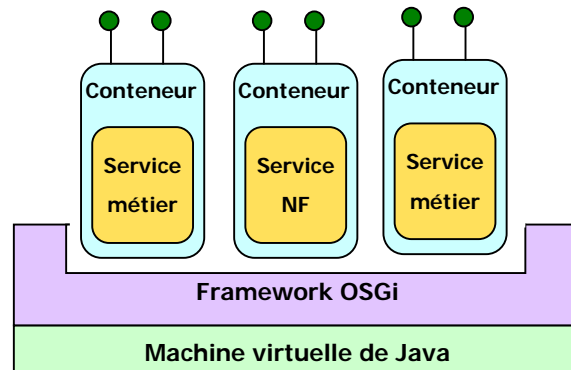


Figure 79. Architecture de l'infrastructure du projet PISE

Le troisième point sur lequel nous comptons travailler à court terme, consiste à optimiser le système DYVA et sa spécialisation pour le modèle OSGi. L'objectif est de le proposer comme une infrastructure de reconfiguration "source-libre" dans la communauté ObjectWeb, dans laquelle nous participons activement. Ceci nous donnera un retour d'expérience et nous permettra de faire évoluer notre travail de recherche autour de la problématique de reconfiguration.

BIBLIOGRAPHIE

- [ACARS] Aircraft Communications Addressing and Reporting System (ACARS)
<http://www.arinc.com/products/globalink/>
- [AJ02] S. Ambler, T. Jewel. *"EJB fondamentale"*. Eyrolles, May 2002.
- [Ala95] M. Alabau. *"Le langage ADA, Théorie et pratique"*. Masson, 1995.
- [AM00] G. D. Abowd , E. D. Mynatt.. A *"Charting past, present, and future research in ubiquitous computing"*CM Transactions on Computer-Human Interaction (TOCHI), v.7 n.1, p.29-58, March 2000.
- [AO+99] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *"The Jini Specification"*. Addison Wesley, 1999.
<http://www.jini.org>
- [ARINC] ARINC, Inc. Site Web
<http://www.arinc.com/>
- [ATB01] M. Aksit, B. Tekinerdogan and L. Bergmans. *"The Six Concerns for Separation of Concerns"*. ECOOP 2001 Workshop on Advanced Separation of concerns, Budapest, 2001.
http://trese.cs.utwente.nl/publications/papers/ecoop2001_asocws.pdf
- [BBOX] Sun Microsystems. The BeanBox.
<http://java.sun.com/products/javabeans/software/beanbox/>
- [BBUI] Sun Microsystems. The bean builder.
<http://java.sun.com/products/javabeans/beanbuilder/>
- [BC01-a] G. Bieber, J. Carpenter. *"Openwings - A Service-Oriented Component Architecture for Self-Forming, Self-Healing, Network-Centric Systems (Rev 2.0)"*, 2001.
<http://www.openwings.org>
- [BC01-b] G. Bieber, J. Carpenter. *"Introduction to Service-Oriented programming (Rev 2.1)"*, 2001.
<http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf>
- [BEAN97] Sun Microsystems. JavaBeans 1.01 specification, 1997.
<http://java.sun.com/products/javabeans/>
- [Ber96] P. A. Bernstein. *"Middleware: A Model for Distributed System Services"*. Communications of the ACM, Vol. 39 No. 2, pp.86-98, February 1996.

- [Bez01] J. Bézivin "From Object Composition to Model Transformation with the MDA". TOOLS USA'2001, Volume IEEE TOOLS-39, Santa Barbara, USA, August 2001.
<http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda>
- [BH02] S. Bouchenak, and D. Hagimont. "Services de Mobilité et de Persistance des Applications Java". Book Chapter No 6 of Les intergiciels - développements récents dans CORBA, Java RMI et les agents mobiles, Edited by Hermès, 2002.
- [BHD03] S. Bouchenak, D. Hagimont, N. De Palma. "Efficient Java Thread Serialization". 2nd ACM International Conference on Principles and Practice of programming in Java (ACM PPPJ'03), Kilkenny, Ireland, June 2003.
- [Bou00] N. Bouraqadi. "Java et la réflexion". Technical Report 2000-4-INFO, École des Mines de Nantes, janvier 2000.
- [BR00] E. Bruneton et M. Riveill. "Javapod : une plate-forme composants adaptable et extensible". Rapport de recherche 3850 - INRIA, janvier 2000.
- [BULL] Bull
<http://www.bull.com>
- [CASTOR] ExoLab Group. The Castor Project.
<http://castor.exolab.org/>
- [CCL00] W. Cazzola, S. Chiba, and T. Ledoux. "Reflection and metalevel architectures : State of the art and future trends". Lecture Notes in Computer Science - ECOOP, 2000.
- [Cer04] H. Cervantes. "Vers un modèle à composants orienté services pour supporter la disponibilité dynamique". Thèse de Doctorat en Informatique, Université Joseph Fourier, Grenoble, mars 2004.
- [CF+90] J. D. Case, M. Fedor, M. L. Schostall, and C. Davin. "A Simple network management protocol (SNMP)". RFC 1157, IETF, May 1990.
<http://www.ietf.org/rfc/rfc1157>
- [Cha96] D. Chappell. "Understanding Activex and Ole". Microsoft Press ; 1st edition, January 1996.
- [Chi00] S. Chiba, "Load-Time Structural Reflection in Java", in Proceedings of ECOOP'2000, 2000, Springer Verlag LNCS 1850.
- [Chi98] S. Chiba. "Javassist --- A Reflection-based Programming Wizard for Java". Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java October, 1998.
- [COM95] Microsoft. Component object model (COM) specification.
<http://www.microsoft.com/com/resources/comdocs.asp>
- [COO88] R. P. Cook. "StarMod - A Language for Distributed Programming". In Concurrent Programming, Addison-Wesley, edited by N. Gehani and A.D. McGettrick, pp. 93-111, 1988.
- [CS+99] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. "Architectural reflection : Concepts, design, and evaluation". Technical report, Technical Report RIDS1 234-99, DSI, Università degli studi di Milano, May 1999.

- [DC01] J. Dowling, V. Cahill. "*Dynamic Software Evolution and The K-Component Model*". Technical report, Trinity College Dublin, TCD-CS-2001-51. December 2001. Presented in the Workshop on Software Evolution, OOPSLA 2001.
- [DDW03] N. B. Dale; N. Dale; C. Weems. "*Introduction to Pascal and Structured Design*". Editions: Paperback, February 2003.
- [Dep01] N. De Palma. "*Services d'Administration d'Applications Réparties*". Thèse de Doctorat en Informatique, Université Joseph Fourier, Grenoble, 2001.
- [DES02] F. Duclos, J. Estublier R. Sanlaville. "*Une Machine à Objets Extensibles pour la Séparation des Préoccupations*". Journées Systèmes à Composants Adaptables et Extensibles, Grenoble, octobre 2002.
- [DPC] Sun Microsystems. "*Dynamic Proxy Classes*".
<http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html>
- [EAB02] T. Elrad, O. Aldawud, A. Bader. "*Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design*". Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE, Pittsburgh, USA, October 2002.
- [EJB] Sun Microsystems. "*Enterprise JavaBeans Technology*".
<http://java.sun.com/products/ejb/>
- [Eri97] J. Eriksson. "*Real-Time and Active Databases: A Survey*". Proceedings of the 2nd International Workshop on Active, Real-Time, and Temporal Database Systems, volume 1553 of Lecture Notes in Computer Science, pages 1-23. Springer, 1997.
- [Fab76] R. S. Fabry. "*How to design a system in which modules can be changed on the fly*". Proceedings of the 2nd international conference on Software engineering, p.470-476, San Francisco, California, United States, October 1976.
- [Fer89] J. Ferber. "*Computational reflection in class based object-oriented languages*". ACM SIGPLAN Notices, 24(10):317-326, October 1989.
- [FH+93] C. Falkenberg, C. Hofmeister, C. Chen, E. White, J. Atlee, P. Hagger, and J. Purtilo. "*The Polylith Interconnection System: Programming Manual for the Network Bus*". University of Maryland, College Park, 3.0 edition, September 1993.
- [FRACTAL] The Fractal Project
<http://fractal.objectweb.org/>
- [Gar00] D. Garlan. "*Software Architecture: a Roadmap*". The Future of Software Engineering, ACM Press, pp.91-101, 2000.
<http://www-2.cs.cmu.edu/~able/publications/roadmap2000/>
- [Gen97] W. M. Gentleman. "*Effective Use of COTS (Commercial-Off-the-Shelf) Software Components in long Lived Systems*". Proceedings of International Conference on Software Engineering (ICSE'97), Boston, USA, May 1997.
- [GH+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "*Design Patterns : Elements of Reusable Object-Oriented Software*". Addison-Wesley, 1995.

- [Gow97] B. Gowing, "A Reflective Programming Model and Language for Dynamically Modifying Compiled Software", PhD Thesis, Dept. of Computer Science, Trinity College Dublin, 1997.
- [GRAPHVIZ] Graphviz - open source graph drawing software
<http://www.research.att.com/sw/tools/graphviz/>
- [GRAPPA] Grappa - A Java Graph Package
<http://www.research.att.com/~john/Grappa/>
- [Gue04] K. Guellil. "Auto-Reconfiguration Dynamique des Systèmes Logiciels à Composants : Application au système DYVA". Rapport de Master 2 Recherche, Université Joseph Fourier, Grenoble, septembre 2004.
- [HC01] G. T. Heineman, W. T. Council. "Component-based software engineering : putting the pieces together". Addison-Wesley Professional, August 2001.
- [HC98] J. Hunter, W. Crawford. "Java Servlets". Edition française, page 190, O'Reilly & Associates, 1998.
- [HG98] G. Hjálmtýsson, R. Gray. "Dynamic C++ classes - A Lightweight mechanism to update code in a running program". In proceedings of the USENIX Annual Technical Conference, pp. 65-76, June 1998.
- [Hic01] M. Hicks. "Dynamic Software Updating". PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.
- [IDL3] The Object Management Group. "OMG IDL Syntax and Semantics - CORBA 3.0"
<http://www.omg.org/cgi-bin/apps/doc?formal/02-06-07.pdf>
- [J2EE] Sun Microsystems. "Java 2 Platform, Enterprise Edition (J2EE)".
<http://java.sun.com/j2ee/>
- [Jar01] Z. Jarir. "Adaptabilité dynamique des services dans JOnAS". Projet de DEA à l'Ecole des Mines de Nantes, 2001.
<http://www.emn.fr/x-info/ledoux/Publis/rapport-Zahi.pdf>
- [JAVA] Sun Microsystems. "Java Technology".
<http://java.sun.com>
- [JAVASS] Site Web de Javassist
<http://www.csg.is.titech.ac.jp/~chiba/javassist/>
- [JES] Sun Microsystems. "Java Embedded Server".
<http://www.sun.com/software/embeddedserver/>
- [JMX] Sun Microsystems, "Java Management Extensions (JMX)".
<http://java.sun.com/products/JavaManagement/>
- [JONAS] Le projet JOnAS
<http://jonas.objectweb.org/>

- [KB03] A. Ketfi and N. Belkhatir. "*Dynamic Interface Adaptability in Service Oriented Software*". 8th International Workshop on Component-Oriented Programming (WCOP'03), Darmstadt, Germany, July 2003.
- [KB04] A. Ketfi and N. Belkhatir. "*A metamodel-based approach for the dynamic reconfiguration of component-based software*". The Eighth International Conference on Software Reuse, Madrid, Spain, July 2004.
- [KBC02-a] A. Ketfi, N. Belkhatir and P.Y. Cunin. "*Dynamic updating of component-based applications*". International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas, Nevada, USA, June 2002.
- [KBC02-b] A. Ketfi, N. Belkhatir and P.Y. Cunin. "*Adapting Applications on the Fly*". Doctoral Symposium - 17th IEEE International Conference Automated Software Engineering, Edinburgh, UK, September 2002.
- [KC+02] A. Ketfi, H. Cervantes, R. S. Hall and D. Donsez. "*Composants Adaptables au dessus d'OSGi*". Journées Systèmes à Composants Adaptables et Extensibles, Grenoble, Octobre 2002.
- [Ker00] Kernighan. "*Le langage C, norme ANSI*", Dunod, janvier 2000.
- [Kir97] M. Kirtland. "*The COM+ programming model makes it easy to write components in any language*". Microsoft Systems Journal, December 1997.
- [KM90] J. Magee, J. Kramer. "*The evolving philosophers problem : Dynamic change management*". IEEE Transactions on Software Engineering, pp. 1293–1306, November 1990.
- [Kor02] M. Korkea-aho. "*A Survey of Context-Aware Computing*". Internal Technical Report, Department of Computer Science, Helsinki University of Technology, March 2002.
<http://www.hut.fi/~mkorkeaa/doc/context-aware.html>
- [Kru98] P. Kruchten. "*Modeling Component Systems with the Unified Modeling Language*". Rational Software Corp. - Proceedings of the International Workshop on Component-Based Software Engineering, 1998.
<http://www.sei.cmu.edu/cbs/icse98/papers/p1.html>
- [Les03] V. Lestideau. "*Modèles et environnement pour configurer et déployer des systèmes logiciels*". Thèse de Doctorat en Informatique, LISTIC/LSR, Université de Savoie, décembre 2003.
- [LEWYS] LeWYS Project
<http://forge.objectweb.org/projects/lewys/>
- [Lit01] R. C. Litiu. "*Providing Flexibility in Distributed Applications Using a Mobile Component Framework*". PhD thesis, University of Michigan, Ann Arbor, MI, September 2001.
- [Lit01] R. C. Litiu. "*Providing Flexibility in Distributed Applications Using a Mobile Component Framework*". Ph.D. dissertation, University of Michigan, Electrical Engineering and Computer Science, Septembre 2000.
<http://www.eecs.umich.edu/~aprakash/dacia/publications.html>
- [LK03] S. Lakare , A. Kaufman. "*OpenVL: The Open Volume Library*". Proceedings of the Eurographics/IEEE TVCG Workshop on Volume graphics, Tokyo, Japan, July 07-08, 2003.

- [Mae87] P. Maes. "*Concepts and experiments in computational reflection*". Conference proceedings on Object-oriented programming systems, languages and applications, p.147-155, Orlando, Florida, United States, October 1987.
- [Mar01] C. A. Marcos. "*Design Patterns as First-Class Entities*". PhD thesis, Universidad Nacional del Centro de la Provincia de Buenos Aires, mars 2001.
<http://www.exa.unicen.edu.ar>
- [MC99] Microsoft Corporation. "*Microsoft Mastering : Microsoft Visual Basic 6.0 Fundamentals*". Microsoft Press, August 1999.
- [MDC92] J. Malenfant, C. Dony, and P. Cointe. "*Behavioral reflection in a prototype-based language*". In A. Yonezawa and B. Smith, editors, Proceedings of Int'l Workshop on Reflection and Meta-Level Architectures, pp. 143–153, Tokyo, RISE and IPA(Japan), November 1992.
- [MJD96] J. Malenfant, M. Jaques, and F.N. Demers. "*A tutorial on behavioral reflection and its implementation*". Proceedings of the Reflection 96 Conference, Gregor Kiczales editor, pp. 1-20, San Francisco, California, USA, April 1996.
- [MODULA] Page Web du langage Modula
<http://www.modula2.org/>
- [MP+00] S. Malabarba , R. Pandey , J. Gragg , E. Barr , J. F. Barnes. "*Runtime Support for Type-Safe Dynamic Java Classes*". Proceedings of the 14th European Conference on Object-Oriented Programming, p.337-361, June 12-16, 2000.
- [MySQL] MySQL AB. "*MySQL: open source database*"
<http://www.mysql.com/>
- [NETB] NetBeans. Ide.
<http://www.netbeans.org/>
- [NS98] B. S. Northcote, D. E. Smith. "*Service control point overload rules to protect intelligent network services*". IEEE/ACM, Transactions on Networking (TON) 6(1): p. 71-81, February 1998.
- [OGB98] A. Oliva, I. C. Gracia, and L. E. Buzato. "*The reflective architecture of guaraná*". Technical Report IC-98-14, Institute of Computing, State University of Campinas, Campinas,São Paulo, Brazil, September 1998.
- [OGC00] Z. Qian , A. Goldberg , A. Coglio. "*A formal specification of Java class loading*". Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 325-336, Minneapolis, Minnesota, USA, , October 2000.
- [OM+98] P. Oreizy, N. Medvidovic, R. Taylor, D. Rosenblum. "*Software Architecture and Component Technologies: Bridging the Gap*", Workshop on Compositional Software Architectures, Monterey, January 1998.
- [OMG] The Object Management Group (OMG)
<http://www.omg.org/>
- [OSCAR] OSCAR - An OSGi framework implementation
<http://oscar.objectweb.org/>

- [OSGI99] Open Services Gateway initiative (OSGi)
<http://www.osgi.org/>
- [Pat00] A Patzer. "*Programmation Java côté serveur - Servlets, JSP et EJB*". Eyrolles, janvier 2000.
- [PBJ97] F. Plasil, D. Balek, R. Janecek. "*DCUP: Dynamic Component Updating in Java/CORBA Environment*". Tech. Report No. 97/10, Dep. of SW Engineering, Charles University, Prague, 1997.
- [PBJ98] F. Plasil, D. Balek, R. Janecek. "*SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*". Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998.
- [PISE] Page Web du projet PISE : Passerelle Internet Sécurisée et flexible
<http://www.telecom.gouv.fr/rnrt/projets/PISE.htm>
- [Poo01] J. Poole. "*Model-Driven Architecture: Vision, Standards and Emerging Technologies*". ECOOP 2001, Workshop on Meta-modeling and Adaptive Object Models, April 2001.
- [PRS04] R. Pawlak - J.-P. Retaillé - L. Seinturier. "*Programmation orientée aspect pour Java/J2EE*". Eyrolles, Mai 2004.
- [PSW91] J. M. Purtilo, R. Snodgrass, A. Wolf. "*Software bus organization: Reference model and comparison of existing systems*". MIFWG Tech. Rep. 8, Computer Science Dept., University of Arizona, Tucson, 1991.
- [Pur94] J. M. Purtilo. "*The POLYLITH Software Bus*". ACM TOPLAS, vol. 16(N.1), pp. 151-174. January 1994.
- [RC00] B. Redmond, V. Cahill. "*Iguana/J: Towards a Dynamic and Efficient Reflective Architecture for Java*". Workshop on Reflection and Meta-Level Architectures at 14th European Conference on Object-Oriented Programming, Cannes, France, June 2000.
- [RC02] B. Redmond, V. Cahill. "*Supporting Unanticipated Dynamic Adaptation of Application Behaviour*". Proceedings of 16th European Conference on Object-Oriented Programming, Malaga, Spain, June 2002.
- [Rey83] R.F. Rey, Ed. "*Engineering and Operations in the Bell System*". 2nd Edition, AT&T Bell Laboratories, Murray Hill, N.J. 1983.
- [Rod00] J. Rodley. "*Programmer des applets Java*". Eska Interactive - Sybex, octobre 2000.
- [SAW94] B. N. Schilit, N. Adams, and R. Want. "*Context-Aware Computing Applications*". IEEE Workshop on Mobile Computing, December 1994.
- [SB96] G. Sabah, X. Briffault. "*Smalltalk : Programmation orientée objet et développement d'applications*". Eyrolles, mai 1996.
- [SF93] M. E. Segal , O. Frieder. "*On-the-Fly Program Modification: Systems for Dynamic Updating*". IEEE Software, v.10 n.2, pp.53-65, March 1993.
- [SG96] M. Shaw, D. Garlan. "*Software Architecture. Perspectives on an Emerging Discipline*". Prentice-Hall, 1996.

- [SM02] Sun Microsystems. "*Computing at the Edge*". White Paper, 2002.
http://www.sun.com/aboutsun/media/presskits/entrysystems2002/Edge_Final.pdf
- [SMF] IBM. "*Service Management Framework (SMF)*".
<http://www-306.ibm.com/software/wireless/smf/>
- [Smi82] B.C. Smith. "*Reflection and semantics in a procedural language*". Technical report, Technical Report 272, MIT Laboratory for Computer Science, 1982.
- [Smi84] B.C. Smith. "*Reflection and semantics in lisp*". Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages, pages 23-35, January 1984.
- [Smi95] D. E. Smith. "*Ensuring robust call throughput and fairness for SCP overload controls*". IEEE/ACM, Transactions on Networking (TON), v.3 n.5, pp.538-548, October 1995.
- [SOFA] SOFA (SOFTware Appliances) Project
<http://sofa.objectweb.org/>
- [Som03] N. Le Sommer. "*Contractualisation des ressources pour les composants logiciels : une approche réflexive*". Thèse de Doctorat en Informatique, Université de Bretagne Sud, décembre 2003.
- [Str97] B. Stroustrup. "*The C++ Programming Language*". Addison-Wesley, 3rd edition, 1997.
- [Szy02] C. Szyperski. "*Component Software: Beyond Object-Oriented Programming*". Second edition, ACM Press, Component Software Series, Addison-Wesley, 2002.
- [Tan00] Z. Tang. "*Dynamic reconfiguration of component-based applications in java*". Master's thesis, Massachusetts Institute Of Technology, 2000.
- [TD+00] P. L. Tarr , M. D'Hondt , L. Bergmans , C. V. Lopes. "*Workshop on Aspects and Dimensions of Concern: Requirements on, and Challenge Problems for Advanced Separation of Concerns*". Proceedings of the Workshops, Panels, and Posters on Object-Oriented Technology, pp.203-240, June 12-16, 2000.
- [Tha99] T.L. Thai. "*Learning DCOM*". Ed. A. Oram, Sebastopol, CA 95472 O'Reilly & Associates, 1999.
- [Tra01] G. Travis. "*Understanding the Java ClassLoader*". Tutorial, developerWorks, April 2001.
- [UML97] The Object Management Group. "*The Unified Modeling Language (UML) version 1.0*". January 1997.
<http://www.uml.org/>
- [Vil03] J. Villalobos. "*Fédération de Composants : une Architecture Logicielle pour la Composition par Coordination*". Thèse de Doctorat en Informatique, Université Joseph Fourier, Grenoble, Juillet 2003.
- [Voa98] J. M. Voas. "*The Challenges of Using COTS Software in Component-Based Development*". IEEE Computer, Vol. 31, No. 6, pp. 44-45, June 1998.
- [WO97] H. Wong, S. Oaks. "*Java Threads*". O'Reilly 1ère édition, janvier 1997.

- [XML00] World Wide Web Consortium (W3C). "*Extensible Markup Language (XML) 1.0, second edition*", octobre 2000.
<http://www.w3.org/TR/REC-xml>
- [XMLS01] World Wide Web Consortium (W3C). "*XML Schema*".
<http://www.w3.org/XML/Schema>