Composants adaptables au dessus d'OSGi

Abdelmadjid KETFI, Humberto CERVANTES, Richard HALL, Didier DONSEZ,

Laboratoire LSR, Institut Imag, Universite Joseph Fourier
Bat. C, 220 rue de la Chimie, Domaine Universitaire
BP 53, 38041 Grenoble Cedex 9, France
{Abdelmadjid.Ketfi, Humberto.Cervantes, Didier.Donsez,Rick.Hall}@imag.fr

***RÉSUMÉ**. Le travail présenté dans ce papier s'articule autour d'OSGi, une spécification ouverte destinée à héberger des services et à gérer leur cycle de vie. OSGi constitue un bon point de départ pour développer des applications flexibles et facilement gérables. Cependant, il ne décrit pas l'architecture des applications ce qui rend difficile d'avoir une vision globale d'une application constituée de plusieurs composants. Nous décrivons dans cet article une plate-forme construite au dessus d'OSGi qui permet de définir des composants, de décrire l'architecture des applications, de déployer ces applications et de les administrer en cours d'exécution. Egalement, pour répondre aux besoins des applications critiques, à haute disponibilité, nous proposons une machine d'adaptation, encapsulée sous forme d'un composant OSGi. Cette machine implémente les concepts de base liés à l'adaptation dynamique et permet entre autres de modifier les connexions et de remplacer à temps réel un composant par un autre.*

ABSTRACT. This paper is centered around a OSGi, an open specification for a component model that allows services to be deployed and mananged. OSGi is a good starting point to develop flexible applications that are easily manageable. However, the specification doesn't describe anything about the architecture of applications, something that complicates the task of having a global vision of an application built out of components. This paper describes a platform built on top of OSGi that allows components to be defined, the architecture of an application to be described and whole applications to be deployed and managed during runtime. Moreover, to solve the high availability needs of critical applications, an adaptation machine, which itself is an OSGi bundle, is proposed. This machine implements the basic concepts that are tied to dynamic adaptation and allows among other things to modify the connections or the replacement of a component during runtime.

MOTS-CLÉS : *OSGi, Composant, Adaptation Dynamique.*
KEYWORDS: *OSGi, Component, Dynamic Adaptation.*

1. Introduction

Les modèles à base de composants [Riv00] assurent un développement modulaire et une maintenance plus aisée des applications. Parmi les principaux modèles, on peut citer JavaBeans [Sun97], EJB[Sun01][Rom99], CCM [OMG99], .NET [Tha01]. Un composant est un module logiciel décrit par des contrats proposés, des contrats requis, des propriétés configurables et par des contraintes techniques. Les contrats sont définis par des interfaces et des événements. Les applications sont obtenues par assemblage (ou tissage) de composants configurés. L'assemblage consiste à connecter les composants entre eux par les contrats requis et fournis. L'exécution de l'application est démarrée lorsque ses composants sont déployés sur la plate-forme à composants (qui peut être distribuée sur plusieurs hôtes). Chaque composant est pris en charge par un conteneur dont le rôle consiste à isoler le composant de l'environnement et de prendre en charge les contraintes techniques liées aux aspects non fonctionnels (distribution, persistance, fiabilité, sécurité, …).

Traditionnellement, le cycle de vie de l'application suit les phases strictement successives de développement, assemblage, configuration, déploiement et exécution. La maintenance de l'application comporte généralement des opérations de reconfiguration et d'adaptation de composants. La reconfiguration consiste à changer les valeurs des propriétés des composants. L'adaptation consiste en général à remplacer un composant par un autre qui peut éventuellement fournir et requérir des contrats différents du composant initial. L'adaptation revient à déconnecter le composant de l'application, à transférer son état courant vers un nouveau composant et à connecter ce dernier à l'application. Traditionnellement, l'opération d'adaptation requiert généralement l'arrêt de l'application et oblige à reprendre le cycle de vie à partir de la phase d'assemblage.

L'adaptation peut être réalisée pour corriger le mauvais comportement d'un composant, pour répondre aux changements affectant l'environnement d'exécution, pour étendre l'application avec de nouvelles fonctionnalités ou pour améliorer ses performances. L'adaptation est un événement de plus en plus fréquent dans le cycle de vie d'une application. L'opération d'adaptation doit répondre à plusieurs contraintes comme la sécurité, la complétude et la possibilité de retour en arrière au cas où l'adaptation ne peut se terminer avec succès.

L'adaptation dynamique consiste à introduire des modifications dans l'application au cours de son exécution contrairement à l'approche traditionnelle qui nécessite l'arrêt total de l'application. Les principaux concepts liés au problème d'adaptation dynamique sont décrits plus en détail dans [Ket02a]

Open Services Gateway Initiative (OSGi) est une spécification ouverte [OSGI] qui définit les API de serveurs embarqués destinés à héberger des services qui peuvent être installés, activés, mis à jour, désactivés et désinstallés sans interruption de service du serveur embarqué. OSGi cible en premier lieu le domaine de

l'informatique embarquée dans des domaines variés comme la domotique, la gestion de bâtiments, les véhicules de transports, la santé, etc.

OSGi définit principalement le conditionnement (bundle) des services, les mécanismes de résolution des dépendances de code, le courtage des services actifs et l'activation de nouveaux services. Cependant, OSGi ne décrit pas l'architecture des applications ce qui rend difficile la vision globale d'applications constituées de plusieurs composants.

Nous pensons qu'OSGi est un bon point de départ pour construire une plate-forme à composants dynamiquement adaptables. Nous décrivons dans cet article deux propositions d'extension d'OSGi, la première permet de définir une plate-forme, appelée Beanome, permettant de définir des composants, de décrire l'architecture des applications, de déployer ces applications et de les administrer en cours d'exécution, la deuxième permet de réaliser de l'adaptation dynamique dans OSGi. La section suivante décrit la spécification OSGi. La section 3 présente le modèle à composants Beanome, la section 4 traite de l'adaptation dynamique des composants dans une application OSGi. Enfin, la conclusion présente les expériences menées avec cette plate-forme ainsi que les perspectives sur les compléments à mener.

2. Open Services Gateway Initiative

Open Services Gateway Initiative (OSGi) [OSGI][Che01] est une corporation indépendante qui a pour but de définir et de promouvoir des spécifications ouvertes pour la livraison de services administrables dans des réseaux résidentiels, véhiculaires et autres types d'environnements. OSGi définit la spécification d'une plate-forme de services qui inclut un modèle à composants minimal et un framework de petite taille pour administrer les composants. Les composants sont conditionnés dans un format spécifique. La sous-section suivante décrit plus en détail la spécification de la plate-forme. L'extensibilité, l'adaptabilité et l'évolution font l'objet des sections suivantes.

2.1 *Vue d'ensemble de la plate-forme*

La plate-forme de services OSGi se divise en deux parties : le framework OSGi d'une part et les services standards OSGi d'autre part. Dans cet article, nous nous intéresserons plus au framework qu'aux services standards, tels que le service HTTP. Dans le contexte de cet article, les services OSGi sont simplement des définitions d'interfaces Java avec une sémantique précise et tout objet qui implémente une interface de service est supposé obéir à son contrat.

Le framework OSGi définit un environnement hôte pour administrer des *bundles* ainsi que les services qu'ils fournissent. Un bundle est une unité physique de

livraison ainsi qu'un concept logique employé par le framework pour organiser son état interne. Concrètement, un bundle est une archive Java qui contient un manifeste et un ensemble de classes, des bibliothèques de code natif et d'autres ressources associées. Un bundle installé est identifiable de manière unique et il est important de noter qu'il n'est pas possible d'installer deux bundles à partir du même emplacement.

Les mécanismes d'administration fournis par le framework permettent l'installation, l'activation, la désactivation, la mise à jour et le retrait d'un bundle. Un bundle installé dans le framework peut avoir l'un des états suivants : *installé, résolu, actif, en démarrage, arrêté* ou *désinstallé*. L'état des bundles actifs est persistant tout au long des activations du framework, autrement dit, les bundles actifs préservent leur état après le redémarrage du framework.

Les méta-données d'un bundle sont décrites dans un fichier associé au JAR du bundle appelé manifeste. Le manifeste contient un ensemble de paires attribut-valeur. Certains attributs sont standardisés par la spécification OSGi. Ils décrivent le chemin des classes (*classpath*) du bundle, les packages Java exportés ou importés par le bundle, les besoins par rapport aux bibliothèques de code natif et l'information intelligible par l'administrateur du framework (e.g. nom du bundle, description).

Le manifeste peut également contenir un attribut qui spécifie une classe d'activation du bundle appelée « activateur » *(Activator)*. L'activateur joue un rôle important car il permet au bundle d'obtenir un *contexte* pour accéder aux fonctionnalités du framework. Le contexte permet aux bundles de rechercher des services dans le registre de services du framework, d'enregistrer leurs propres services, d'accéder à d'autres bundles et d'installer des bundles additionnels. La classe activateur enregistre chaque service avec au minimum le nom de l'interface qu'il implémente, et éventuellement des propriétés supplémentaires (version,…). La recherche de services se fait au moyen d'une requête LDAP simple qui peut limiter la recherche à l'ensemble des services qui ont des propriétés spécifiques (e.g. version>2.0). Une requête peut retourner zéro ou plusieurs références de services.

Le bundle ne requiert pas obligatoirement une classe activateur. Il peut être simplement une bibliothèque de packages Java qui ne nécessitent pas l'accès au framework.

2.2 *Limitations dans OSGi*

Dans OSGi, les bundles sont utilisés pour conditionner et livrer les services avec leurs implémentations et leurs ressources. Les services sont les unités principales de construction d'applications dans le framework OSGi, et la plupart des services OSGi sont construits en séparant l'interface de son implémentation. Ceci a comme résultat que les applications OSGi sont construites exclusivement à partir de ces interfaces et qu'elles n'ont aucune connaissance de leurs implémentations. De plus, les

applications sont construites de façon dynamique, au moment où de nouveaux services apparaissent ou disparaissent.

Bien que OSGi introduise un modèle différent de programmation, c'est-à-dire au moyen de services, ceci peut se révéler complexe pour plusieurs raisons :

- Il n'y a pas de moyens pour décrire une architecture comme un ensemble d'instances de composants connectées les unes aux autres.
- Les services sont essentiellement différents des instances de composants communes aux modèles 'classiques' dans le sens qu'ils s'agissent d'objets partagés qui, de préférence ne possèdent pas d'état (du fait que le changement de cet état par un des clients affecte tous les autres clients).
- Les dépendances entre services ne sont pas gérées par le framework OSGi. Il est possible qu'un service nécessite un autre pour fonctionner, et de plus cette dépendance peut être statique (au démarrage du service) ou dynamique (le service a déjà été enregistré). Ceci doit être programmé dans l'activateur du bundle, ce qui peut être une tâche complexe à réaliser et difficile à changer.

Bien que OSGi fournisse des mécanismes très efficaces pour gérer le déploiement des unités de livraison de services, il ne peut être considéré comme un modèle à composants 'complet' car plusieurs concepts communs aux modèles à composants ne sont pas présents dans celui-ci. Nous pouvons citer en particulier le concept de type de composant, à partir duquel des instances sont créées. La section suivante introduit Beanome, un modèle à composants simple qui a pour objectif l'enrichissement d'OSGi avec les concepts manquants. Beanome constitue alors un modèle à composants au dessus de la plate-forme OSGi.

3. Beanome, une plate-forme à composants adaptables pour OSGi

Beanome [Cer02] ajoute une couche fine au dessus du framework standard OSGi. Il offre des fonctionnalités similaires à celles des modèles à composants existants. Beanome définit un modèle à composants simple et un framework pour supporter ce modèle. Les sections suivantes présentent les caractéristiques principales de Beanome puis positionnent Beanome par rapport à d'autres modèles à composants existants actuellement.

3.1 *Le modèle à composants Beanome*

Le modèle à composants Beanome s'appuie sur le concept de *types* de composants (voir Figure 1). Un type de composant est employé comme un modèle pour créer des instances des composants. La structure d'un type de composant est

définie dans un fichier appelé descripteur de composant. Plusieurs types de composants peuvent être décrits dans le même descripteur.

Le type de composant Beanome le plus simple est constitué d'une interface et d'une classe Java qui implémente cette interface. Dans ce contexte, le terme *interface* est abstrait et ne fait pas référence seulement aux interfaces, mais aussi aux classes Java. Un type de composant peut fournir plus d'une interface et chacune d'entre elles peut être implémentée par une classe différente. Les classes implémentant un type de composant particulier peuvent être connectées entre elles. Le descripteur du composant permet par ailleurs de décrire directement les noms des classes qui implémentent les interfaces d'un type de composant ou de déléguer vers une classe particulière la responsabilité de fournir une implémentation pour une interface particulière. Ceci est également valable pour les connections. L'avantage de cette approche est qu'elle rend possible de décrire des composants simples directement dans le fichier et que des choix d'implémentation ou de connexion plus complexes sont aussi supportés sans avoir besoin d'étendre les fichiers descripteurs de composants. Finalement, les types de composants sont identifiés par un nom unique et ont un numéro de version associée.

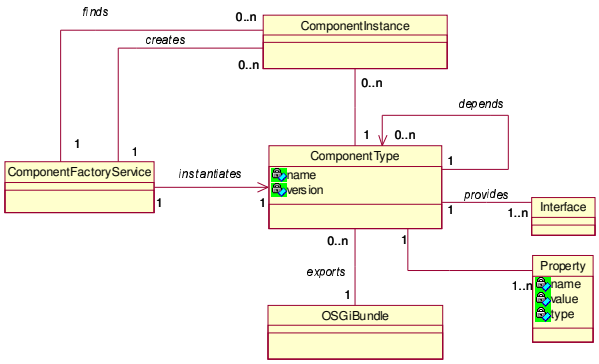

Figure1. *Modèle de Beanome*

3.2 *Framework de composants de Beanome*

Le framework de composants de Beanome fournit, à l'exécution, les fonctionnalités nécessaires au support du modèle à composants de Beanome. Le framework de composants, appelé *runtime*, se base sur deux concepts principaux : les fabriques de composants et le registre de composants. Le registre de composants maintient la liste de toutes les fabriques disponibles lors de l'exécution. Une fabrique de composants est associée à un type de composant spécifique et implémente le patron *factory method* [Gam95] permettant de créer des instances de composants. Les fabriques de composants fournissent aussi des méthodes appelées *finder* qui

permettent de récupérer des instances de composants pouvant être partagées et pouvant être persistantes. Le *runtime* permet l'accès au registre de composants et aux mécanismes d'introspection sur les types de composant et sur leurs instances.

Pour créer une instance à partir d'un type de composant, le *runtime* récupère la fabrique correspondante au type du composant en question à partir du registre. Le *runtime* crée alors des objets à partir des classes qui implémentent les interfaces fournies par le type du composant. Si le type du composant dépend d'autres types, il est possible qu'au moment de l'instantiation, ces derniers soient créés. Le *runtime* doit alors récursivement trouver les fabriques des composants dont le type de composant dépend et de les instancier. Si le *runtime* n'est pas capable de résoudre les dépendances, l'instantiation échoue. Il est à noter que les clauses 'requires' peuvent être complémentées par un nom d'une instance partagée, ce qui voudra dire qu'une instance de composant contiendra une référence vers une instance partagée.

3.3 *Liaison vers OSGi*

Le *runtime* de Beanome est implémenté comme un bundle OSGi. Les bundles sont utilisés pour le packaging d'un ou de plusieurs types de composants Beanome. Du fait qu'une fabrique peut être vue simplement comme un service de création d'instances, les bundles contentant des composants Beanome enregistrent les fabriques de composants Beanome à travers des services dans le registre du framework d'OSGi. Pour cela, une interface identifiant un service de fabrique, appelée `ComponentFactoryService` à été conçue. Ce service permet de créer des instances ainsi que de faire des recherches d'instances partagées (ce qui est l'équivalent du *home* dans le modèle EJB ou CCM). Pour automatiser la tâche d'enregistrement des fabriques, un activateur générique est exporté par le bundle du *runtime* de Beanome. Cet activateur générique parcourt les fichiers descripteurs de types de composants et enregistre automatiquement un service de type pour chaque type de composant dans le fichier.

L'activateur générique enregistre des fabriques en utilisant les mécanismes standard d'OSGi pour attacher à chaque service de fabrique un ensemble de propriétés qui incluent le nom du type du composant, le numéro de version associé et l'ensemble d'interfaces que le composant fournit. La liste des interfaces fournies est nécessaire car un composant peut spécifier sa dépendance vers un autre uniquement à partir d'un ensemble d'interfaces, sans prendre en compte le nom ou la version du composant qui les fournit.

Si un type de composant dépend d'un autre (il existe une clause 'requires' dans le descripteur du composant), la dépendance doit être résolue au moment de l'instantiation des composants. L'expression de ces dépendances peut être faite sous forme de requêtes LDAP. Les requêtes sont écrites en termes de propriétés de services de fabriques de composants, telles que le nom ou la version. Ceci permet d'avoir un système flexible quant à la résolution de dépendances entre composants.

3.4 *Beanome par rapport a d'autres modèles à composants*

Le modèle à composants Beanome permet de rajouter une couche au dessus du modèle proposé par OSGi afin de disposer des concepts nécessaires pour avoir un modèle à composants. Il est important de noter que Beanome n'est pas un modèle à composants orienté vers la construction de systèmes distribués mais permet plutôt de construire des applications à base de composants à l'intérieur du framework OSGi qui n'est pas distribué non plus.

Le modèle à composants Beanome peut être comparé au modèle COM de Microsoft [COM] du fait qu'un type de composant peut afficher plusieurs interfaces, le framework Beanome fournit d'ailleurs un mécanisme de navigation similaire au *QueryInterface* de COM. Les composants sont enregistrés dans le registre OSGi, de façon similaire aux composants COM qui sont enregistrés dans le registry de Windows. A la différence de COM, les types de composant Beanome décrivent explicitement leurs dépendances vers d'autres types de composants. Ceci peut être comparé aux références dans le modèle EJB (`<ejb-ref>`), dans lequel on spécifie qu'un composant EJB nécessite un autre composant EJB. Les propriétés définies au niveau du type du composant sont accessibles depuis leurs instances, ce qui permet d'avoir un moyen de configuration externe.

Par rapport à CCM, Beanome ne propose pas la possibilité d'avoir des instances multiples d'une même interface (facettes) et le concept de réceptacle n'est pas disponible au niveau de la description des types de composants Beanome. Les réceptacles dans CCM permettent d'expliciter des points de connexion au niveau instance qui sont ensuite nécessaires pour créer un assemblage. Dans Beanome (comme dans COM ou dans EJB), les assemblages se font dans le code d'implémentation des composants. Finalement, Beanome ne dispose pas pour le moment du concept de *container.*

Beanome introduit deux différences importantes par rapport aux modèles à composants dits 'classiques' :

- La localisation de composants à base de filtres : Dans COM les implémentations d'un composant sont identifiées de façon unique grâce à un CLSID. Dans Beanome, la localisation d'un composant dont on dépend est réalisée à partir d'un filtre (spécifié dans la clause requires) qui permet de faire une recherche soit précise (en spécifiant un nom et un numéro de version par exemple) ou bien plus ouverte (en spécifiant les interfaces que le composant doit implémenter par exemple).
- La dynamicité au niveau des unités de déploiement : Du fait que Beanome est construit au dessus d'OSGi, il dispose de mécanismes offerts par celui-ci pour l'administration des unités de déploiement (bundles). Ceci résulte du fait que les fabriques de composants peuvent être enregistrées (ou desenregistrées) à tout moment à l'exécution de l'application. Ceci a

l'avantage de permettre par exemple de faire l'installation de composants à la demande, ou bien la mise à jour de ceux ci.

Dans cette section, nous avons introduit le modèle à composants Beanome en le situant par rapport à certains modèles existants. Beanome bénéficie de l'infrastructure OSGi qui lui permet de faire l'enregistrement dynamique de composants. Ces mécanismes sont très importants dans le contexte d'applications qui doivent tourner en continu. Cependant, certains problèmes restent à explorer, en particulier comment il faut gérer les instances d'un composant existantes dans une application dans le cas de la disparition de leur fabrique à cause d'un arrêt ou d'une mise à jour du bundle qui exporte cette fabrique.

Pour le moment, Beanome ne dispose pas du concept de *container* présent dans d'autres modèles, du fait que les aspects non fonctionnels n'ont pas été introduits dans le modèle. Pour réponde à certains besoins et résoudre certains problèmes comme celui énoncé précédemment, il s'avère nécessaire d'introduire le concept de container dans Beanome. Ce concept permet de bénéficier de certains mécanismes comme l'adaptation dynamique des instances au moment d'une mise à jour d'un type de composant. La section suivante présente d'une manière détaillée les différents concepts liés à l'adaptation dynamique d'un composant.

4. Adaptation dynamique dans OSGi

Plusieurs approches d'adaptation (DCUP [Pla97], OLAN [Bel99],...) définissent un modèle à composants où chaque composant est constitué d'une partie fonctionnelle et d'une partie de contrôle. Cette dernière partie regroupe la liste des opérations permettant de gérer le cycle de vie du composant (arrêt, passivation, mise à jour, …). Une telle approche est convenable lorsque l'on définit de nouveaux modèles. Par contre, elle ne peut pas être appliquée aux modèles existants qui n'ont pas été conçus pour supporter l'adaptation dynamique.

Notre approche consiste à séparer la partie contrôle de la partie applicative afin que les solutions proposées soient indépendantes des modèles et de leurs capacités à supporter l'adaptation. Les sections suivantes présentent les concepts liés au processus d'adaptation. Ces concepts sont illustrés en s'appuyant sur l'exemple présenté dans la figure 2 qui présente un Portail Web constitué de plusieurs composants : un serveur http, un moteur de portail, un gestionnaire des profils des usagers et des gestionnaires de contenu (portefeuille boursier, météo, mail, …). Chaque instance de ces composants est principalement associée à une session d'un usager. Dans cette application à haute disponibilité, l'administrateur doit pouvoir adapter un composant par un nouveau composant sans interruption de service du portail. Le nouveau composant peut fournir la même interface (i.e. mise à jour de l'implémentation) ou une interface différente (i.e. intégration d'un composant patrimonial (*legacy*)). L'adaptation peut alors se faire au niveau du modèle et donc être appliquée à toutes les instances existantes du composant ou alors appliquée à

une instance particulière. Dans les deux cas, chaque état d'une instance adaptée doit être transféré vers une nouvelle instance du nouveau composant.

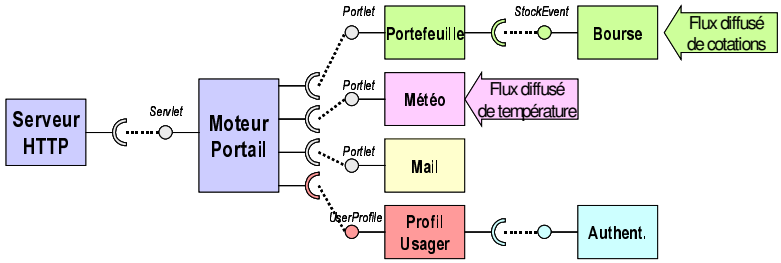

Figure 2. *Exemple d'application à haute disponibilité sur Beanome*

4.1. *Processus d'adaptation*

Le processus d'adaptation que nous avons pu expérimenter à la fois sur les JavaBeans [Ket02b] et sur OSGi est constitué de plusieurs étapes :

1. Définition des règles de correspondance entre l'ancien et le nouveau composant par l'administrateur.
2. Création d'une instance du nouveau composant.
3. Passivation des deux instances.
4. Transfert de l'état de l'ancienne instance vers la nouvelle selon les règles de correspondance définies dans la première étape.
5. Déconnexion de l'ancienne instance et connexion de la nouvelle.
6. Activation de la nouvelle instance.

Ces activités forment les briques de base pour une machine d'adaptation.

4.2. *Technique appliquée*

Afin de pouvoir adapter un composant en cours d'exécution de l'application, il est nécessaire de pouvoir contrôler toutes ses communications entrantes et sortantes (appels depuis et vers le composant). Dans le cas où le composant définit une partie de contrôle, cette communication peut être facilement contrôlée car le composant participe lui-même à sa propre adaptation. Dans le cas contraire, il faut éviter la communication directe entre les objets. Il est donc nécessaire d'intercepter les appels par un objet *proxy* qui les redirige vers les objets concernés. Cependant, comme cette redirection peut coûter cher en terme de performance, la technique n'est appliquée qu'aux composants susceptibles d'être adaptés à l'exécution.

Dans ce travail, nous avons utilisé les *proxys dynamiques* pour relier les composants. Cependant, il est possible d'utiliser d'autres types de proxys comme les proxys statiques générés à la compilation ou à la volée et les RMI locaux. Dans l'exemple présenté dans la figure 2, nous supposons que le composant *Portefeuille* est susceptible d'être adapté, pour cette raison, sa connexion avec le composant

MoteurPortail ne doit pas être directe. Comme le présente la figure 3, le composant *MoteurPortail* qui a besoin d'un service fourni par le composant *Portefeuille* doit le demander non pas au framework mais à la machine d'adaptation dynamique.

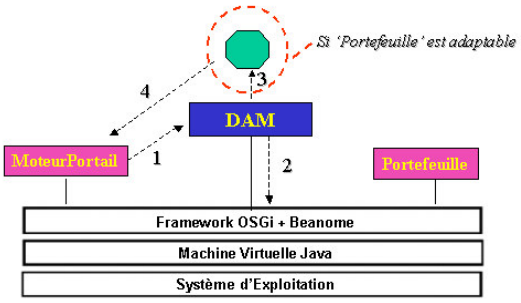

Figure 3. *Demande de services*

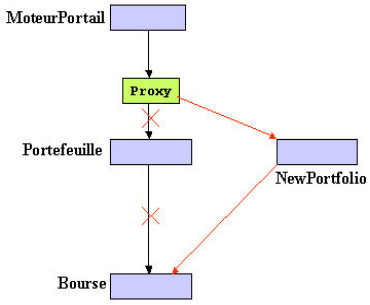

Figure 4. *Reconnexion dynamique*

Si le service est fourni par un composant non adaptable, la référence de ce service est rendue au client et la communication se fait d'une manière directe. Dans le cas contraire, un proxy implémentant les interfaces fournies par le composant fournissant le service (*Portefeuille*) est créé et sa référence est rendue au client (*MoteurPortail)*. La communication transite alors par ce proxy où certains besoins d'adaptation sont définis (passivation, reconnexion,…).

4.3. *Règles de correspondance*

Un composant Beanome est potentiellement constitué d'un ensemble de classes. Cet ensemble peut complètement différer entre l'ancien et le nouveau composant. Il est donc nécessaire de définir des règles de correspondance entre les deux composants entre leurs propriétés d'une part, et entre les services d'autre part.

- Propriétés : une propriété du nouveau composant peut correspondre à une ou plusieurs propriétés de l'ancien. La correspondance de propriétés est nécessaire pour le transfert d'état. Par exemple, la propriété *dernierCours* du composant *Portefeuille* correspond à la propriété *lastTrade* du nouveau composant *NewPortfolio*.
- Services : un service fourni par le nouveau composant peut correspondre à un ou plusieurs services fournis par l'ancien. La correspondance des services est indispensable pour les reconnections dynamiques. Dans l'exemple précédent, supposons que le service *graphique(int code, Date debut, Date fin)* fourni par le composant *Portefeuille* correspond au service *getChart(String symbol , Period p)* du composant *NewPortfolio*.

Notre machine d'adaptation n'implémente pour l'instant qu'une mise en correspondance un vers un pour les propriétés de même type et les méthodes de même signature.

4.4. *Passivation et activation d'un composant*

Pour des raisons de cohérence, un composant doit être mis à l'état *passif* avant d'être adapté. Ce qui signifie qu'aucune requête ne peut être servie par le composant. Ces requêtes sont mises en attente ou non selon les stratégies appliquées. A titre d'exemple, le composant *Portefeuille* doit être mis à l'état passif avant qu'il soit adapté vers une nouvelle version. La mise en attente des requêtes dépend aussi du type de communication entre les composants. Généralement, les communications asynchrone (par événements) sont plus faciles à gérer les communications synchrones (appel de méthodes).

Dans le cadre des composants OSGi, les communications sont des appels de méthodes et deux stratégies peuvent être appliquées lorsqu'un composant est mis à l'état passif. La première consiste à retourner une réponse informant que le serveur est temporaire indisponible. Cette réponse peut être une exception d'un type semblable à *java.rmi.RemoteException*. La deuxième stratégie quant-à-elle consiste à mettre en file d'attente les requêtes en suspendant les threads associés à ces requêtes. Les threads sont réveillés successivement après l'activation du nouveau composant.

4.5. *Transfert d'état*

Le nouveau composant doit démarrer à partir de l'état de l'ancien. Les valeurs des propriétés sont transférées selon les règles de correspondance précédemment définies. L'état d'un composant OSGi est représenté par l'ensemble des valeurs des propriétés de tous les objets constituant le composant, et ayant des correspondances dans le nouveau composant. Dans l'état actuel, le transfert ne concerne que les propriétés publiques. Par exemple, l'état du composant *Portefeuille* consiste en sa propriété *dernierCours*. La valeur de cette propriété est transférée vers la propriété *lastTrade* du nouveau composant *NewPortfolio*.

4.6. *Reconnexions dynamiques*

La reconnexion peut être réalisée à deux niveaux de granularité : *composant* ou *méthode*. Le niveau composant nécessite la reconnexion de toutes les méthodes fournies par le composant. Le niveau méthode ne demande que la reconnexion d'une méthode particulière.

Comme l'illustre la figure 4, le composant *Portefeuille* fournit un service à *MoteurPortail* et utilise un service fourni par *Bourse*. *Portefeuille* est adapté par *NewPortfolio.* Pour établir la reconnexion, la référence de *Portefeuille* dans le proxy *Proxy* est remplacée par la référence de *NewPortfolio*, également, les appels des anciennes méthodes de *Portefeuille* sont adaptés aux nouvelles méthodes de *NewPortfolio* selon les règles de correspondance des services. Dans l'exemple, la connexion entre le composant *Portefeuille* et le composant *Bourse* est directe en supposant que ce dernier n'est pas adaptable.

5. Conclusions et Perspectives

Le travail présenté dans ce papier test orienté autour de deux propositions pour étendre le framework OSGi. OSGi est une plate-forme qui accueille et administre des unités de déploiement appelés bundles et fournit un registre où les bundles peuvent enregistrer et récupérer des services. Les applications dans le framework sont construites dynamiquement a partir des services qui peuvent apparaître ou disparaître a tout moment en conséquence de changements d'état dans les bundles.

La première des propositions présentées décrit une couche permettant de définir un modèle à composants, appelé Beanome, proche des modèles 'standard' (COM, EJB, CCM) au dessus d'OSGi. Cette proposition permet de résoudre certaines limitation introduites par le modèle OSGi, en particulier le manque d'un concept d'instance de composants. Les bundles sont employés comme des unités de livraison des composants OSGi et enregistrent des fabriques de composants dans le registre de services du framework.

La deuxième proposition décrit le moyen de réaliser l'adaptation (substitution avec transfert d'état) en cours d'exécution d'une instance de composant par une autre, qui peut posséder éventuellement une interface différente. Pour réaliser l'adaptation dynamique, il est nécessaire de mettre en place des mécanismes permettant de réaliser la passivation, l'activation et le transfert d'état des instances de composant.

Actuellement il existe une implémentation du modèle à composants Beanome [BWeb] ainsi que d'une machine d'adaptation dynamique permettant de faire des substitutions au niveau des services OSGi et qui a aussi été testée dans le modèle JavaBeans. Ces deux prototypes ont été implémentés sans avoir recours à des modifications au niveau du code du framework OSGi, mais ils sont plutôt installables comme des bundles standard dans le framework.

Le fait d'introduire un modèle à composants au dessus d'une plate-forme de déploiement dynamique comme l'est OSGi a pour conséquence que les fabriques des composants peuvent disparaître ou être mises à jour lors de l'exécution de l'application. Ceci implique que les instances existantes dans une application doivent être détruites ou mises à jour. Dans le deuxième cas, la machine d'adaptation dynamique révèle être une solution idéale pour résoudre le problème des mises a jour au niveau des instances. La machine d'adaptation dynamique nécessite cependant l'emploi d'intermédiaires autour des instances pour pouvoir réaliser les substitutions, et ce mécanisme n'est actuellement pas disponible dans le modèle OSGi. Ce besoin peut être résolu en introduisant le concept de container dans le modèle Beanome.

Les travaux en cours portent sur l'ajout du concept de container dans le modèle Beanome. Ce concept doit être implémenté pour pouvoir réaliser une fusion des deux approches décrites dans ce papier.

6. Bibliographie

[Bel99] Bellissard L., De Palma N., Freyssinet A., Herrmann M., Lacourte S., *An Agent platform for Reliable Asynchronous Distributed Programming* , Actes du Symposium on Reliable Distributed Systems (SRDS'99), Lausanne, Suisse, October 1999.

[BWeb] BeanomeWeb : www-adele.imag.fr/BEANOME

[Cer02] Cervantes H., Hall R.S., *Beanome a component model for the OSGi framework», soumis a Software Infrastructures for Component-Based Applications on Consumer Devices*, September 2002, Lausanne, Switzerland

[Che01] Chen K., Gong L., *Programming Open Service Gateways with Java Embedded Server Technology*, Pub. Addison Wesley, August 2001 ISBN#: 0201711028.

[COM] Microsoft Corp., *Component Object Model* http://www.microsoft.com/COM

[Gam95] Gamma, E., et al., *Design Patterns - Elements of Reusable Object-Oriented Software*, Ed. Addison Wesley, 1995.

[Ket02a] Ketfi M., Belkhatir N., Cunin P.Y., Automatic *Adaptation of Component-based Software : Issues and Experiences*, accepté à PDPTA'02, Juin 2002, Las Vegas, Nevada, USA, http://www-adele.imag.fr/Les.Publications/BD/PDPTA2002Ket.html

[Ket02b] Ketfi M., Belkhatir N., Cunin P.Y., *Dynamic updating of component-based applications*, accepté à SERP'02, Juin 2002, Las Vegas, Nevada, USA, http://www-adele.imag.fr/Les.Publications/BD/SERP2002Ket.html

[OMG99]Object Management Group, *CORBA Components: Joint Revised Submission*, August 1999.

[OSGI] Open Services Gateway Initiative, *OSGi service gateway specification*, Release 2, October 2001, http://www.osgi.org

[Pla97] Plasil, F., Balek, D., Janecek, R, *DCUP: Dynamic Component Updating in Java/CORBA Environment*, Tech. Report No. 97/10, Dep. of SW Engineering, Charles University, Prague, 1997.

[Riv00] Riveill M., Merle P., *Programmation par composants*, Techniques de l'Ingénieur, Traité Informatique, 2000.

[Rom99] Roman E., *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*, Ed. Wiley, 1999

[Sun01] Sun Microsystems, *Enterprise JavaBeans Specification*, Version 2.0, 2001.

[Sun97] Sun Microsystems, « JavaBeans Specification », Version 1.01, 1997.

[Tha01] Thai T., Lam H.Q., *.NET Framework Essentials*, Ed. O'Reilly & Associates, 2001.